

API and Database

A language-agnostic application programming interface (API) allows for broad and diverse integration. Within the Clarity LIMS Rapid Scripting API, REST technology is used to provide data specifically structured for life science research. This section provides information about the Rapid Scripting API, and examples that demonstrate how to work with REST and automation scripts.

API Portal

Together, REST and External Program Integration Plug-ins (EPP)/automation provide powerful and simple-to-use scripting. Before working with the REST API, understand the conceptual structure and design of these interfaces. The links below provide overview information to help you get started, a self-training Cookbook guide with example scripts, and videos that supplement the API training materials.

- [REST General Concepts](#)
- [Getting Started with the API](#)
- [About the Cookbook](#)
- [API Training Videos](#)

Current Version:

[REST API v2 r33](#)—Introduced in BaseSpace Clarity LIMS v6.2.

Previous API Versions

Clarity LIMS Version	API Version
6.1	v2 r32
6.0	v2 r31
5.4	v2 r29
5.3	v2 r28
5.2	v2 r27
5.1	v2 r26
5.0	v2 r25
4.3	v2 r24.1
4.2	v2 r24
4.1	v2 r23
4.0	v2 r22

Clarity LIMS Version	API Version
3.5	v2 r21
3.4	v2 r20
3.1	v2 r19
3.0	v2 r18
2.5	v2 r17
2.4	v2 r16

Overview

The internal Clarity LIMS API (eg, <https://example.claritylims.com/clarity/api>) is the API used to deliver the Clarity LIMS web interface. This interface is not typically meant for public consumption. However, some customers use it for troubleshooting and to mitigate system issues.

Preventing CSRF Attacks

As of Clarity LIMS v5.1, access to the internal Clarity LIMS API changed to enhance security and prevent Cross Site Request Forgery (CSRF) attacks. Two new HTTP headers must now be present when issuing PUT, POST, DELETE, and PATCH requests:

- **Origin**—This header must be set to the scheme and authority of the server being accessed (eg, <https://example.claritylims.com>).
- **X-Requested-With**—This header must be set to XMLHttpRequest.

The attached cURL, Python, and Java examples demonstrate how to authenticate and issue internal API requests. These examples assume a Clarity LIMS server at <https://example.claritylims.com>.

- [csrf_headers.sh.txt](#)
- [csrf_headers.py.txt](#)
- [csrf_headers.java.txt](#)

REST General Concepts

REST and automation are the key interfaces for scripting. A language-agnostic application programming interface (API) is important to scientists as it allows for broad and diverse integration. Together, REST and automation provide powerful and easy-to-use scripting. However, you first need to understand the conceptual structure and design of these interfaces.

Within the Clarity LIMS Rapid Scripting API, REST technology is used to provide data specifically structured for life science research.



The API documentation includes the terms External Program Integration Plug-in (EPP) and EPP node.

As of BaseSpace Clarity LIMS v5.0, these terms are deprecated. The term EPP has been replaced with automation. EPP node is referred to as the Automation Worker or Automation Worker node. These components are used to trigger and run scripts, typically after lab activities are recorded in the LIMS.



If you are new to the REST Web Service, we recommend that you read [Development Prerequisites](#) and [REST Web Services](#).

REST: A Web Service for Data Access

The REST Web Service is the fundamental data access interface using XML over HTTP. It is agnostic to programming languages as most languages support HTTP and XML with libraries or built-in methods.

In life science research labs, tracking samples and the data associated with biology, research, and lab work is complex. The REST resources return information that is human-readable and interpretable. Use a web browser to explore the XML returned.

REST represents real laboratory items and activities in self-contained groups of data called resources. It provides access to recorded lab steps and to sample test results, and it provides this access using resources. For example:

- The process and steps resources track the steps in the lab in terms of who did what and when.
- The sample and artifact resources contain information on the submitted sample and test results on sample derivatives (also referred to as derived samples).

The REST resources and their relationships are explained in [Structure of REST Resources](#).

For full details of each resource, see the [API Portal](#).

General Concepts

Requests are made to the API by sending XML messages:

- POST is used to create an item.
- GET is used to read an item.
- PUT is used to update an item.
- DELETE is used to delete an item.

 HEAD requests are not supported.

The full URL to which requests should be sent varies depending on the specific installation, but it generally follows this format:

```
http[s]://<hostname>:<port>/api/<version.of.api>
```

Automation/EPP: GUI Trigger for Calling Scripts

Automation/EPP is used to trigger scripts from within the Clarity LIMS interface.

Script triggering is often used because the data collected needs to be dispatched for further processing. Automating data processing and returning information to the lab for immediate use, in the appropriate format, increases efficiency and quality.

File handling and file management are fundamental elements in life science scripting. When triggered, scripts can issue a command, transfer files for processing, and collect and transfer files back to the server. To enable triggering of scripts in any programming language, the information and files are provided for batch processing at the operating system command-line level.

For BaseSpace Clarity LIMS (earlier than v5):

To use automation, administrators complete the following steps.

1. Create and configure master steps/processes with automations/EPP.
2. Write scripts that are triggered by these automations/EPP.
3. In the Clarity LIMS web interface, use the configured master steps/processes as building blocks to create and configure steps to be run by lab scientists.

For BaseSpace Clarity LIMS (v5 and later):

 As of Clarity LIMS v5, the Operations Interface Java client, which was used by administrators to configure processes, consumables, user-defined fields, and users, has been deprecated. All configuration and administration tasks are now executed in the Clarity LIMS web interface.

To use automation, administrators complete the following steps.

1. In Clarity LIMS, create and configure master steps.
2. Configure automations that trigger scripts. Enable those automations on the master steps.
3. Use the configured master steps as building blocks to create and configure steps to be run by lab scientists.

Related Resources

- [Automation Overview](#)
- [Work with EPP/Automation and Files](#)

REST General Concepts

REST and automation are the key interfaces for scripting. A language-agnostic application programming interface (API) is important to scientists as it allows for broad and diverse integration. Together, REST and automation provide powerful and easy-to-use scripting. However, you first need to understand the conceptual structure and design of these interfaces.

Within the Clarity LIMS Rapid Scripting API, REST technology is used to provide data specifically structured for life science research.

 The API documentation includes the terms External Program Integration Plug-in (EPP) and EPP node.

As of BaseSpace Clarity LIMS v5.0, these terms are deprecated. The term EPP has been replaced with automation. EPP node is referred to as the Automation Worker or Automation Worker node. These components are used to trigger and run scripts, typically after lab activities are recorded in the LIMS.

i | If you are new to the REST Web Service, we recommend that you read [Development Prerequisites](#) and [REST Web Services](#).

REST: A Web Service for Data Access

The REST Web Service is the fundamental data access interface using XML over HTTP. It is agnostic to programming languages as most languages support HTTP and XML with libraries or built-in methods.

In life science research labs, tracking samples and the data associated with biology, research, and lab work is complex. The REST resources return information that is human-readable and interpretable. Use a web browser to explore the XML returned.

REST represents real laboratory items and activities in self-contained groups of data called resources. It provides access to recorded lab steps and to sample test results, and it provides this access using resources. For example:

- The process and steps resources track the steps in the lab in terms of who did what and when.
- The sample and artifact resources contain information on the submitted sample and test results on sample derivatives (also referred to as derived samples).

The REST resources and their relationships are explained in [Structure of REST Resources](#).

For full details of each resource, see the [API Portal](#).

General Concepts

Requests are made to the API by sending XML messages:

- POST is used to create an item.
- GET is used to read an item.
- PUT is used to update an item.
- DELETE is used to delete an item.

i | HEAD requests are not supported.

The full URL to which requests should be sent varies depending on the specific installation, but it generally follows this format:

```
http[s]://<hostname>:<port>/api/<version.of.api>
```

Automation/EPP: GUI Trigger for Calling Scripts

Automation/EPP is used to trigger scripts from within the Clarity LIMS interface.

Script triggering is often used because the data collected needs to be dispatched for further processing. Automating data processing and returning information to the lab for immediate use, in the appropriate format, increases efficiency and quality.

File handling and file management are fundamental elements in life science scripting. When triggered, scripts can issue a command, transfer files for processing, and collect and transfer files back to the server. To enable triggering of scripts in any programming language, the information and files are provided for batch processing at the operating system command-line level.

For BaseSpace Clarity LIMS (earlier than v5):

To use automation, administrators complete the following steps.

1. Create and configure master steps/processes with automations/EPP.
2. Write scripts that are triggered by these automations/EPP.
3. In the Clarity LIMS web interface, use the configured master steps/processes as building blocks to create and configure steps to be run by lab scientists.

For BaseSpace Clarity LIMS (v5 and later):

 | As of Clarity LIMS v5, the Operations Interface Java client, which was used by administrators to configure processes, consumables, user-defined fields, and users, has been deprecated. All configuration and administration tasks are now executed in the Clarity LIMS web interface.

To use automation, administrators complete the following steps.

1. In Clarity LIMS, create and configure master steps.
2. Configure automations that trigger scripts. Enable those automations on the master steps.
3. Use the configured master steps as building blocks to create and configure steps to be run by lab scientists.

Related Resources

- [Automation Overview](#)
- [Work with EPP/Automation and Files](#)

REST Web Services

The Clarity LIMS Rapid Scripting™ API provides scientific programmers with self-descriptive, yet flexible, data access. It uses a RESTful model for data access because this model is well suited to these requirements. This article provides a high-level introduction to REST concepts and technologies.

Introduction to REST Web Service Technology

Representational State Transfer (REST) is a style of software architecture for distributed information retrieval systems, most commonly observed by using the web.

- REST governs proper behavior. It is not a methodology or a design principal, but rather a set of rules to which a system should conform.
- REST allows a uniform interface between clients and servers that is simple and decoupled, enabling each system to evolve independently.

- REST is referred to as stateless because each new API request contains all the information required to complete it without relying on previous requests. Conforming to these REST principals is referred to as being RESTful.
- REST was developed in parallel with HTTP and makes use of this protocol. It is an elegant way to programmatically access resources over HTTP. It is flexible because you can use it with any language or tool that supports HTTP.

The web is probably the largest known RESTful system. Its behavior is simple:

1. When selecting a link in a web browser, the system requests information by sending a GET request to the specified URL. This URL is a resource.
2. The server that hosts the URL responds, typically with one of two things:
 - If the page exists, the server sends the browser an HTTP 200 response code and the contents of the page.
 - If the page does not exist, the server sends an HTTP 404 response code and an error message indicating that the page cannot be found.

Many software development groups use RESTful APIs. Google, Yahoo, and many public websites use the RESTful model for information access.

Communicating with the REST API

The REST API allows you to retrieve and update information using HTTP operations. This ability provides some flexibility in how to communicate with the system.

While REST requests and responses can be in various formats, we chose XML. Each resource and XML element is detailed in the [REST API Documentation](#).

To use the REST API, sign in using HTTP BASIC authentication. The method used to authenticate depends on how you use the API:

- When using a browser to retrieve information from the API, sign in to the browser with a user name and password. When signing in using a browser, the session remains open until the browser is closed.
- When using an HTTP request tool to retrieve, add, update, or remove data using the API, the tool asks for a user name and password each time you submit a request to the system.
- When using a script to communicate with the API, the script must first authenticate with the API. The session remains open for as long as the script is being actively read by the system.

The account you use to sign in to the API must have System Administrator or Facility Administrator privileges.

The API allows self-discovery of an object. When you request information about an object, the system typically returns URIs to its children and, sometimes, its parent. Use one URI to find the next URI in a hierarchy.

When viewing XML in a browser, tools can automatically create links from the URIs returned by the system. Examples of such tools are the Firefox [Text Link](#) or [Linkificator](#) add-ons. This way, you can select URIs to browse through the API.

Requests are made to the API by sending XML in HTTP calls:

- GET is used to read an item.
- POST to create an item.
- DELETE to delete an item.
- PUT is used to update an item.

In its simplest form, use a browser to enter and read the content of a URI, which allows browsing through the system. When using this method, a GET request is issued to the API for a specified object (referred to as a resource). The request returns XML containing the metadata about that resource. See the following section for details.

If you want to add, update, and delete small amounts of data using the API, use an HTTP request tool, such as the Firefox [RESTClient](#) add-on.

Resources and Namespaces

When working with REST, there are references to resources and namespaces.

For references:

A RESTful API groups related information into resources, each of which is referenced with a global identifier (URI).

In the API, for example, every sample in the LIMS has a resource for its information. When scripting, the resource is created or updated with POST and PUT HTTP calls. There are two types of resources: single and list types.

- A list resource is used to access a collection of single resources (such as a listing of all samples).
- The single resource type is used to access details on just one resource (a sample, for example).

It's important to understand how the information in the LIMS has been grouped and structured into resources. To learn more, see [Structure of REST Resources](#).

For namespaces:

An API that uses XML relies on namespaces. In XML, namespaces define the vocabulary of elements and attributes in an XML document. Each REST resource references the XML structure defined by a particular namespace.

When scripting, we use namespaces to look up specific details related to the XML data elements, attributes, and formats that represent a resource. Namespaces also order the subelements of the XML document.

In the current revisions of the API, the PUT and POST methods read the subelements of the XML independent of order, but the namespace still defines the order of the XML provided in GET calls.

Additional Information

For sophisticated write operations and automation of work, you must use a script to communicate with the API. The [About the Cookbook](#) contains examples that demonstrate how to use scripts to perform your work.

HTTP Response Codes and Errors

The REST API methods attempt to return appropriate HTTP status codes for every request. To use the REST API effectively, a good understanding of HTTP and status codes is required. A complete list of HTTP status codes and definitions can be found at the following website: [HTTP/1.1 Status Code Definitions](#)

The primary status codes used by the REST API are as follows:

- **200 OK**—Success.
- **201 Created**—A resource was successfully created.
- **400 Bad Request**—Invalid data was supplied for the relevant resource type.
- **401 Unauthorized**—The requested resource cannot be loaded until valid sign-in credentials have been entered. If this error is received after credentials have been entered, it indicates that the credentials are not valid.
- **403 Forbidden**—Access to the requested resource has been denied. (Make sure that the authorized user has administrative privileges.)
- **404 Not Found**—The URI requested is invalid or the resource requested does not exist.
- **413 Request Entity Too Large**—The request is larger than the server is able to process.
- **500 Internal Server Error**—A generic error message, given when there is no suitable specific message.

Error Message Format

Error messages are returned as exception elements with a message element containing a user-facing error message.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<exc:exception xmlns:exc="http://genologics.com/ri/exception">
  <message>Process Not Found</message>
</exc:exception>
```

The exception may also include a suggested-actions element with more detail on how to resolve the error.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<exc:exception xmlns:exc="http://genologics.com/ri/exception">
  <message>Invalid Artifact</message>
  <suggestedActions>'ml' is not a valid unit for volume. Expected 'uL'.</suggestedActions>
</exc:exception>
```

User-facing XML error messages are not returned for 401 and 403 errors. In these cases, the HTTP error must be resolved.

XML UTF-8 Character Encoding

When generating XML, explicitly set the document to UTF-8 character encoding.

If using other encoding methods (eg, MacRoman for OS X), special characters such as µg/ml are stored incorrectly. This could cause data integrity issues.

In Groovy, set the encoding attribute on the StreamingMarkupBuilder object as shown in the following example:

```
StreamingMarkupBuilder builder = new StreamingMarkupBuilder()
    builder.setEncoding('UTF-8')
```

Requesting API Version Information

When submitting a request to the REST API, specify the version of the API being used. The version number is a path parameter in each resource URI. The desired API version is substituted into the request URI as follows:

```
http[s]://<hostname>:  
<port>/api/<version.of.api>/<name.of.resource>
```

Changes to the API are tracked with a version number (version major) and a revision number (version minor).

- The version number indicates forwards and backwards compatibility.
- The revision number within the version describes features added to the API that do not negatively affect current functionality.

Only the version number is referenced as part of the request. The revision number simply tracks incremental enhancements to the API.

When a new version of the API is released, update the scripts and code as soon as possible.

Check API Version

To find out what version of the API is available from a given server, submit a GET request to the base API URI. For example, in a web browser, browse to:

```
http[s]://<hostname>:<port>/api/
```

The system returns the version:

```
<ver:versions>  
  <version minor="r12" major="v1"  
uri="http://yourIPAddress:8080/api/v1"/>  
</ver:versions>
```

Exception for Legacy Systems

The API was originally intended for internal use or for just a few customers. In those early days, API versioning was different. If working with legacy scripts, this older functionality can be maintained. For example, if scripts were written before v2 and have nondefault system configuration properties for `api.prefix` and `api.rewrite` on the server, the `.../api/` URI lists the resources and does not provide version information.

Viewing Paginated List Resources

The REST API only returns 500 results per request. Because of this feature, with certain resources, you can use the `start-index` parameter and `previous-page` and `next-page` elements to work with large amounts of data.

For example, the following request is submitted to the API:

```
http://host/api/version/samples
```

If the system contains more than 500 samples, the system responds with:

```
<smp:samples>
  <sample uri="http://host/api/version/samples/SAMPLE1" limsid="LIMSID1"/>
  <sample uri="http://host/api/version/samples/SAMPLE2" limsid="LIMSID2"/>
  <sample uri="http://host/api/version/samples/SAMPLE3" limsid="LIMSID3"/>
  <sample uri="http://host/api/version/samples/SAMPLE4" limsid="LIMSID4"/>
  <sample uri="http://host/api/version/samples/SAMPLE5" limsid="LIMSID5"/>
  ...
  <next-page uri="http://host/api/version/samples?start-index=500"/>
</smp:samples>
```

Then, use the URI provided by the next-page element to view the next 500 results:

```
http://host/api/version/samples?start-index=500
```

The response looks like this:

```
<smp:samples>
  <sample uri="http://host/api/version/samples/SAMPLE501"
limsid="LIMSID501"/>
  <sample uri="http://host/api/version/samples/SAMPLE502"
limsid="LIMSID502"/>
  <sample uri="http://host/api/version/samples/SAMPLE503"
limsid="LIMSID503"/>
  ...
  <previous-page uri="http://host/api/version/samples?start-index=0"/>
  <next-page uri="http://host/api/version/samples?start-index=1000"/>
</smp:samples>
```

Note the presence of the previous-page and next-page URIs, which allow moving within the pages of results.

Viewing Results from a Specified Point

Use the start-index parameter to view results from a specified point in a list. The first record in a list is index 0, and you can use values that are positive, whole numbers. If the value specified is greater than the number of results available from a resource, the system returns an empty list.

Configuring the Maximum Number of Requests

By default, the REST API only returns 500 results per request. To change the default number, contact the Illumina Support Team.

Filtering List Resources

When submitting a GET request to certain REST API resources (also known as list resources), the system returns a list of records. For example, submitting a GET request to the samples resource returns a list of all submitted samples stored in the system. Depending on the resource being used, use various query parameters to filter the records based on certain criteria. For more information about the parameters that are available, refer to the reference documentation for the desired resource.

To filter a list, the resource and parameter must be separated with a question mark (?). The parameter and the value you want to base the query on must be separated with an equal sign (=).

```
http[s]://<hostname>:<port>/api/<version.of.api>/<name.of.list.resource>?  
parameter=value
```

When filtering a list of artifacts, combine parameters within the same query statement. You can also repeat certain parameters, specifying a new value with each occurrence of the parameter.

The first parameter must be preceded with a question mark (?). Add additional parameters by separating each parameter with an ampersand (&).

Repeating a parameter with new values:

```
http[s]://<hostname>:<port>/api/<version.of.api>/<name.of.list.resource>?  
parameterA=value1&parameterA=value2
```

Combining parameters:

```
http[s]://<hostname>:<port>/api/<version.of.api>/<name.of.list.resource>?  
parameterA=value&parameterB=value
```

When combining or repeating parameters, each record returned matches one of the parameter values, or all the parameter values, depending on the usage:

- If the query statement contains multiple values for the same parameter, the ampersands are treated as an OR.
- If the query statement contains values for multiple parameters, the ampersands are treated as an AND.

For example, if a project LIMS ID and a process type are provided as parameters, the system returns only the files that match both the project LIMS ID and the process type. To see the files that match the project LIMS ID or the process type, issue two separate GET requests and combine the results.

- **/api/v2/processes**—This URI returns all processes run in the system.
- **/api/v2/processes?type=MALDI**—This URI returns all MALDI processes run in the system.
- **/api/v2/processes?type=Sample Prep&type=MALDI**—This URI returns all MALDI or Sample Prep processes run in the system.
- **/api/v2/containers**—This URI returns all containers in the system.
- **/api/v2/containers?type=Tube**—This URI returns all tubes in the system.

- **/api/v2/containers?type=Tubeandname=27-111andname=27-112**—This URI returns the tubes in the system that are named 27-111 OR 27-112.

Using the Last-Modified Parameter

Certain resources include the last-modified query parameter.

When used, the system displays only the results that have been modified because the last-modified date. The last-modified date is represented in ISO 8601 Complete date including hours, minutes, and seconds format: YYYY-MM-DDThh:mm:ssTZD.

Viewing Paginated Results

Lists of records are often large, spanning multiple pages. Many list resources include parameters that are used to work with paginated results.

Using UDF and UDT Parameters

Certain resources include parameters that can be used to filter the results displayed based on UDF information that is associated with the results:

- **udf.UDFNAME[.OPERATOR]=UDFVALUE**—This parameter filters the results based on a specified value for a specified UDF. Any item that contains the value for the UDF is returned, unless parameters include an optional operator filter ([.OPERATOR] in the expression provided). The lowercase filter operators of min or max are described later.
- **udt.name=UDTNAME**—This parameter filters the results based on a specified UDT. Any item that has the UDT selected is returned.
- **udt.UDTNAME.UDFNAME[.OPERATOR]=UDFVALUE**—This parameter filters the results based on a specified value for a specified UDF that resides within a specified UDT. Any item that contains the value for the UDF is returned, unless parameters include an optional operator filter ([.OPERATOR] in the expression provided). The lowercase filter operators of min or max are described later.

To filter results using UDF information, use the following query structure:

```
http[s]://<hostname>:<port>/api/<version.of.api>/<name.of.list.resource>?
udf.<name.of.udf>.<operator>=<udf.value>
```

To filter results using the name of a UDT, use the following query structure:

```
http[s]://<hostname>:<port>/api/<version.of.api>/<name.of.list.resource>?
udt.name=<name.of.udt>
```

To filter results using UDF information that is part of a specific UDT, use the following query structure:

```
http[s]://<hostname>:<port>/api/<version.of.api>/<name.of.list.resource>?
udt.<name.of.udt>.<name.of.udf>.<operator>=<udf.value>
```

When filtering lists and using date or numeric UDF or UDT values, use operators to restrict a query. The following operators are supported:

- **.min**—This operator displays results that are greater than or equal to the specified value.
- **.max**—This operator displays results that are less than or equal to the specified value.

Examples:

- **/api/v2/processes?type=Sample Prepandudt.name=Plasma**—This URI returns all Sample Prep processes run in the system that have the Plasma UDT selected.
- **/api/v2/processes?type=Sample Prepandudt.Plasma.Platelet Count.min=50**—This URI returns all Sample Prep processes that have a UDF named Platelet Count with a value of 50 or greater, within a UDT named Plasma.
- **/api/v2/processes?type=Sample Prepandudf.Sample=Serumandudf.Sample=Tissue**—This URI returns all Sample Prep processes that have a UDF named Sample with a value of Serum OR a UDF named Sample with a value of Tissue. More examples of filtering exist in the Cookbook.

For more examples of filtering, see the [About the Cookbook](#).

When filtering with UDT or UDF parameters, all special characters in the parameter string must be URL encoded. The Pipe (|) or the URL-encoded pipe (%7C) cannot be used.

When filtering on a UDF that is configured as a Multiline Text UDF, if a value contains a hard return, the value must include the URL-encoded line feed (%0A) at the appropriate location. Depending on how API requests are issued (via a browser or a script), spaces in names or values may require URL encoding, and trailing spaces in a name or value always require encoding. For example, for results to be returned, 'name ' requires 'name '.

Working with User-Defined Fields (UDF) and Types (UDT)

Removing Fields and Types

To remove a UDF or UDT value, submit a PUT request with the desired UDF or UDT omitted from the XML.

Updating Resources

When submitting a PUT, it is critical to update all information. The submitted XML must include all the current UDFs and UDTs for the resource. If the field and type elements for a UDF and UDT are not included, the system removes those fields and types.

To update the UDF information for an item, the PUT request can add new UDF values and update or remove current UDF values. When working with UDTs, replace the current UDT with another UDT, or add or remove fields within the current UDT.

- Update all UDFs and UDTs in a PUT.

- Even if the current user-defined values are not changing, include the current UDF and UDT values in the XML representation for a PUT request.

Filtering with UDF and UDT Values

Data formatting of the UDF and UDT values is important when filtering a resource list with a query parameter. When using UDF or UDT values as a query parameter, all nonalphanumeric characters must be URL encoded.

Data Type Formats

UDFs and UDTs are presented as fields and types in the XML. The following example shows a representation of fields and types returned by a GET request for a sample:

```
<smp:sample uri="http://host/api/version/samples/ADM5A12" limsid="ADM5A12">
  <name>Test Sample-1</name>
  <date-received>2010-12-17</date-received>
  <project uri="http://host/api/version/projects/ADM5" limsid="ADM5"/>
  <submitter uri="http://host/api/version/researchers/2">
    <first-name>Facility</first-name>
    <last-name>Administrator</last-name>
  </submitter>
  <artifact uri="http://host/api/version/artifacts/ADM5A12PA1?state=44"
limsid="ADM5A12PA1"/>
  <udf:type name="This is a UDT">
    <udf:field type="String" name="This is a UDF within a UDT">Value for a UDF within a
UDT</udf:field>
  </udf:type>
  <udf:field type="Date" name="Date UDF">2010-12-01</udf:field>
  <udf:field type="String" name="Single-line text UDF">Single line of
information</udf:field>
  <udf:field type="Text" name="Multi-line text UDF">Multiple
lines
of
information</udf:field>
  <udf:field type="Numeric" name="Numeric UDF">123456.7</udf:field>
  <udf:field type="Boolean" name="Check box UDF">true</udf:field>
</smp:sample>
```

Data Type Values in XML

XML resource representations do not render UDF values in the same format as views in the client user interface. The table later in this section compares images taken from the client user interface and the XML from an http GET.

Image of Client User Interface	Image of API XML
spaces: space mutlisingle: space multi precision4: 4.5300 dateExample: Feb 1, 2011	<pre> <udf:field type="String" name="spaces"> space </udf:field> <udf:field type="Text" name="mutlisingle"> space multi </udf:field> <udf:field type="Numeric" name="precision4">4.53</udf:field> <udf:field type="Date" name="dateExample">2011-02-01</udf:field> </pre>

The differences are intentional, to remove ambiguity and aid script writers when handling the data values. The following table compares the values displayed by the user interface and the API for UDF data types.

Configured Data Type	API XML Response Element Type Name	Client Display	API Element Type Format
Single-line Text	String	Leading and trailing spaces rendered	Leading and trailing spaces rendered
Multi-line Text	Text	Leading and trailing spaces rendered	Leading and trailing spaces rendered
Numeric	Numeric	Set by display-precision ie, 4.5300 is displayed when display-precision equals 4	Simplest numeric form, removing trailing zeros ie, 4.53
Date	Date	mmmm dd, yyyy ie, Feb 15, 2019	yyyy-mm-dd ie, 2019-02-15

Numerics and Significant Digits

Trailing zeros are removed to support both integer and real numerics. Determine the significant digits by looking up the display-precision element of the `/configuration/udfs/{udfid}` resource.

EPP UDF Date format

When Date UDFs are expanded on the EPP command line, their format differs from the one used in the Clarity LIMS GUI and the REST API.

Traversing a Genealogy

Traversing Up a Genealogy

The artifacts resource includes a parent-process element that provides a URI to the process that created an artifact.

```

<artifact ...>
  <parent-process uri="http://host/api/apiversion/process/LIMS-ID"
  limsid="LIMS-ID"/>
  ...
</artifact>

```

To facilitate walking up the genealogy, the processes resource exposes the parent process for an input artifact in the input-output-map of a process:

```

<process ...>
  <input-output-map>
    <input uri="http://host/api/apiversion/artifact/LIMS-ID?state=123"
    post-process-uri="http://host/api/apiversion/artifact/LIMS-ID?state=125"
    limsid="LIMS-ID">
      <parent-process uri="http://host/api/apiversion/process/LIMS-ID"
      limsid="LIMS-ID"/>
    </input>
    ...
  </input-output-map>
  ...
</process>

```

The parent-process element does not display if the parent process is not supported by the API.

Traversing Down a Genealogy

The processes resource supports an inputartifactlimsid query parameter. This parameter limits the list of processes to those processes with one of the specified artifacts as an input.

1. Start with the initial sample.

```

<sample ...>
  ...
  <artifact limsid="ARTIFACT-LIMS-ID"
  uri="http://host/api/apiversion/artifacts/ARTIFACT-LIMS-ID">
  </sample>

```

2. The processes for the sample can be queried, as follows.

```

http[s]://<hostname>:<port>/api/<version.of.api>/processes?
inputartifactlimsid=ARTIFACT-LIMS-ID

```

3. The process contains an input-output-map for the input artifact.

```

<process ...>
  <input-output-map>
    <input limsid="ARTIFACT-LIMS-ID"
uri="http://host/api/apiversion/artifacts/ARTIFACT-LIMS-ID" .../>
    <output limsid="ANOTHER-ARTIFACT-LIMS-ID"
uri="http://host/api/apiversion/artifacts/ANOTHER-ARTIFACT-LIMS-ID"
.../>
  </input-output-map>
</process>

```

4. The steps can then be repeated using the LIMS ID of each output artifact that is associated with the input artifact.

Batch Resources Overview

When multiple users are working on multiple plates in high-throughput labs, programmers may find that the large number of HTTP method calls to the REST API can slow down their scripts.

To improve performance, Illumina has created the following batch resources:

- artifacts.batch.retrieve
- artifacts.batch.update
- containers.batch.create
- containers.batch.retrieve
- containers.batch.update
- files.batch.retrieve
- files.batch.update
- samples.batch.create
- samples.batch.retrieve
- samples.batch.update

Use the batch resources to access a group of artifacts or a group of containers using a single batch method call. Using these resources to iterate a list of items significantly improves script execution times.

Key Concepts

Batch resources are best thought of as unordered collections or lists of items accessed. A POST to batch/create, batch/update or batch/retrieve, therefore, is a request to create, update, or retrieve those items. There is no guaranteed order to batch responses.

Batch resources are nonbreaking additions to the existing REST API. Updated scripts can still use their existing nonbatch methods.

For example, the resources may have URIs (Universal Resource Identifiers) such as:

```
http[s]://<hostname>:8080/api/<v2>/artifacts/batch/retrieve  
http[s]://<hostname>:8080/api/<v2>/artifacts/batch/update  
http[s]://<hostname>:8080/api/<v2>/containers/batch/retrieve  
http[s]://<hostname>:8080/api/<v2>/containers/batch/update  
http[s]://<hostname>:8080/api/<v2>/containers/batch/create
```

Batch operations do not require sophisticated HTTP client or server methods. The only HTTP method for batch resources is POST.

What is Sent in a Batch Resource POST?

To update a group of artifacts, use a POST operation to the /artifacts/batch/update resource. The XML input payload consists of a series of elements, as follows.

```

<art:details xmlns:ri="http://genologics.com/ri"
xmlns:udf="http://genologics.com/ri/userdefined"
xmlns:file="http://genologics.com/ri/file"
xmlns:art="http://genologics.com/ri/artifact"
xmlns:con="http://genologics.com/ri/container">
  <art:artifact
uri="http://localhost:8080/api/v2/artifacts/LUN3A1PA1?
state=1088" limsid="LUN3A1PA1">
    <name>Cancer Sample-1</name>
    <type>Analyte</type>
    <output-type>Analyte</output-type>
    <volume unit="uL">10.0</volume>
    <qc-flag>UNKNOWN</qc-flag>
    <location>
      <container
uri="http://localhost:8080/api/v2/containers/65-1"
limsid="65-1"/>
        <value>A:1</value>
      </location>
      <working-flag>false</working-flag>
      <sample uri="http://localhost:8080/api/v2/samples/LUN3A1"
limsid="LUN3A1"/>
    </art:artifact>
  <art:artifact
uri="http://localhost:8080/api/v2/artifacts/LEU9A1PA1?
state=1131" limsid="LEU9A1PA1">
    <name>Diseased Sample</name>
    <type>Analyte</type>
    <output-type>Analyte</output-type>
    <volume unit="uL">10.0</volume>
    <qc-flag>UNKNOWN</qc-flag>
    <location>
      <container
uri="http://localhost:8080/api/v2/containers/27-111"
limsid="27-111"/>
        <value>A:0</value>
      </location>
      <working-flag>false</working-flag>
      <sample uri="http://localhost:8080/api/v2/samples/LEU9A1"
limsid="LEU9A1"/>
    </art:artifact>
  </art:details>

```

What is returned from a batch resource POST?

As large data transfers can affect performance, it is important to return concise XML in response to a batch resource request. Therefore, except for retrieve resources, the XML output payload consists of a list of created or updated URI links, such as the following:


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ri:links xmlns:ri="http://genologics.com/ri"><link
uri="http://localhost:8080/api/v2/artifacts/LUN3A1PA1"
rel="artifacts"/><ri:links
xmlns:ri="http://genologics.com/ri">
  <link
uri="http://localhost:8080/api/v2/artifacts/LUN3A1PA1"
rel="artifacts"/>
  <link
uri="http://localhost:8080/api/v2/artifacts/LEU9A1PA1"
rel="artifacts"/>
</ri:links>
```

Return Codes

The batch resources use common HTTP return codes:

- An HTTP 200 (OK) code is returned when batch resources have been successfully created or updated.
- An HTTP 400 error code is returned if the input payload details included incorrect, mixed, or duplicate URI links. For example, if the details of an artifacts.batch.update (list) request included a container resource.

Getting Started with the API

The internal Clarity LIMS API (eg, <https://example.claritylims.com/clarity/api>) is the API used to deliver the Clarity LIMS web interface. This interface is not typically meant for public consumption. However, some customers use it for troubleshooting and to mitigate system issues.

Preventing CSRF Attacks

As of Clarity LIMS v5.1, access to the internal Clarity LIMS API changed to enhance security and prevent Cross Site Request Forgery (CSRF) attacks. Two new HTTP headers must now be present when issuing PUT, POST, DELETE, and PATCH requests:

- **Origin**—This header must be set to the scheme and authority of the server being accessed (eg, <https://example.claritylims.com>).
- **X-Requested-With**—This header must be set to XMLHttpRequest.

The attached cURL, Python, and Java examples demonstrate how to authenticate and issue internal API requests. These examples assume a Clarity LIMS server at <https://example.claritylims.com>.

Attachments

- [csrf_headers.sh.txt](#)
- [csrf_headers.py.txt](#)
- [csrf_headers.java.txt](#)

Getting Started with the API

The internal Clarity LIMS API (eg, `https://example.claritylims.com/clarity/api`) is the API used to deliver the Clarity LIMS web interface. This interface is not typically meant for public consumption. However, some customers use it for troubleshooting and to mitigate system issues.

Preventing CSRF Attacks

As of Clarity LIMS v5.1, access to the internal Clarity LIMS API changed to enhance security and prevent Cross Site Request Forgery (CSRF) attacks. Two new HTTP headers must now be present when issuing PUT, POST, DELETE, and PATCH requests:

- **Origin**—This header must be set to the scheme and authority of the server being accessed (eg, `https://example.claritylims.com`).
- **X-Requested-With**—This header must be set to `XMLHttpRequest`.

The attached cURL, Python, and Java examples demonstrate how to authenticate and issue internal API requests. These examples assume a Clarity LIMS server at `https://example.claritylims.com`.

Attachments

- [csrf_headers.sh.txt](#)
- [csrf_headers.py.txt](#)
- [csrf_headers.java.txt](#)

Understanding API Terminology (LIMS v5 and later)

The following table shows how API terminology maps to terminology used in the Clarity LIMS v5.x interface.

API Terminology	Clarity LIMS Terminology	Notes
Analyte	Derived sample	Not applicable
Artifact	An item that is input to or generated by a step. Derived samples and measurements are both artifacts.	Not applicable
Lab	Account	Accounts are not fully supported in the Clarity LIMS v5.x web interface. However, lab is supported in the API.
Process	Step	step and process both exist in the API. While related, they are not synonyms and have different uses.
Process type	Master step	Not applicable

API Terminology	Clarity LIMS Terminology	Notes
Researcher	Client or user	Not applicable
Resultfile	Measurement or file / file placeholder	A file could be a log file that is shared across all samples in the step or a file that belongs to a single sample, such as an Electropherogram.
Sample	Submitted sample	The original sample submitted to the system.
UDF	Custom field	User-defined types (UDTs) are not supported in the Clarity LIMS v5.x web interface. However, udt is supported in the API.

See also the [Terms and Definitions](#) section.

API-Based URIs (LIMS v4 and later)

Clarity LIMS version 4.0 introduced architectural changes that enforce SSL-based security. As a result, the structure of the URIs that reference the Clarity LIMS API was modified, and scripts written before Clarity LIMS v4.0 may require updating.

Details

Scripts that use the API do so by using RESTful methods on specific URIs. The base portion of the URI references the server on which the Clarity LIMS application is running.

Before Clarity LIMS v4.0 the base portion of the URI took the following form:

```
http[s]://<your_server_name>:<your_port_number>/api
```

Where:

- The protocol could either be HTTP or HTTPS, depending on whether the application was SSL-enabled or not.
- <your_server_name> represented the fully qualified domain name (or IP address) relating to the server on which the Clarity LIMS application was running.
- <your_port_number> represented the port number (typically 8080) on which the Clarity LIMS application was listening.

In Clarity LIMS v4.0 and later, the base portion of the URI is in the following form:

```
https://<your_server_name>/api
```

Where:

- The protocol must be HTTPS, because the Clarity LIMS application is now installed with SSL enabled.
- The server name must match the certificate that was purchased and installed into Clarity LIMS.

- The port number (and the colon) is no longer required. Do not provide it.

When to Update Existing Scripts

The following information should help determine if updates to the scripts are needed.

Scripts generally determine the API URI in one of the following ways:

1. The URI is passed to the script by the automation or External Program Plugin (EPP) component, as a parameter or command-line argument.
2. The URI is passed to the script by another script or a command line embedded in a crontab file.
3. The script contains the URI as a hard-coded string literal.
4. The script determines the fully qualified domain name of the server and adds the prefix (`http://`) and suffix (`:8080`) accordingly.
5. The script imports, or includes a file that contains, the URI.

Most scripting uses methods one or three. However, other methods may be used in the facility.

- If method one is used, it is not necessary to update the scripts because Clarity LIMS passes in the new form of the URI.
- If other methods are used, you likely need to update the scripts to convert the URI to the new format. Often, a search and replace tool is able to make these changes.

To make sure that the correct locations are searched, keep in mind that scripts are often stored in the following locations:

1. In the `/opt/gls/clarity/customextensions` folder (and subfolders) on the server where Clarity LIMS is running. This location is the domain of the default Automation Worker (AW)/Automated Informatics (AI) node, which listens on the channel name of `limsserver`.
2. If there are additional AW/AI nodes on the server, in the folders used by these nodes.
3. If there are additional AW/AI nodes external to the Clarity LIMS server, within the folders used by these nodes.
4. If scripts are launched by cron, or other mechanisms, they could be stored anywhere and may not even be on the Clarity LIMS server itself.

For points 1–3, query Clarity LIMS (via either the API or the database) to produce a listing of all the scripts it is configured to use. As a result, determine on which node they run and their location.

For point 4, there is no easy answer. Hopefully, if the script is important, the location has been documented.

 | As of BaseSpace Clarity LIMS v5.0, the terms External Program Integration Plug-in (EPP), EPP node, and AI node are deprecated.

The term EPP has been replaced with automation, while the Automated Informatics (AI) node is referred to as the Automation Worker (AW) node.

Development Prerequisites

Understanding lab information management in a scientific context is one of the more powerful skills in genomics research today. The Clarity LIMS Rapid Scripting™ API is designed to use these skills, allowing a knowledgeable scientific programmer to adapt lab informatics with scripts and automation.

 | Based on experience working with bioinformaticians and scientific programmers, assumptions about your background, setup, and skills have been made.

Before You Begin

Before using the API Cookbook, set up a [Nonproduction Scripting Sandbox Servers](#).

If any of the topics covered on this page are a concern, contact the Clarity LIMS Support team for additional training or custom scripting services.

Terminology

Within the Cookbook, the term scripting refers to programs running independently of the client and server that direct the input and output of information. Use scripts and the API for file handling and text processing in the context of biological samples, containers, and instruments.

Skills and Training

- The API Cookbook assumes that you can program in modern computer languages and are comfortable with scripting and bioinformatics.
- The topics are best understood by those users who can program small applications and are experienced with experimental processes in molecular biology.
- The topics assume that you have received administrator-level training or know how to configure the system. The topics also assume that a nonproduction server is set up to play with cookbook examples, develop real scripts, and test before deploying in production.
- Be comfortable with the following skills:
 - XML
 - System file handling
 - General-purpose scripting languages
 - Working on the command line

Nonproduction Scripting Sandbox Servers

Illumina provides multiple server licenses for API users: a production server license and one or more nonproduction server licenses for developing and testing.

To allow developers to design, build, test, and upgrade efficiently, it is recommended to install at least two servers. Installing three is even better.

The nonproduction server licenses serve the following purposes:

- To provide a sandbox to experiment with the API and the system configuration.
- To provide a verification platform for upgrading scripts, software components, and overall system integration before deploying to production.

All the examples in the [About the Cookbook](#) are intended to be used with the nonproduction scripting sandbox server. See [Useful Tools](#).

i | If you do not have the time or resources to use the API, but are interested in expanding your implementation, contact the Clarity LIMS Support team. There are various consulting, training, and scripting services available.

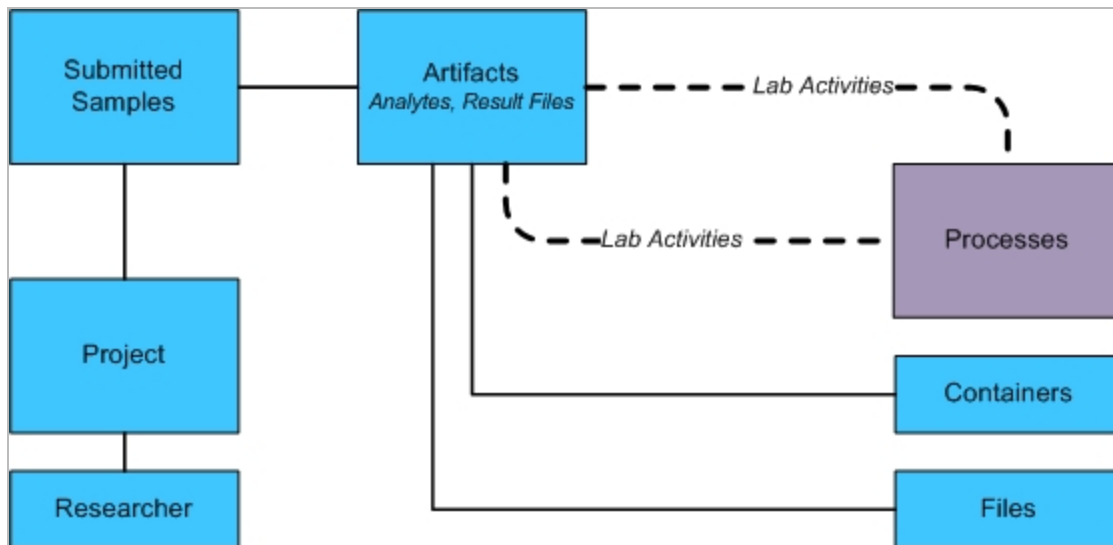
Related Resources

- [Integrating Scripts](#)
- [Useful Tools](#)
- [Suggested Reading](#)

Structure of REST Resources

The information recorded in BaseSpace Clarity LIMS is organized into resources within the REST API. Each resource refers to an XML schema associated with a namespace. Before working with the REST Web Service, understand how the information recorded in Clarity LIMS translates to the REST resources.

The following diagram highlights the major REST resources. Each resource is discussed further in the following sections.



- Samples are the objects that are entered into Clarity LIMS before processing begins. Every sample belongs to a single project and has a related analyte (sample) artifact. Every project must have an associated researcher.

- When you add a sample to the system, it is classified as a submitted sample. This feature allows the original samples, and any related data, to remain separate and distinct, even as processing and aliquot creation occurs. Every sample or file created by running a step from the Clarity LIMS user interface can be traced back to a submitted sample.
- In Clarity LIMS, processes (known as steps in the user interface) are run on analyte (derived sample) artifacts. Samples must always be in containers.
- Clarity LIMS v4.x and earlier: In the Clarity LIMS Operations Interface processes are run on analyte (sample) or result file artifacts. Samples must always be in containers.
 - As of BaseSpace Clarity LIMS v5, the Operations Interface Java client used by administrators to configure processes, consumables, user-defined fields, and users have been deprecated. All configuration and administration tasks are now executed in the Clarity LIMS web interface.
 - To understand how API terminology maps to terminology used in the Clarity LIMS v5 interface, see [Understanding API Terminology \(LIMS v5 and later\)](#).

Samples

Within the REST Web Service, the samples resource is key.

The samples resource represents submitted samples and contains information about those samples, including:

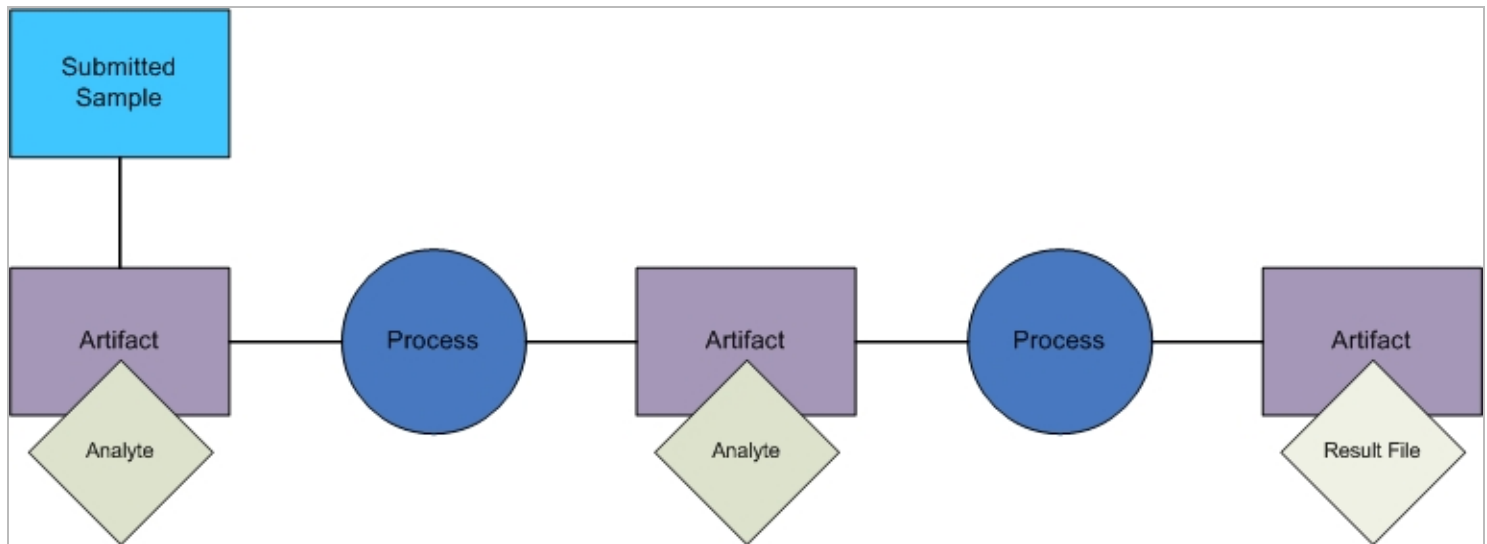
- The dates samples are entered and received.
- Any user-defined data related to the samples.

When a sample is added to Clarity LIMS, the system also creates an artifact (see [Artifacts](#)).

While the artifact associated with a submitted sample is only seen at the database or REST level, and is never exposed in the Clarity LIMS interface, the system uses this artifact when running protocol steps.

When running a step on a submitted sample, the artifact is used as an input to the step, not the submitted sample itself. All artifacts reside within the artifacts resource.

When a submitted sample is processed, the system generates output artifacts. Depending on the configuration of the process, many types of artifacts (including result files) can be generated. Any downstream sample created by running a process is considered an analyte artifact (referred to as a derived sample in the user interface).



Projects

Projects are used to group samples based on the originating lab (account) or study. Projects collect all records related to that sample in Clarity LIMS.

A project stores information about:

- The client (researcher) who owns it
- Significant dates
- The status of the project
- Any user-defined information that the lab needs to collect

After creating a project, you can add samples to it. Samples can then be added to workflows, and steps (processes) are run on those samples to reflect the analysis performed in the lab.

In the REST Web Service, a submitted sample can only belong to one project. You can use the projects resource to return projects.

Note the following details regarding projects:

- Every submitted sample must belong to a project.
- Every project must be assigned to a researcher (an owner) that corresponds to a client in the system.
- In the Clarity LIMS user interface, the term contact has been replaced with client. However, the API permission is still called contact.

The researchers resource represents clients in the system.

When working with projects, each project must list a client as the owner of the project. This role generally represents the person who submitted the original samples.

The client does not need to have a user account.

Processes

- i** | In Clarity LIMS v5, the API still uses the term process. However, in the user interface, this term has been replaced with master step. Also, the Operations Interface has been deprecated.
- **Clarity LIMS v4.2 and earlier**—Created in the Clarity LIMS Operations Interface, process types model and track the work performed on the samples in the lab. In the Clarity LIMS web interface, these process types are then used as building blocks to create and configure steps in the Clarity LIMS web interface. These steps are known as processes in the API.
 - **Clarity LIMS v5 and later**—Created in the Clarity LIMS web interface, master steps model and track the work performed on the samples in the lab. These master steps are then used as building blocks to create and configure steps. These steps are known as processes in the API.

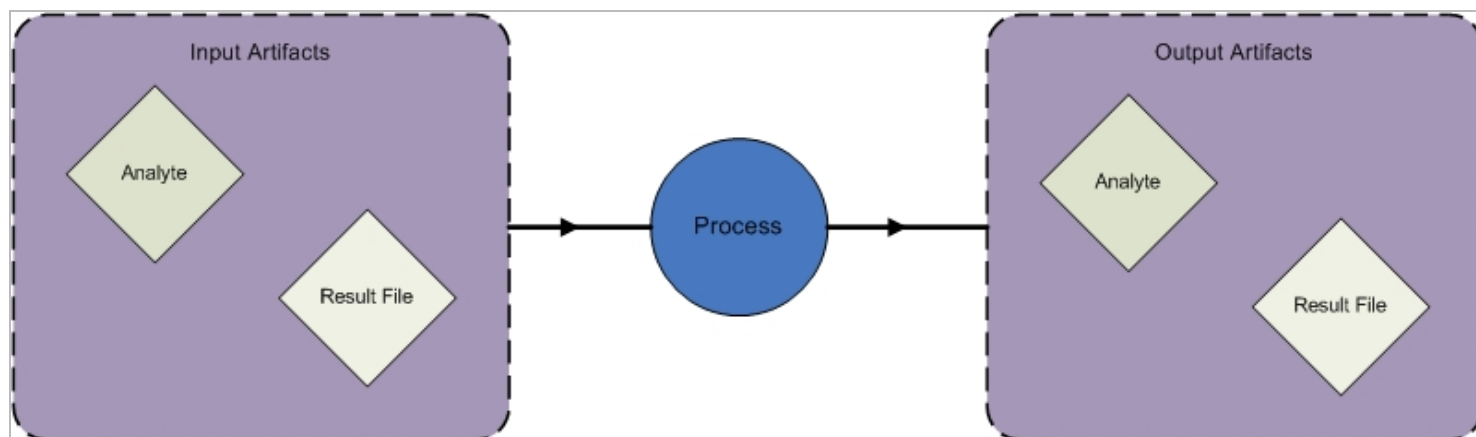
Different interfaces may allow you to run steps/processes on different artifacts.

In the API view, a process takes in one or many analytes and/or result files and creates one or many analytes and/or result files.

When running a step in Clarity LIMS, lab scientists record information about the step, the instruments used, and the properties and characteristics of the samples.

Depending on the configuration of the process/master step on which it is based, the step can generate another sample analyte and/or placeholders to which result files can be attached for storage in the system.

With the REST Web Service, the processes resource is used to track these activities.



Note the following details regarding processes:

- Processes are used to represent work that occurs in the lab or in silico.
- Processes take inputs and create outputs. With the REST processes resource, it is modeled using the input-output-map element.

In addition to tracking historical work via the processes resource in the REST Web Service, use the service to POST new processes to the system.

POSTing a process to the REST Web Service creates the process itself, along with the outputs of the process. However, all the input and output containers must exist in the system already.

- For a simple example of the XML required to POST a process, see the processes (list) section of the REST resources space.
- For basic details about POSTing processes, see *Working with Processes/Steps* in the Cookbook section.
- For examples of process POSTing, see *Pooling Samples with Reagent Labels* and *Demultiplexing* in the Cookbook section.
- To find out how to integrate automation with process POSTing to set quality control flags, see the *Setting Quality Control Flags* application example.

Query the processes resource using input artifact LIMS IDs. This query allows you to find the processes that were run at each step in the workflow or on each artifact generated during processing.

Artifacts

All inputs and outputs of a process are artifacts and can be returned via the artifacts resource.

Note the following details about artifacts:

- An artifact is a derivative of a sample and is used as an input to a process.
- An artifact may be a sample analyte or a result file.
- The artifacts resource includes artifacts for the submitted sample and all process outputs, both file- and sample-based.

Artifacts are categorized by type, to distinguish between pure information results (file-based artifacts, such as result files) and the biological material created by processing the sample (analyte artifacts).

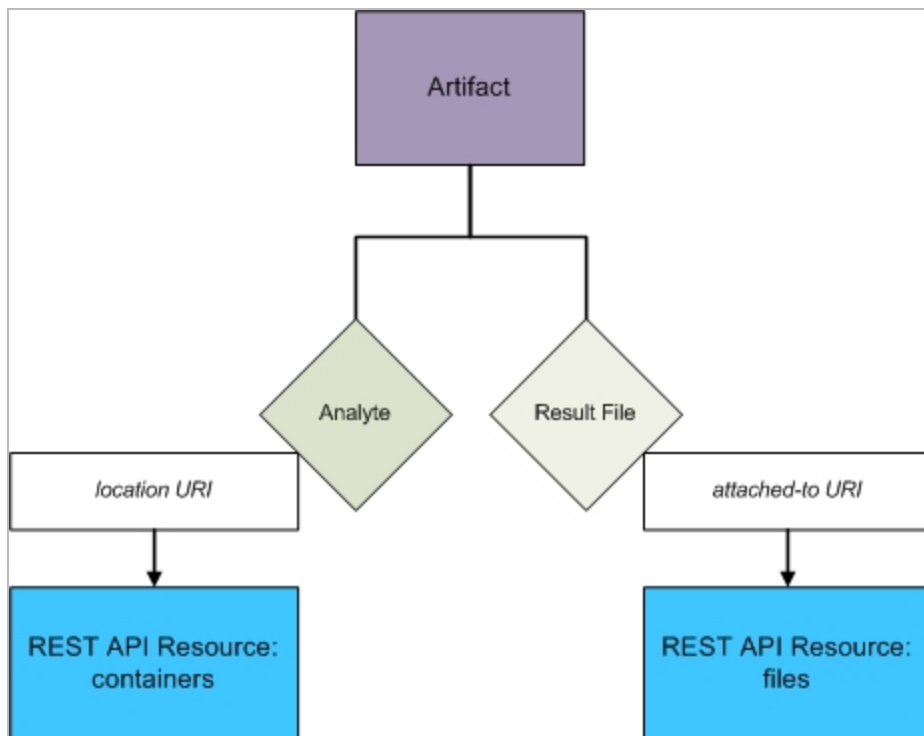
In Clarity LIMS, the term artifact is used to describe items needing to be processed. Think about artifacts as the intellectual property added by the lab.

For example, applying reagents to change the nature of a sample creates an artifact, as does generating and analyzing data files by running a sample on a NextGen or microarray instrument.

Anything created by a process in the system is an artifact. In the REST Web Service, there are several types of artifacts, but this article focuses on two:

- Computer-generated files called result files
- Physical sample derivatives called analytes

The high-level relationship between artifacts, analytes, and result files is shown in the following diagram.



An artifact references data elements, which vary depending on the type of artifact you are working with. For example, a result file has an attached-to URI that links to a files resource, whereas an analyte has a location URI that links to the containers resource.

Artifacts are key to tracking lab process activities and also link to a submitted sample.

All artifacts include one or more sample URI data elements, which make it easy to trace any lab-generated product or result directly back to its original sample.

When working with artifacts in the REST API, their URIs often include a numeric state. The state is used to track historical QC, volume, and concentration values.

Unless you are interested in a historical state, it is best practice not to include state when using an artifact URI. When state is omitted, the API defaults to the most recent state.

Containers

When samples are processed in the lab, they are always placed into containers of some sort (tubes, 96-well plates, flow cells, etc.) and moved into new containers as processing occurs.

For many kinds of processing, the container placement is a critical piece of information. Further processing of the sample, and data files created by analyzing the sample, are often linked based on the placement of the sample in the container.

Containers are central to processing in the lab. In Clarity LIMS, therefore, the samples (analyte artifacts) must also always be placed into a container resource.

When working with the REST Web Service, analyte artifacts include a URI that links to the container housing the artifact. Use the containers resource to view all the containers registered in the system.

Details on finding contents of a container can be found in the [About the Cookbook](#).

Note the following details about containers:

- Containers represent the tubes, plates, flow cells, and other vessels that can be populated with a sample.
- All samples/analytes must reside in a container or they are not visible in Clarity LIMS client.

All containers include a name and a LIMS ID.

- The name is a text element over which the scientific programmer has full control.
- The LIMS ID is a unique identifier generated by the system in a fixed format.

The name, LIMS ID, and any container-level [User-Defined Fields \(UDFs\)](#) provide various options for container labeling.

For assistance, the Illumina Consulting team can recommend various settings, such as uniqueness constraints, based on your requirements.

Files

A lab produces various files: large scientific result data files, summary result files, image files, label files, equipment and robotic setup files, and software logs.

These files are stored in different locations and it can be challenging to manage the relationship between a file on a computer or hard disk and the sample, step, or project with which it is associated.

Clarity LIMS lets you store files related to a project or sample and files generated during a step in a workflow. These files can be imported in various locations within the client and are stored on the file server.

To model this feature within the REST Web Service, there are two resources:

- files resource
- glsstorage resource

Within the REST Web Service, files are represented by the files resource. This resource manages files and the resources or artifacts to which they are related, and stores information about:

- The sample, project, or process output with which the file is associated, referenced by the attached-to URI.
- Where the file was imported from, and its original name, referenced by the original-location URI.
- The location of the file, referenced by the content-location URI. It also specifies the transfer protocol that can be used to retrieve the file. The following transfer protocols are supported:
 - FTP
 - SFTP
 - HTTP

Files Added using the LIMS Client

If using REST to view a file that was added through the Clarity LIMS client, the content-location URI references a location on the file server. This location is where the system stores all files that are imported through the Clarity LIMS client.

Files Added using REST

If you are using REST to import a file into the system, do one of the following:

Store the file on the file server:

1. Use the glsstorage resource to create a unique storage location and file name on the file server.
2. After this step is complete, the system returns a location and file name using the content-location URI element.
3. Then do as follows.
 - Provide the URI to the files resource.
 - Put the file in the specified location.

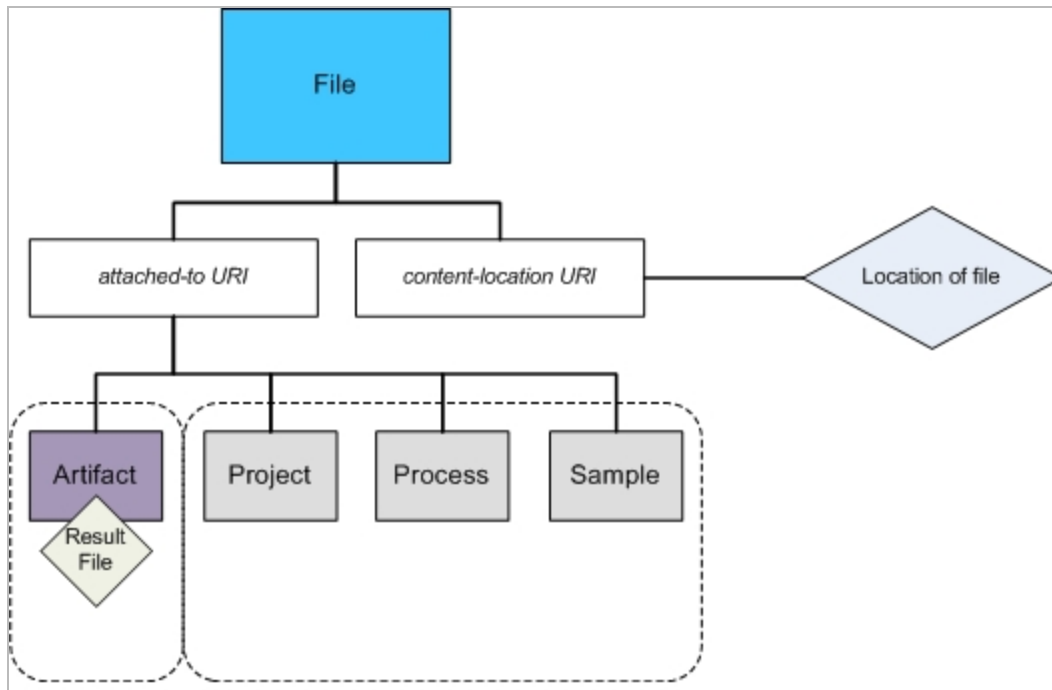
Store the file somewhere other than on the file server:

- Use the files resource and reference the name and location of your file with the content-location URI element.
- This feature must be configured by Illumina. For more information, contact the Support team.

Not the following key concepts:

- **Files**—The files resource defines the location of a file and its relationship with other REST resources, such as artifacts and projects.
- **Glsstorage**—The glsstorage resource allocates space on the file server.
- **XML elements**—Within the XML used by the files and glsstorage resources, the attached-to and content-location URIs are used to link disk files to file-based artifacts produced by a process or to link disk files to projects or samples.

The following diagram outlines how the XML elements link files to system resources and artifacts:



Using the REST Web Service to Work with Files

In the lab, one of the most important associations to make is between:

- A file that is the result of an instrument run.
- and -
- The sample that was analyzed to produce that file.

In Clarity LIMS, this association is represented by creating a process that takes a sample analyte and produces a result file.

When you run a process configured to create a result file, the process generates a placeholder for a file. To populate the placeholder, simply import the result file generated by the instrument into Clarity LIMS.

While working in the lab, lab scientists can upload result files that are used or produced while samples are processed. However, it may sometimes be more appropriate to automate this work. In these cases, you can use the REST files and the `glsstorageresource`.

Depending on the file storage needs and how the files are generated, there are two ways to do this process.

- Import a file and store it on the file server.
- Import a file and store it on a different server.

To import a result file and store it on the file server:

1. POST conforming XML to the `glsstorage` resource.
2. This action returns XML that includes a name and storage location for the file.

3. Place the file into the specified location using the file name provided in the XML.
4. POST the returned XML to the files resource, which links the file on disk to the result file placeholder.

To import a result file and store it on a different server:

1. Make sure that the file exists in the desired location.
2. POST conforming XML to the files resource, referencing the name and location of your file with the content-location element. The file path must contain the transfer protocol supported by the server. For example:

```
sftp://192.168.13.247/home/glsftp/Process/2010/10/SCH-RAA-101013-87-1/ADM53A1PS3-40-1.dat
```

 | It is not necessary to POST to the glsstorage resource.

Attaching Reference Information

If you have files that were not generated during the analysis of a sample, you can also attach reference information to projects and samples.

For example, suppose you receive an email when a sample is submitted to the lab, you may want to store that information in Clarity LIMS. In this case, when you POST XML to the files resource, the XML links the file to the desired submitted sample, instead of to a result file placeholder.

Clarity LIMS v5.x and later:

In Clarity LIMS, the file is attached to the Sample Details section of the Sample Management screen.

1. On the Projects and Samples screen, select the project containing the sample for which you have posted a result file.
 - a. Scroll down to the Samples and Workflow Assignment section of the screen and select the appropriate sample.
 - b. Select **Modify 1 Sample**.
2. On the Sample Management screen, scroll to the bottom of the Sample Details section to find the attached file.

Before Clarity LIMS v5:

In the Clarity LIMS web interface, the file is attached to the Sample Details section of the Sample Management screen.

 | For details on accessing the file, see the previous content on Clarity LIMS v5.x and later.

In the Clarity LIMS Operations Interface, the file is attached to the Files tabbed page of the applicable submitted sample.

1. In the Clarity LIMS Explorer, select **Opened Projects**.
2. In the Opened Projects list, double-click the project containing the sample for which you have posted a result file.
3. On the project details page, select the Samples tab.
4. At the bottom of the tab in the Containers pane, double-click the appropriate sample.
5. On the sample details page, select the Files tab to find the attached file.

- ! | Before POSTing to the files resource, make sure that the file exists in the location referenced by the content-location element. If the file does not exist in this location, the POST fails.

The REST Web Service separates the resources needed for files and file storage.

This action allows for greater control and the flexibility to apply various tracking and storage strategies. The content-location element can be used to define the file location without having to move the file. This ability is key in next-generation sequencing, which requires the management of large files, such as assemblies.

The content-location element needs to reference the file location in a storage system using a specific file transfer protocol. Currently only FTP, SFTP, and HTTP protocols are supported.

This mechanism makes file management flexible, but it maintains access to the file from REST with a single link. However, this feature must be configured by Illumina. For more information, contact the Support team.

User-Defined Fields (UDFs)

Note the following key concepts about UDFs and UDTs:

- UDFs and UDTs are configured to collect information that is important to the lab.
- With the REST Web Service, include UDF and UDT values in the XML representation of any individual resource that has a UDF or UDT defined.
- Not all artifacts have both UDFs and UDTs.

In Clarity LIMS v5 and later:

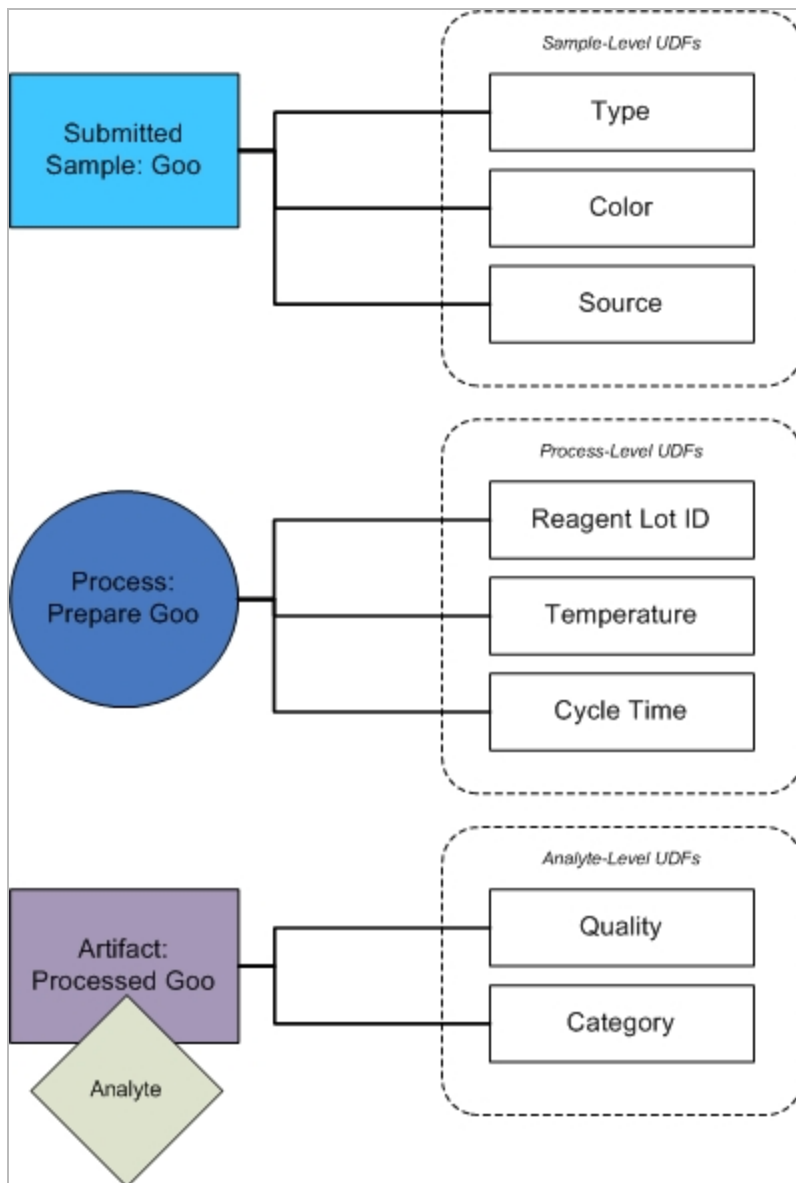
- The API still uses the term UDF. However, in the user interface, this term has been replaced with custom field.
- UDTs are not supported.

You can configure the system to collect user-defined information. Consider the following examples:

- You can create UDFs to add options and fields to the user interface when working with samples, containers, artifacts, processes/protocol steps, and projects.
- You can also create User-Defined Types (UDTs), which are organized subsets of related UDFs. As you add and process samples, you can add information to these options and fields.

In the following example, UDFs are added to submitted samples, processes, and sample analytes (derived samples).

- For the submitted sample named Goo, there are UDFs named Type, Color, and Source.
- For the Prepare Goo process/step, there are UDFs named Reagent Lot ID, Temperature, and Cycle Time.
- The output of the Prepare Goo process/step is an analyte named Prepared Goo, which contains UDFs named Quality and Category.



i | Any downstream sample created by running a process is considered an analyte artifact. In the Clarity LIMS interface, analyte artifacts are referred to as derived samples. For more information, see [Samples](#), [Artifacts](#), and [Processes](#).

To record information for these UDFs:

1. Add Goo to the system and populate the sample-level UDFs.
2. In Clarity LIMS, run the Prepare Goo step and complete the following actions:
 - Populate the Step Details fields (the process-level UDFs).
 - Populate the Sample Details table (the analyte-level UDFs).

Using the REST Web Service to Collect UDF Information

You can also use the REST Web Service to collect user-defined information for samples, containers, artifacts, and processes.

After you have configured UDFs, the XML of the appropriate resource expands with data elements for the field values. For example, the Prepared sample of goo artifact would have the following XML:

```
<art:artifact uri="http://yourIPAddress:8080/api/v1/artifacts/ADM54A1PR4?state=321"
limsid="ADM54A1PR4">
  <name>Prepared sample of goo</name>
  <type>Analyte</type>
  <output-type>Analyte</output-type>
  <parent-process uri="http://yourIPAddress:8080/api/v1/processes/PRE-MMS-100903-24-63"
limsid="PRE-MMS-100903-24-63"/>
  <qc-flag>UNKNOWN</qc-flag>
  <location>
    <container uri="http://yourIPAddress:8080/api/v1/containers/27-82" limsid="27-82"/>
    <value>1:1</value>
  </location>
  <working-flag>true</working-flag>
  <sample uri="http://yourIPAddress:8080/api/v1/samples/ADM54A1" limsid="ADM54A1"/>
  <udf:field type="String" name="Quality">Excellent</udf:field>
  <udf:field type="Numeric" name="Category">1</udf:field>
</art:artifact>
```

Configuring UDFs/Custom Fields

UDFs/custom fields are useful for collecting data at various stages of your workflow. In next-generation sequencing, it is important to record information, such as who submitted a sample, the tested concentration of a library, the reagents that were used during library prep.

As illustrated in the previous Goo example, collect this information by adding UDFs/custom fields for the samples, artifacts, and processes resources:

- A submitted sample UDF/custom field named Type
- An artifact-level UDF/derived sample custom field named Validated Concentration
- A process-level UDF/master step field named Reagent Name

Artifact UDFs/custom fields are flexible.

- You can configure different sets of UDFs/custom fields for the analyte artifact type and the result file artifact type.
- You can configure different sets of UDFs/custom fields based on the process type.

This flexibility means that:

- Process type/master step A can display fields m and n on a result file and fields q and r on an analyte.
- Process type/master step B can display fields m and o on a result file and fields q and s on its output analyte.

Process Type/Master Step	Result File Field Exposed	Analyte Field Exposed
A	m, n	q, r
B	m, o	q, s

Control how users access artifact-level UDFs/custom fields by configuring the type of artifact or process type/master step to which they apply.

Not every detail tracked and recorded needs a UDF. To optimize lab efficiency, it is recommended that you define an essential UDF set.

Increasing the complexity of information collected and managed does not necessarily improve operations or scientific quality. It may be more effective to store files, because the complete details are then available and secure within the attached file.

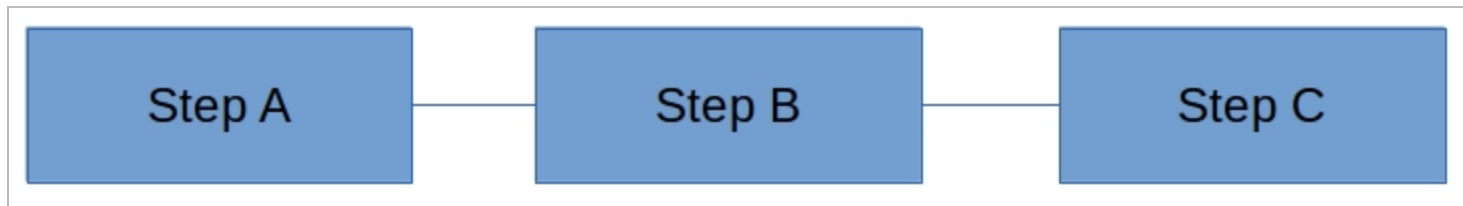
The Life Cycle of a Sample: Stages Versus Steps

As an API programmer, it is important to understand the difference between steps and stages. This distinction is especially important because the concept of stages is hidden from the end user. As such, when receiving requirements from end users, steps sometimes means steps. At other times, steps mean stages. This article highlights the differences between these two entities.

The Life Cycle of a Sample

We tend to think of a protocol as being a linear collection of steps, as shown below.

Figure 1

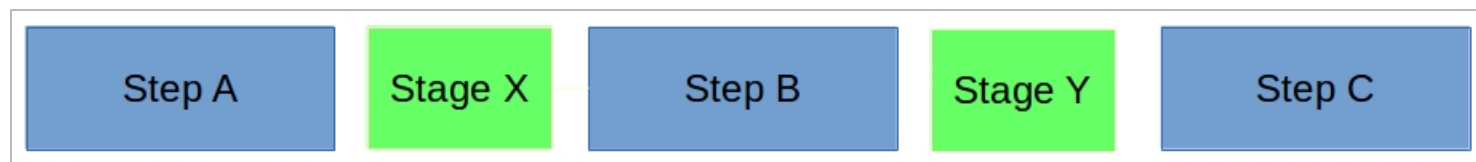


However, this illustration is not complete as the life cycle of a sample modeled within Clarity LIMS reflects what happens in reality. The workflow is broken into periods of activity and inactivity. If a workflow is comprised of three steps (A, B, and C—as shown in Figure B), Step B does not begin at the exact time that Step A is complete.

Stages and Steps

To reflect these inactive periods, Clarity LIMS uses the concept of stages in addition to steps. A more complete representation of a workflow is shown below, with the stages occurring between the steps.

Figure 2



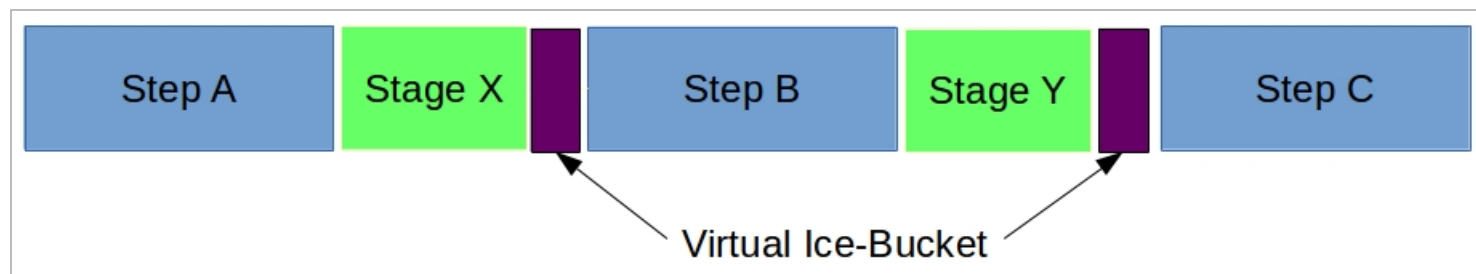
The following phrase simplifies this concept:

If the sample isn't active in a step, it's waiting in a stage.

Note:

The Clarity LIMS concept of the virtual ice bucket is another state that occurs when a sample leaves a stage, but work on the step has not started. This scenario is represented in the Figure 3, with the virtual ice bucket appearing between stages and steps, as the sample moves from left to right. However, virtual ice buckets are largely irrelevant to this discussion. While recognizing their existence, they are discounted from further explanation.

Figure 3



Having simplified our model of a sample passing through a workflow to resemble Figure 2, we can now add the next layer of complexity.

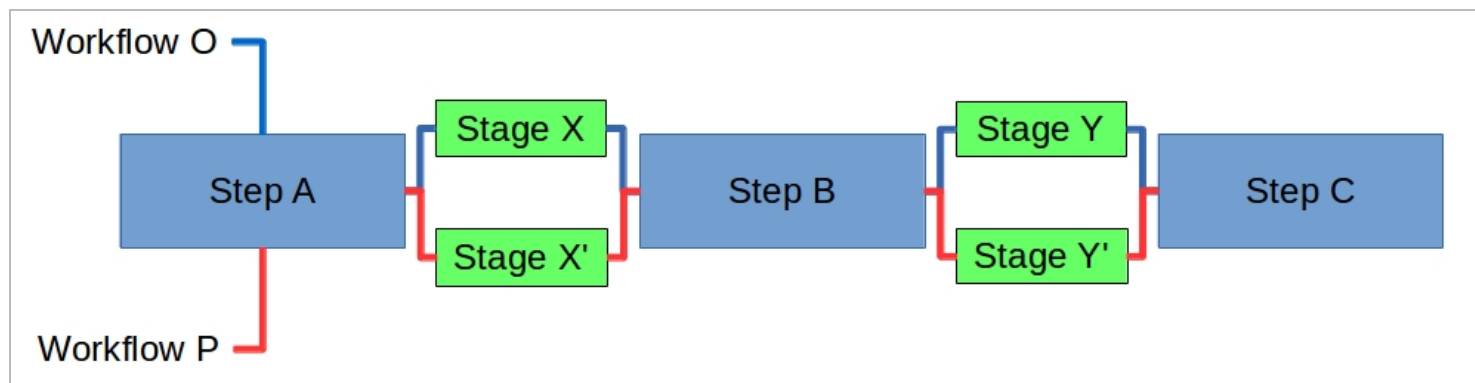
Protocols are components of workflows. As such, it is easy to imagine two or more workflows sharing a protocol. This detail leads to the following summary:

Steps belong to protocols, whereas stages belong to workflows.

This summary means that the stages that exist between steps are part of the workflow (represented in Figure 4 below). For example, samples passing through Workflow O proceed through Step A, Stage X, Step B, Stage Y, and Step C.

Samples passing through Workflow P (which shares the protocol with Workflow O) pass through the same steps. However, samples pass through a different set of stages (Stage X' and Stage Y').

Figure 4



Why Stages Are Important

Looking at the counts of samples associated with steps in a protocol (for example, in the Lab View dashboard in Clarity LIMS), the number of samples awaiting a particular step is actually the total number of samples across all relevant stages that feed into the step.

For bioinformaticians and programmers who are using the Clarity LIMS API, stages have an additional function. Route samples in ways that vary from the expected, linear route by manipulating which stages the artifacts are in. For example, using the API via a script, do the following actions:

- Implement a forking workflow by assigning artifacts to one (or more) additional stages.
- Create iterative (or looping) workflows by routing artifacts to an earlier stage for additional work.

Integrating Scripts

The BaseSpace Clarity LIMS Rapid Scripting™ API adapts lab informatics using the Clarity LIMS platform.

It is important to integrate scripting into the overall processes. Begin by identifying any areas that may require adaptation to fit the lab workflow. It also helps if users are involved in the early stages of the software system analysis process.

Most scripts in an implementation are finalized towards the end of the process, as the full impact and benefits of the new system become clear.

Take some time to become familiar with the user interface, learn how to configure the product, and work with the tools that the lab uses. Also, establish the workflows and the configuration of the system before investing in API scripts and automation.

Start with Administrator Training

New customers receive administrator-level training before working with the API.

If you are not comfortable configuring steps, custom fields, containers, etc., in Clarity LIMS, you may find the API material difficult to understand. Contact Illumina for more information on administrator training and training materials.

Define the Lab Solution

Before committing time and resources to using the API, it is important to define what you would like to accomplish. Understanding the key outcomes, use cases, users, and constraints of the lab helps with learning the API more quickly and improves efficiency.

If you require assistance, Illumina can provide expert resources to audit and analyze the laboratory users, processes, workflows, instrumentation, data production, and environment. This careful and focused analysis results in a requirements specification that provides extensive value to the facility.

Related Resources

- [Useful Tools](#)
- [Suggested Reading](#)

Automation Overview

To trigger scripts and third-party programs from the BaseSpace Clarity LIMS user interface, use command-line calls configured in step automations.

There are many ways to use automation in Clarity LIMS. Consider the following examples:

- Automate sample tracking and enhance the information recorded.
- Generate and attach specially formatted text files.
- Simplify data entry.
- Automate the population or updating of data fields and data files.

Before using automation, become familiar with the topics discussed in this article and understand how automation functionality interacts with users, Clarity LIMS, and REST.

As of BaseSpace Clarity LIMS v5.0, several terms have been deprecated:

- External Program Integration Plug-in (EPP) has been replaced with automation.
- Automated Informatics (AI) node has been replaced with Automation Worker (AW) node.
- Parameter has been replaced with token.

Common Applications of Automation

Automation scripts are often used to create and attach specially formatted text files automatically, such as the following examples:

- Files containing sample lists (sometimes called instrument driver files), which users can import into control software to save time and ensure accurate sample processing.
- Barcode or label files, which are specially formatted files that can be supplied to barcode software systems to allow users to print container labels.
- Summary analysis results, for example, from alignment or molecular identification and quantification algorithms.

The two common applications of automation are:

- To create files and attach them to process output placeholders.
- To update data fields with information created during data analysis.

Scripts triggered by automation also use REST to update information directly within the REST resources.

For details, see [REST Web Services](#) and the version-specific documentation in the following sections:

- [Automation/EPP Parameters \(LIMS v4.x & earlier\)](#)
- [Automation Tokens](#)

How Users Trigger Scripts

Record lab work in Clarity LIMS by running steps on samples. These steps may be configured in Clarity LIMS by an administrator.

Most steps can be configured with an automation trigger that invokes an external script. The script may include fixed and variable information parameters/tokens on the command line.

 | As of BaseSpace Clarity LIMS v5.0, the term command-line parameter has been replaced with token.

Configuring Processes/Steps to Trigger Scripts

After an AI/AW node is installed, processes (in Clarity LIMS v4.2 and earlier) or steps (Clarity LIMS v5 and later) must be configured to call out to it.

This configuration is executed by the Clarity LIMS administrator:

- In Clarity LIMS v4.2 and earlier, execute configuration in the Operations Interface process configuration dialog on the External Programs tab.
- In Clarity LIMS v5 and later, execute configuration on the Automation configuration screen.

When configuring an automation, the following values must be defined:

- Name
- Channel
- Command-line call
- Trigger style and location (see also [Automation Triggers and Command Line Calls](#))

 | Only a brief summary of automation configuration is provided here. This material should be familiar from the Clarity LIMS administration training.

Configuring Scripts that are Called by Automation

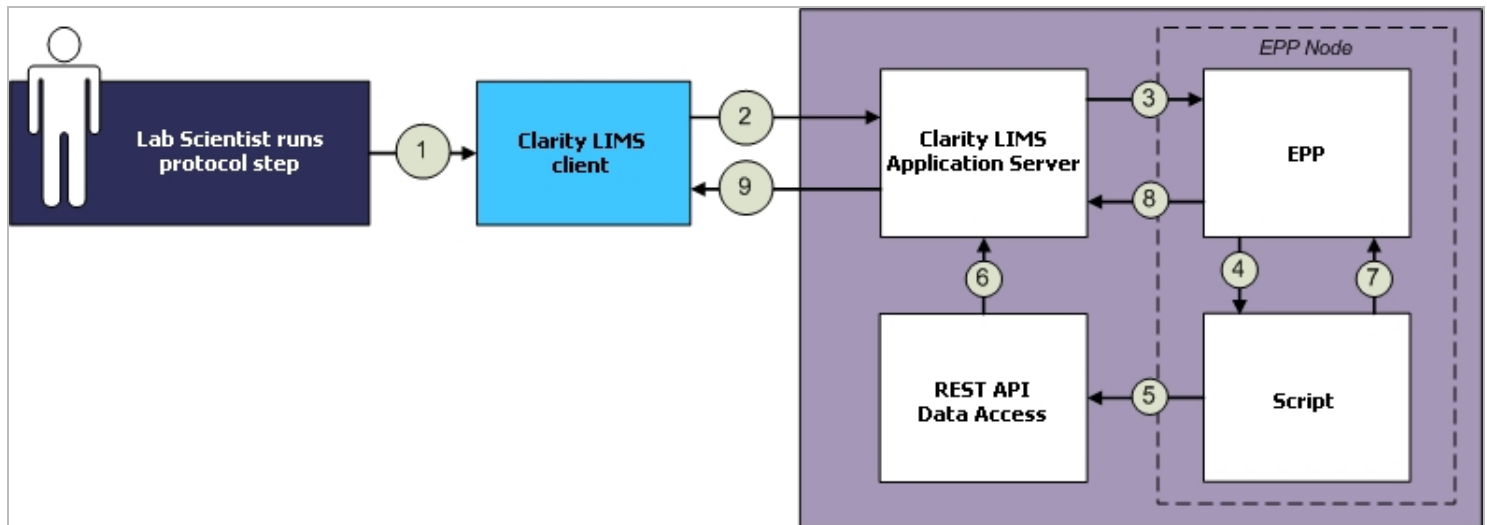
Scripts or third-party programs are called using the operating system command line. They must meet the following requirements:

- Be callable on the command line and, preferably, be able to read and respond to command-line parameters.
- Be accessed by the user account running the automation, with appropriate permissions and disk locations.
- Be able to exit with appropriate exit codes, otherwise the automation may record the script completed with an error.

How Information Flows Between a User and a Script

- *Step 1: A User Runs a Step*
- *Step 2: Server Creates the New Process/Step*
- *Step 3: The Automation Invokes the Operating System Shell Command*
- *Steps 4 to 7: The External Program Executes (Optionally Calling REST Resources)*
- *Step 8: The Automation Returns (Optionally Attaching Created Files)*
- *Step 9: The User Accesses Updated Results*

The following diagram illustrates what happens when a step is run in Clarity LIMS.



Step 1: A User Runs a Step

The lab scientist tracks activities by running a step in Clarity LIMS. The step is configured to display a button that invokes the configured automation script.

Step 2: Server Creates the New Process/Step

i | As of BaseSpace Clarity LIMS v5.0, the term parameter has been replaced with token.

Common Applications of Automation

The application server creates a new step, which is much like POSTing to the processes resource of the REST API. The server resolves any parameters/tokens found in the string and sends the resolved command-line string to the automation.

In this example, two of the most common parameters/tokens used when working with automation are discussed:

- **{processURI:version:scheme}**—This parameter/token passes the REST API URI of the step that issued the command-line string.
- **{outputFileN}**—This parameter/token passes the LIMS ID of the specified expected output file of the step that issued the command-line string. You can use 0 (zero) for the first file, 1 (one) for the second file, etc.

For more information about the parameters/tokens available for use, refer to the articles in the following API documentation sections:

- [Automation/EPP Parameters \(LIMS v4.x & earlier\)](#)
- [Automation Tokens](#)

 As of API v1 r12, the version and scheme values for {processURI:version:scheme} are automatically populated, based on the REST version and protocol of the deployed server.

Step 3: The Automation Invokes the Operating System Shell Command

The automation program receives the command-line string from the application server. It may also receive other process(step)-related information, such as temporary files. The command-line string is executed by the operating system of the host computer.

Steps 4 to 7: The External Program Executes (Optionally Calling REST Resources)

The automation can work with any third-party program that supports command-line parameters/tokens. The program may simply create files or it may manipulate information directly via the REST API (steps 5 and 6).

Simple automation operation does not require anything of the REST API. If a third-party program creates files that users would like brought back into Clarity LIMS, scripts should use the outputFileN parameter/token to specify that the program create file names that are expected by the client. The files are placed in a temporary local working directory and automatically imported into the client. With this method, the automation automatically handles many of the things that would need manually scripts using the REST API processes, artifacts, files, and glsstorage resources.

For more complicated scenarios, you may want to use automation with the REST API. This situation is where the processURI parameter/token is used. A GET request on the URI of the step that issues the command-line string provides all the information recorded by the user. This information includes links to the analytes (samples) used as inputs. The script can then use other REST API resources to create or update information.

On completion, the third-party program exits (step 7). Standard shell exit codes apply: zero (0) equals successful completion.

Step 8: The Automation Returns (Optionally Attaching Created Files)

On exit of the third-party program, the automation software updates the application server. If the system finds files with names that match the file placeholders produced by a process/step, the files are uploaded to the file server and attached to the appropriate placeholders.

A nonzero exit code sets a flag on the step, indicating that there is an error.

Step 9: The User Accesses Updated Results

With updates complete, the application server sends refresh events to Clarity LIMS. The user sees that files have been uploaded.

Verifying Automation Scripts

Verifying and testing scripts is an important part of working with automation. Remember that there are three software components:

- The server
- The automation instance (calling scripts)
- The script

The best way to debug scripts is to unit test each component separately. For example, a logical order to work on a script is as follows.

1. Define and test the REST calls required in a web browser.
2. Define the command-line parameters/tokens sent to the automation at step completion.
3. Test the script running from the command line.
4. Test automation calls to the script at step completion by running the step from the Clarity LIMS interface.

Installing an Automation Worker/AI Node

Before the system can use the automation, a system administrator installs one or more automation workers/AI nodes within the lab network. The installation typically occurs on the server that contains the script program, or third-party application, to be integrated.

The installer program is contained in the Automated Informatics/Automation Worker software package.

Automation Overview

To trigger scripts and third-party programs from the BaseSpace Clarity LIMS user interface, use command-line calls configured in step automations.

There are many ways to use automation in Clarity LIMS. Consider the following examples:

- Automate sample tracking and enhance the information recorded.
- Generate and attach specially formatted text files.
- Simplify data entry.
- Automate the population or updating of data fields and data files.

Before using automation, become familiar with the topics discussed in this article and understand how automation functionality interacts with users, Clarity LIMS, and REST.

As of BaseSpace Clarity LIMS v5.0, several terms have been deprecated:

- External Program Integration Plug-in (EPP) has been replaced with automation.
- Automated Informatics (AI) node has been replaced with Automation Worker (AW) node.
- Parameter has been replaced with token.

Common Applications of Automation

Automation scripts are often used to create and attach specially formatted text files automatically, such as the following examples:

- Files containing sample lists (sometimes called instrument driver files), which users can import into control software to save time and ensure accurate sample processing.
- Barcode or label files, which are specially formatted files that can be supplied to barcode software systems to allow users to print container labels.
- Summary analysis results, for example, from alignment or molecular identification and quantification algorithms.

The two common applications of automation are:

- To create files and attach them to process output placeholders.
- To update data fields with information created during data analysis.

Scripts triggered by automation also use REST to update information directly within the REST resources.

For details, see [REST Web Services](#) and the version-specific documentation in the following sections:

- [Automation/EPP Parameters \(LIMS v4.x & earlier\)](#)
- [Automation Tokens](#)

How Users Trigger Scripts

Record lab work in Clarity LIMS by running steps on samples. These steps may be configured in Clarity LIMS by an administrator.

Most steps can be configured with an automation trigger that invokes an external script. The script may include fixed and variable information parameters/tokens on the command line.

 | As of BaseSpace Clarity LIMS v5.0, the term command-line parameter has been replaced with token.

Configuring Processes/Steps to Trigger Scripts

After an AI/AW node is installed, processes (in Clarity LIMS v4.2 and earlier) or steps (Clarity LIMS v5 and later) must be configured to call out to it.

This configuration is executed by the Clarity LIMS administrator:

- In Clarity LIMS v4.2 and earlier, execute configuration in the Operations Interface process configuration dialog on the External Programs tab.
- In Clarity LIMS v5 and later, execute configuration on the Automation configuration screen.

When configuring an automation, the following values must be defined:

- Name
- Channel
- Command-line call
- Trigger style and location (see also [Automation Triggers and Command Line Calls](#))

i | Only a brief summary of automation configuration is provided here. This material should be familiar from the Clarity LIMS administration training.

Configuring Scripts that are Called by Automation

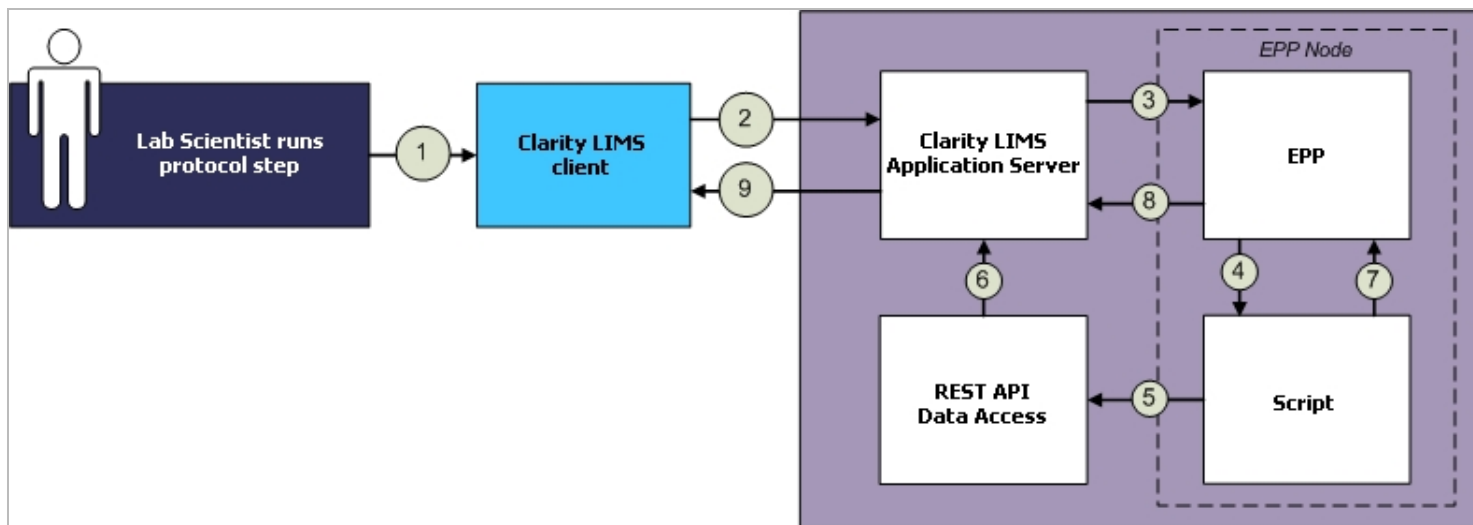
Scripts or third-party programs are called using the operating system command line. They must meet the following requirements:

- Be callable on the command line and, preferably, be able to read and respond to command-line parameters.
- Be accessed by the user account running the automation, with appropriate permissions and disk locations.
- Be able to exit with appropriate exit codes, otherwise the automation may record the script completed with an error.

How Information Flows Between a User and a Script

- [Step 1: A User Runs a Step](#)
- [Step 2: Server Creates the New Process/Step](#)
- [Step 3: The Automation Invokes the Operating System Shell Command](#)
- [Steps 4 to 7: The External Program Executes \(Optionally Calling REST Resources\)](#)
- [Step 8: The Automation Returns \(Optionally Attaching Created Files\)](#)
- [Step 9: The User Accesses Updated Results](#)

The following diagram illustrates what happens when a step is run in Clarity LIMS.



Step 1: A User Runs a Step

The lab scientist tracks activities by running a step in Clarity LIMS. The step is configured to display a button that invokes the configured automation script.

Step 2: Server Creates the New Process/Step

i | As of BaseSpace Clarity LIMS v5.0, the term parameter has been replaced with token.

Common Applications of Automation

The application server creates a new step, which is much like POSTing to the processes resource of the REST API. The server resolves any parameters/tokens found in the string and sends the resolved command-line string to the automation.

In this example, two of the most common parameters/tokens used when working with automation are discussed:

- **{processURI:version:scheme}**—This parameter/token passes the REST API URI of the step that issued the command-line string.
- **{outputFileN}**—This parameter/token passes the LIMS ID of the specified expected output file of the step that issued the command-line string. You can use 0 (zero) for the first file, 1 (one) for the second file, etc.

For more information about the parameters/tokens available for use, refer to the articles in the following API documentation sections:

- [Automation/EPP Parameters \(LIMS v4.x & earlier\)](#)
- [Automation Tokens](#)

i | As of API v1 r12, the version and scheme values for {processURI:version:scheme} are automatically populated, based on the REST version and protocol of the deployed server.

Step 3: The Automation Invokes the Operating System Shell Command

The automation program receives the command-line string from the application server. It may also receive other process(step)-related information, such as temporary files. The command-line string is executed by the operating system of the host computer.

Steps 4 to 7: The External Program Executes (Optionally Calling REST Resources)

The automation can work with any third-party program that supports command-line parameters/tokens. The program may simply create files or it may manipulate information directly via the REST API (steps 5 and 6).

Simple automation operation does not require anything of the REST API. If a third-party program creates files that users would like brought back into Clarity LIMS, scripts should use the `outputFileN` parameter/token to specify that the program create file names that are expected by the client. The files are placed in a temporary local working directory and automatically imported into the client. With this method, the automation automatically handles many of the things that would need manually scripts using the REST API processes, artifacts, files, and `glsstorage` resources.

For more complicated scenarios, you may want to use automation with the REST API. This situation is where the `processURI` parameter/token is used. A GET request on the URI of the step that issues the command-line string provides all the information recorded by the user. This information includes links to the analytes (samples) used as inputs. The script can then use other REST API resources to create or update information.

On completion, the third-party program exits (step 7). Standard shell exit codes apply: zero (0) equals successful completion.

Step 8: The Automation Returns (Optionally Attaching Created Files)

On exit of the third-party program, the automation software updates the application server. If the system finds files with names that match the file placeholders produced by a process/step, the files are uploaded to the file server and attached to the appropriate placeholders.

A nonzero exit code sets a flag on the step, indicating that there is an error.

Step 9: The User Accesses Updated Results

With updates complete, the application server sends refresh events to Clarity LIMS. The user sees that files have been uploaded.

Verifying Automation Scripts

Verifying and testing scripts is an important part of working with automation. Remember that there are three software components:

- The server
- The automation instance (calling scripts)

- The script

The best way to debug scripts is to unit test each component separately. For example, a logical order to work on a script is as follows.

1. Define and test the REST calls required in a web browser.
2. Define the command-line parameters/tokens sent to the automation at step completion.
3. Test the script running from the command line.
4. Test automation calls to the script at step completion by running the step from the Clarity LIMS interface.

Installing an Automation Worker/AI Node

Before the system can use the automation, a system administrator installs one or more automation workers/AI nodes within the lab network. The installation typically occurs on the server that contains the script program, or third-party application, to be integrated.

The installer program is contained in the Automated Informatics/Automation Worker software package.

Automation Triggers and Command Line Calls

As of BaseSpace Clarity LIMS v5, the Operations Interface Java client, which is used by administrators to configure processes, consumables, user-defined fields, and users, has been deprecated. All configuration and administration tasks are now executed in the Clarity LIMS web interface.

In addition, several terms have been deprecated:

- External Program Integration Plug-in (EPP) has been replaced with automation.
- EPP/Automated Informatics (AI) node has been replaced with automation worker/AW node.
- Parameter has been replaced with token.

Use step automations to trigger a command-line call on a process/step or a file attachment event. The steps required differ depending on the LIMS version.

This article provides an overview of the steps required to configure automations and automation triggers. For detailed version-specific instructions, see the following documentation:

- Clarity LIMS v5: [Add and Configure Automations](#)
- Clarity LIMS v4.2 and earlier: *Configuring Protocol Steps*

Configuration (LIMS v5 and later)

1. On the main menu bar, select **Configuration**, then select the Automation tab.
2. On the Automation configuration screen on the Step Automation tab, add a new automation:
 - Name the automation.
 - Set the channel name.
 - Define the command line.

- Enable the automation on the desired steps.
3. On the Master Step Settings or Step Settings screen of the related step, set the following:
 - **Trigger Location**—The stage at which the script it is to be initiated (beginning of step, end of step, on entry to/exit from a screen, etc.).
 - **Trigger Style**—How the script is to be initiated (automatically or manually when the user selects a button in the interface).

Configuration (LIMS v4.2 and earlier)

When running processes and protocol steps, note the following details:

- In the Clarity LIMS Operations Interface, processes are run on samples (analytes) or files.
- In the Clarity LIMS web interface, steps are run on samples only.

As of Clarity LIMS v5, the Operations Interface Java client has been deprecated.

1. In the Operations Interface, Select **Configuration > Processes > Process Types** and select the desired process.
2. On the External Programs tab, select the **Process is associated with an external program** checkbox, then add a new process parameter:
 - Name the parameter.
 - Set the channel name.
 - Set the trigger mode (the default PostProcess trigger mode is recommended, see the following note).
 - Define the command-line call.

i | While it is possible to set the trigger mode to PostProcess after each file attachment event, this mode is not supported in the Clarity LIMS web interface. As a result, it is not recommended. For details, see [Advanced Mode: Multiple Triggers on Process Updates Including Each Input File Attachment](#) .

3. In the Clarity LIMS web interface in the configuration area, modify the configuration of the step associated with the process previously configured. On the Automation tab, set the following:
 - **Behavior**—How the script is to be initiated.
 - **Stage of Step**—The stage at which the script it is to be initiated.
 - **Timing**—When the script is initiated.

Configuring Trigger Modes in the Operations Interface

In the Operations Interface, a client process can be run on empty file placeholders many hours or days before that data is available. The client automatically dispatches and triggers the process completion event when the data becomes available.

When configuring a process, the trigger mode is controlled by the option chosen in the Add Process Parameter dialog, specifically the Trigger Mode drop-down list. There are two options to choose from:

- PostProcess (default mode)
- PostProcess after each file attachment event (advanced mode)

Default Mode: One Trigger when Process Completes and All Files are Attached

This mode is enabled when the PostProcess option is selected. This option is the default mode.

For event timing, consider the following details:

1. LIMS web interface:
 - When running a step on samples, trigger the script by selecting a button in the interface. The script must be able to handle multiple invocations because users may select the button more than one time.
2. Operations interface:
 - A process/step with sample artifact inputs that triggers the script after process/step completes.
 - A process with file artifact inputs that triggers the script when all empty file placeholders have been populated with attached input files.
 - A process run on previously populated file placeholders that triggers the script after a process/step completes.
 - In this mode, the automation can transfer input files to the working directory for processing. Files created by the script in the temporary directory can be automatically attached to the output file placeholders.

Advanced Mode: Multiple Triggers on Process Updates Including Each Input File Attachment

This mode is enabled when the PostProcess after each file attachment event option is selected. This mode is the advanced, or verbose, mode.

- The PostProcess after each file attachment event mode is only supported in the Operations Interface. It is for advanced users, because the scripts triggered require more complexity.
- The Run each time a file attachment event occurs advanced mode requires REST. This mode requires the script to transfer files and limits the command-line parameter strings. The REST process, artifact, and file resources are used with this trigger.

For event timing, consider the following details:

- A process with sample artifact inputs triggers the completion event after a process completes.
- A process with file artifact inputs triggers each time a file placeholder is populated with an attached input file.
- The last input file attachment is the final trigger event. No other triggers fire after all file placeholders are populated, even if a file is later deleted and reimported or other process modifications occur.
- A process run on previously populated file placeholders triggers the completion event after a process/protocol step completes.

When attaching output files, consider the following information:

As there are many events being handled by the script in this mode, no assumptions are made about when to transfer input or output files. Therefore, no automatic attachment of output files is available in this mode.

Use the REST API to attach files (see [Attach a File to a File Placeholder with REST](#) for more information).

Additional Resources

For more information, see [API Training Videos](#).

Automation Execution Environment

As of BaseSpace Clarity LIMS v5.0, several terms have been deprecated:

- External Program Integration Plug-in (EPP) has been replaced with automation.
- EPP/AI node has been replaced with automation worker/AW node.
- Parameter has been replaced with token.
- User-defined field (UDF) has been replaced with custom field.

When a job is dispatched to the AI node/automation worker, the following steps occur:

1. A temporary working directory is created on the AI node/automation worker.
 - In AllInstallDirectory/temp/
 - With a unique name including the client process LIMS ID
2. The command configured and selected as part of the step run in Clarity LIMS is then sent to the AI node/automation worker. Any specified parameters/tokens are replaced with actual values.
3. The command is executed on the AI node/automation worker, generating step execution using the temporary working directory as the working directory context.
 - Script processing can use stdout, stderr, and return codes following standard shell programming packages.
4. When the script exits, the AI node/automation worker automatically retrieves any files with matching LIMS IDs from the temporary working directory. The files are attached to the appropriate output file placeholders.

How the API Infrastructure is Used

The automation API infrastructure can be used alone or with the REST API infrastructure.

For example:

- Simple scripts can use automation parameters/tokens and data files directly from the current working directory. They can write results back to the current working directory, associating them back to the relevant placeholders in Clarity LIMS.
- More advanced scripts can also use the REST API infrastructure to retrieve additional required information and place relevant data back into UDFs/custom fields. Advanced scripts can also attach and associate data files to placeholders, which may be in different locations, while the script is still running.

Supported Command Line Interpreters

LIMS automations typically call scripts or third-party programs written for a shell or command-line interpreter, of either a Linux or Windows operating system (OS). Although the use of any system shell is acceptable, Bash is recommended.

Depending on the systems that integrate with the given automations, various restrictions apply to the string parameters/tokens and formatting used in the automation command line.

 For BaseSpace Clarity LIMS v5, several terms are deprecated:

- Automation replaced External Program Integration Plug-in (EPP).
- Automation worker / AW node replaced EPP/AI node.
- Token replaced Parameter.

Environment Variables

Environment variables can be used to aid in configuration. However, automation commands are generated with a limited shell. For full access to environment variables, the recommended practice is to start the command to instantiate a 'full' user shell. For example, for Bash use the following command:

```
bash -l -c
```

This procedure provides the following advantages:

- Ensures updates of the environment variables, removing the need for repeated AI node/automation worker restarts.
- Ensures access to all environment variables, including the full path to Groovy.
- Allows certainty of the shell being used.

Operating system shell formatting, spaces, and special characters

The various operating system (OS) shells each have their own rules and regulations. When creating command-line strings, be aware of the considerations described in the following sections.

Operating System Shell Formatting Differences

Windows shell command-line interpreters require different syntax and formatting than Linux shell variants. For example, the following scripts are identical, but are formatted for different AI nodes/automation workers running on different operating systems.

On Linux

```
groovy -cp ../../scripts ../../scripts/flexcontrol_drivergen.groovy -l {processU:v1:http} -a {artifactsU:v1:http} -u {username} -p {password} -f {compoundOutputFile0}.txt
```

On Windows

```
C:\Windows\System32\cmd /c "groovy -cp ../../scripts ../../scripts/flexcontrol_drivergen.groovy -l {processU:v1:http} -a {artifactsU:v1:http} -u {username} -p {password} -f {compoundOutputFile0}.txt"
```

Most of the examples in this specification use Windows formatting, because Windows is the most common platform found in the lab.

Spaces

Spaces in paths, file names, or parameter/token data can cause commands to be misinterpreted as information passes between systems.

Many OS shells automatically parse command-line contents by space, which cannot be what is intended. Enclose commands in double quotes " " to avoid misinterpretation of spaces by the OS shell command-line interpreter.

Special Characters

OS shell command-line interpreters can attempt to interpret and act upon certain special characters, rather than passing them along as textual information. A character can have a rule applied to it within one OS shell environment, and a different rule under another environment. To use a character in its literal form, escape the characters. The escape character used varies depending on your OS shell. The most common escape character is the backslash character.

The most common OS shell characters that require escaping are:

```
% ( ) / \ * & |
```

 To make sure that a configured command in the client is properly interpreted, test it on the AI node/automation worker machine command line.

Automation Channels

The automation and integration of the day-to-day work in the lab requires different Automated Informatics (AI) nodes/automation workers to perform different tasks.

For BaseSpace Clarity LIMS v5.0, several terms are deprecated:

- Automation replaced External Program Integration Plug-in (EPP).
- Automation Worker (AW) node replaced EPP/AI node.

Specifying Channels

Channels are manually named, and ideally clearly represent the task performed (eg. Type_2_Analysis). To make sure that dispatched automation work is routed to the correct destination, specify a channel in the following places:

- On the AI node/automation worker

- LIMS v5 and later: When configuring step and derived sample automations on the Automation tab
- LIMS v4.2 and earlier:
 - In the Operations Interface, when configuring process parameters on the External Programs tab
 - In the web interface, when configuring derived sample automations.

How Channels Work

1. When the automation trigger conditions are met in the LIMS, the automation job first enters a channel-specific 'first in, first out' (FIFO) queue of work for completion.
2. Jobs queue in this channel until one of the AI nodes/automation workers operating on the channel completes its previous work, and indicates it is free to accept more.
3. The next job is then dispatched from the channel queue to the node. This strategy allows a single channel queue to receive service by one, or many, AI nodes/automation workers servicing the specified channel.

It is possible to have multiple AI nodes/automation workers performing the same type of work all configured on the same channel, allowing a simple but effective way to increase throughput of a particular analysis bottleneck, or to ensure redundancy during a single node failure.

Error Handling

Scripts produce a numeric code on exit. By convention and by default, a successful exit has a code of 0 (zero). Within error handling, select different nonzero exit codes to indicate various error conditions.

 | For BaseSpace Clarity LIMS v5.0, the term External Program Integration Plug-in (EPP) is deprecated and replaced with automation.

Script Error Logging and Standard Streams

Though logging information in the BaseSpace Clarity LIMS user interface is useful, scripts can also write debugging/troubleshooting information into the `automatedinformatics.log` file using `stderr` (standard error). When a line is printed to `stderr`, a [WARN] line is written to this log, which is useful for troubleshooting interactions between the script and the automation programs. For more information on the `automatedinformatics.log` file, see [Troubleshooting Automation](#). For more information on using `stderr` on the command line, refer to Unix and Windows documentation on standard I/O streams, especially standard error.

Error Handling

In BaseSpace Clarity LIMS, the last line written to `stdout` (standard out) is automatically captured and shown in the interface.

If the exit code is zero, the message displays in green. If the exit code is nonzero, the message displays in red.

The Operations Interface Java client is deprecated in BaseSpace Clarity LIMS v5. All configuration and administration tasks are currently executed in the LIMS web interface.

If the script exits with a nonzero code, a sample genealogy flag is automatically added to the process outputs, with a standard error message indicating there is an External Program Error.

If the script is complex, or includes several error conditions, configure an additional result file process output (eg result.log) designed to capture status and error information. Make the external script write additional information to this file. If an error occurs, the file is still captured in the client, and is available to view for troubleshooting purposes.

Automation Tokens

This section provides information to help you work with Clarity LIMS automation tokens in Clarity LIMS v5 and later.

Derived Sample Automation Tokens

When configuring automations in the BaseSpace Clarity LIMS, copy tokens from the Tokens list and paste them into the Command-Line field.

These tokens are available for use in derived sample automations. If using multiple variables, add a space between each entry. All tokens and parameters are case-sensitive.

Token	Purpose	Example
{username}	Supplies the username of the current user running the step to the triggered automation script.	<pre>cmd /c "C:\ai\ai.bat {username}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat adminuser</pre>
{password}	Supplies the password of the current user running the step to the triggered automation script.	<pre>cmd /c "C:\ai\ai.bat {password}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat 3BlindMice</pre> In log files, the password supplied on the command line is replaced with a series of *** characters.
{baseURI}	Supplies the base API URI to the triggered automation script.	<pre>cmd /c "C:\ai\ai.bat {baseURI}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat https://lims.lan.29/api</pre>
{derivedSampleLuids}	Supplies the derived sample LIMS IDs to the triggered automation script.	<pre>cmd /c "C:\ai\ai.bat {derivedSampleLuids}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat 2-1641 2-1642 2-1643</pre>

{userinput:customParameterName}	Allows for data input to supply the triggered automation script. Custom parameters are identified with the prefix 'userinput:'	The following command line requires the user to input a value for 'more_yield': yieldsript.sh -y {userinput:more_yield} -u {username}
---------------------------------	--	--

Step Automation Tokens

When configuring automations in BaseSpace Clarity LIMS, copy tokens from the Tokens list and paste them into the Command Line field.

These tokens are available for use in step automations. If using multiple variables, add a space between each entry. All tokens and parameters are case-sensitive.

Token	Purpose	Example
{username}	Supplies the username of the current user running the step to the triggered automation script	cmd /c "C:\ai\ai.bat {username}" resolves to: cmd /c C:\ai\ai.bat adminuser
{password}	Supplies the password of the current user running the step to the triggered automation script.	cmd /c "C:\ai\ai.bat {password}" resolves to: cmd /c C:\ai\ai.bat 3BlindMice In log files, the password supplied on the command line is replaced with a series of *** characters.
{baseURI}	Supplies the base API URI to the triggered automation script.	cmd /c "C:\ai\ai.bat {baseURI}" resolves to: cmd /c C:\ai\ai.bat https://lims.lan.29/api
{stepURI}	Supplies the URI of the step to the triggered automation script. Include the version parameter (ie, {stepURI:version}) to specify the version of the REST API to be accessed.	cmd /c "C:\ai\ai.bat {stepURI:v2}" resolves to: cmd /c C:\ai\ai.bat https://yourServerNameOrIP/api/v2/steps/CAM-CSB-100212-24-197

Token	Purpose	Example
{artifactsURI}	Supplies the URI of the artifacts root to the triggered automation script. Include the version parameter (ie, {artifactsURI:version}) to specify the version of the REST API to be accessed.	<pre>cmd /c "C:\ai\ai.bat {artifactsURI:v2}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat https://yourServerNameOrIP/api/v2/artifacts</pre>
{processURI} {stepURI} token is preferred. {processURI} is deprecated and less accurate. May be removed in future versions.	Supplies the URI of the step to the triggered automation script. If using the deprecated {processURI} token, the addition of the version and scheme parameters is recommended ({processURI:version:scheme}). Adding the version and scheme reduces the chance of a server and REST version upgrade unknowingly affecting your scripts.	<pre>cmd /c "C:\ai\ai.bat {processURI:v2:http}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat https://yourServerNameOrIP/api/v2/processes/CAM-CSB-100212-24-197</pre>
{processLuid}	Supplies the LIMS ID of the step that triggered the automation script.	<pre>cmd /c "C:\ai\ai.bat {processLuid}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat CAM-CSB-100212-24-169</pre>
{udf:nameOfUDF}	Supplies the current value stored within a UDF configured as nameofUDF.	<pre>cmd /c "C:\ai\ai.bat {udf:injection_volume}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat 12.4</pre>

Token	Purpose	Example
{parentProcessUdf:nameOfUDF}	Supplies the current value stored within a UDF configured as nameofUDF of the immediate parent step to the step that triggered the automation script. The parent step must provide the inputs (derived samples) to the step. In cases where there are multiple parents (ie, the inputs are derived from various steps) only the first of these parents is returned.	<pre>cmd /c "C:\ai\ai.bat {parentProcessUdf:RunID}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat RUN_BW1765</pre>
{parentProcessUdfN:nameOfUDF}	Supplies the current value stored within a UDF configured as nameofUDF of the immediate parent step to the step that triggered the automation script. The parent step must provide the inputs (derived samples) to the step. In cases where there are multiple parents (ie, the inputs are derived from various steps) all parent step IDs are treated as an array-based list. N specifies the array list index position 0..n of the desired step. {parentProcessUdf0:nameofUDF} is equivalent to {parentProcessUdf:nameofUDF}.	<pre>cmd /c "C:\ai\ai.bat {parentProcessUdf1:RunID}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat RUN_HJ1865</pre>

Token	Purpose	Example
{outputFileLuids}	Supplies the LIMS IDs of all step output file placeholders.	<pre>cmd /c "C:\ai\ai.bat {outputFileLuids}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat "BAR103A1CO248" "BAR103A1CO249" "BAR103A1CO250" "BAR103A1CO251" "BAR103A3CO158" "BAR103A3CO159" "BAR103A3CO160" "BAR103A3CO161"</pre>
{outputFileLuidN}	Supplies the LIMS ID for the specified step output file placeholder. All output file placeholders applying to inputs of the step are treated as an array-based list, where N specifies the array list index position [0..n] of the desired file.	Assuming the same eight output files as in the previous example: <pre>cmd /c "C:\ai\ai.bat {outputFileLuid0}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat BAR103A1CO248</pre> <pre>cmd /c "C:\ai\ai.bat {outputFileLuid1}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat BAR103A1CO249</pre> <pre>cmd /c "C:\ai\ai.bat {outputFileLuid7}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat BAR103A3CO161</pre>
{compoundOutputFileLuids}	Supplies the LIMS IDs for all shared step output file placeholders.	<pre>cmd /c "C:\ai\ai.bat {compoundOutputFileLuids}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat "92-527" "92-528" "100-541" "100-544"</pre>

Token	Purpose	Example
{compoundOutputFileLuidN}	Supplies the LIMS ID for the specified step output file placeholder that applies to an individual input. All output file placeholders applying to individual inputs for the step are treated as an array-based list, where *N specifies the array list index position [0..n] of the desired file.	Assuming the same four output files as in the previous example: <pre>cmd /c "C:\ai\ai.bat {compoundOutputFileLuid0}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat 92-527</pre> <pre>cmd /c "C:\ai\ai.bat {compoundOutputFileLuid1}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat 92-528</pre> <pre>cmd /c "C:\ai\ai.bat {compoundOutputFileLuid2}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat 100-541</pre> <pre>cmd /c "C:\ai\ai.bat {compoundOutputFileLuid3}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat 100-544</pre>
Deprecated {parentProcessLuid*} tokens	The following tokens have been deprecated: • {parentProcessLuid} • {parentProcessLuids} • {parentProcessLuidN} These tokens were only applicable to steps that take file inputs. File inputs are no longer supported in the LIMS.	

Project Automation Tokens

When configuring automations in BaseSpace Clarity LIMS, copy tokens from the Tokens list and paste them into the Command Line field.

These tokens are available for use in project automations. If using multiple variables, add a space between each entry. All tokens and parameters are case-sensitive.

Token	Purpose	Example
{username}	Supplies the username of the current user running the step to the triggered automation script	<pre>cmd /c "C:\ai\ai.bat {username}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat adminuser</pre>
{password}	Supplies the password of the current user running the step to the triggered automation script.	<pre>cmd /c "C:\ai\ai.bat {password}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat 3BlindMice</pre> In log files, the password supplied on the command line is replaced with a series of *** characters
{baseURI}	Supplies the base API URI to the triggered automation script.	<pre>cmd /c "C:\ai\ai.bat {baseURI}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat https://lims.lan.29/api/</pre> <pre>cmd /c "C:\ai\ai.bat {baseURI}v2"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat https://lims.lan.29/api/v2</pre> Note: To access the endpoints, make sure that the {baseURI} is appended with v2. You can include this in the token in the command line, as shown above, or in the script itself.
{projectLuid}	Supplies the URI of the step to the triggered automation script. Include the version parameter (ie, {stepURI:version}) to specify the version of the REST API to be accessed.	<pre>cmd /c "C:\ai\ai.bat {C:\ai\ai.bat {projectLuid}"</pre> resolves to: <pre>cmd /c C:\ai\ai.bat ADM123</pre>

Automation/EPP Parameters (LIMS v4.x & earlier)

This section provides information to help you work with Clarity LIMS automation/EPP parameters in LIMS v4.x and earlier.

Automation/EPP Parameters and REST (LIMS v4.x)

The main purpose of automation/External Program Integration Plug-ins (EPP) is to trigger command-line scripts from process runs. The script parameters provide most of the common fields and file handling required; however, REST provides much richer information access.

- The {stepURI:version} and {artifactsURI:version} parameters are used to supply the external script with REST API entry points.
- The {username} and {password} parameters are also discussed here as they are often used in REST-enabled scripts.

stepURI:version

{stepURI:version}

Purpose: Supplies a deployment-specific REST API URI to the step that triggered the script.

- `version` is used to specify the version of the REST API to be accessed (eg. v2).

 `version` is case-sensitive and should be specified in lowercase.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {stepURI:v2}"
```

the parameter is replaced with:

```
cmd /c C:\ai\ai.bat https://yourServerNameOrIP/api/v2/steps/CAM-CSB-100212-24-197
```

artifactsURI:version

{artifactsURI:version}

Purpose: Supplies a deployment-specific REST API URI to the artifacts root.

- `version` is used to specify the version of the REST API to be accessed (eg. v2).

 `version` is case-sensitive and should be specified in lowercase.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {artifactsURI:v2}"
```

the parameter is replaced with:

```
cmd /c C:\ai\ai.bat https://yourServerNameOrIP/api/v2/artifacts
```

username

{username}

Purpose: Supplies the username that is used by the AI node to connect with the LIMS using the REST API.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {username}"
```

the parameter is replaced with:

```
cmd /c C:\ai\ai.bat shorton
```

password

{password}

Purpose: Supplies the password that is used by the AI node to connect with the LIMS using the REST API. In log files, the password supplied on the command line is replaced with a series of *** characters.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {password}"
```

the parameter is replaced with:

```
cmd /c C:\ai\ai.bat 3BlindMice
```

Security:

Users with command line access to the AI node can expose the full password by running the following ps command:

```
ps aux
```

One way to increase security and restrict access to computers running as AI nodes:

- Rather than using the {password} parameter on the command line of the script, pass this parameter inside a credentials file understood by your script.

If you use this method, the password will be located on the AI node inside a temporary parameter file. Note, however, that users with sufficient access and knowledge would be able to open this file and gain access to the password.

Automation/EPP Parameters

When configuring processes, any string can be passed to the operating system command line. The string can then be used as input to a third-party program, or to a script.

These command-line input parameters can provide context to a task that was completed in the lab, or to information recorded by the lab scientist.

Several predefined parameters are available to provide these details, and can be easily added to the processes that trigger the EPP scripts.

Accessing the Predefined EPP Parameters

1. In the Operations Interface, select Configuration, and then select **Processes > Process Types**.
2. In the Process Types drop-down list, select a process type and select **Modify**.
3. In the Modify Process Parameter dialog, select the External Programs tab. The process parameters display.
4. Select a process and select **Modify**.
5. In the Modify Process Parameter dialog, right-click inside the Command Line Call field to access the predefined parameters for the selected process.

i | Notice that parameter names are enclosed in brackets. When EPP triggers and generates a command-line string, the parameter names are replaced with values specific to the associated process run.

Examples for each parameter are provided in the following articles:

- [Process Output File Parameters](#)
- [Process Input File Parameters](#)
- [Process Property Parameters](#)

Case Sensitivity and Parameters that Return Lists

- Pay careful attention when using these parameters, as they are all case sensitive. For example, while {inputFileLuid0} will work, {InputFileLuid0} and {inputfileluid0} will not.
- Some parameters return multiple objects. The script receiving multiple objects often depends on the order in which these values are returned. Values are returned in the order in which they appear in the genealogy, and this order is fixed. If you test your script to understand the order, you can be sure that the objects will always be returned in that order.

Parameter Categories

For information on specific parameters, refer to the [API Portal](#).

Browse the tree at the left of the page. Parameters are categorized based on the information they provide.

Process Output File Parameters

EPP/automation scripts often create files and attach those files as outputs to the process (step) that triggered them. Such files can be attached using the output file placeholders.

File attachment options are configured in the LIMS web interface as part of the Record Details configuration.

Output File Placeholders in the LIMS Web Interface

The following image of the LIMS web interface shows the file placeholders associated with the NanoDrop QC step. Here, the user is required to manually upload a file to the NanoDrop NDT Result File placeholder; the remaining files are generated by the script.

Files	
NanoDrop NDT Result File (required)	Upload File
NanoDrop NDT Parsing Log File	Script Generated
QC Assignment Log File	Script Generated
QC Assignment Report	Script Generated

Output File Placeholders in the LIMS Operations Interface

The image below shows the Operations Interface. Here, the output file placeholders associated with a process—before the actual files are attached to them.

Outputs		
QC	Name	LIMS ID
	 [92-527] Shared RF (1 of 2) [1of2]	92-527
	 [92-528] Shared RF (2 of 2) [2of2]	92-528
	 [100-541] Shared SRF (1 of 2) [1of2]	100-541
	 [100-544] Shared SRF (2 of 2) [2of2]	100-544
	 [BAR103A1CO248] RF Output (1 of 2) for Input 1 [1of4]	BAR103A1CO248
	 [BAR103A1CO249] RF Output (2 of 2) for Input 1 [2of4]	BAR103A1CO249
	 [BAR103A1CO250] SRF Output (1 of 2) for Input 1 [1of4]	BAR103A1CO250

Basic Concepts:

- EPP automatically attaches some output files.
- Output files are either individual or shared.

Automatic Output File Attachment

One of the powerful features of the EPP/automation mechanism is automatic file attachment.

If a script creates a file in the working directory, the EPP/automation can upload it and attach it to an output file placeholder in the LIMS. To use this feature, the script or external program must use a specific file-naming pattern.

How it Works

1. When a script exits, the AI node automatically retrieves any files in the temporary working directory.
2. It then uploads these files to the LIMS and associates them to the correct file output placeholders.

- The output file name must begin with the output file LIMS ID, but can contain other information.
 - For best practice, you should use an underscore or a space between the output file LIMS ID and the subsequent name and extension information.
- For example, if your parameter resolved to LIMS ID 92-527 your file name could be "92-527 Analysis Summary Data.XLS" or "92-527_Analysis Summary Data.XLS".

Individual and Shared Output Files

Output files are either associated with individual inputs or with shared inputs.

The difference between these two types of output files is important, because different parameters are used depending on their type.

***Individual Output File**

Apply to individual Inputs

Parameter Names contain OutputFile

****Shared Output File**

Apply to all Inputs

Parameter Names contain CompoundOutputFile

** Individual output files are also referred to as non-shared output files.*

*** Shared output files are also referred to as compound output files.*

Configuring File Attachment Options in the LIMS Web Interface

File attachment options are configured in the LIMS web interface, as part of Record Details configuration. You can specify the following:

- Whether a file is to be generated by script or manually attached by the user.
- Whether file outputs are shared by all sample inputs or are generated for individual samples.

Parameters for Process Output Files

All of the parameters described below supply output file placeholders – each with unique LIMS IDs – to scripts.

External scripts use the LIMS IDs to name files, such that the files are retrieved from the temporary working directory and associated to output file placeholders when the script exits.

- [outputFileLuids](#)
- [outputFileLuidN](#)
- [compoundOutputFileLuids](#)
- [compoundOutputFileLuidN](#)

outputFileLuids

{outputFileLuids}

Purpose: Supplies the LIMS IDs for all process output file placeholders that apply to individual inputs.

- Configured in the Operations Interface, under **Process Outputs to apply to individual Inputs**.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {outputFileLuids}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat "BAR103A1CO248" "BAR103A1CO249" "BAR103A1CO250" "BAR103A1CO251"
"BAR103A3CO158" "BAR103A3CO159" "BAR103A3CO160" "BAR103A3CO161"
```

outputFileLuidN

{outputFileLuidN}

Purpose: Supplies the LIMS ID for the specified process output file placeholder that applies to an individual input.

- Configured in the **Operations Interface** , under **Process Outputs to apply to individual Inputs** .
- All output file placeholders applying to individual inputs for this process are treated as an array based list, and **N** specifies the array list index position [0..n] of the file desired.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {outputFileLuid0}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat BAR103A1CO248
```

Similarly, in the configured parameter string:

```
cmd /c "C:\ai\ai.bat {outputFileLuid1}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat BAR103A1CO249
```

Finally, in the configured parameter string:

```
cmd /c "C:\ai\ai.bat {outputFileLuid7}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat BAR103A3CO161
```

compoundOutputFileLuids

{compoundOutputFileLuids}

Purpose: Supplies the LIMS IDs for all shared process output file placeholders that apply to 1..n inputs.

- Configured in the Operations Interface, under **Shared Outputs that apply to all Inputs** and **Process Outputs that apply to Inputs per Reagent Label**.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {compoundOutputFileLuids}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat "92-527" "92-528" "100-541" "100-544"
```

compoundOutputFileLuidN

{compoundOutputFileLuidN}

Purpose: Supplies the LIMS ID for the specified process output file placeholder that applies to an individual input.

- Configured in the **Operations Interface**, under **Shared Outputs that apply to all Inputs** and **Process Outputs that apply to Inputs per Reagent Label**.
- All output file placeholders applying to individual inputs for this process are treated as an array based list, and *N specifies the array list index position [0..n] of the file desired.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {compoundOutputFileLuid0}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat 92-527
```

Similarly, in the configured parameter string:

```
cmd /c "C:\ai\ai.bat {compoundOutputFileLuid1}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat 92-528
```

Finally, in the configured parameter string:

```
cmd /c "C:\ai\ai.bat {compoundOutputFileLuid3}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat 100-544
```

Process Input File Parameters



- This section relates to processes run from the LIMS Operations Interface.
- Steps run from the LIMS web interface only support analyte (derived sample) inputs.

The inputs to a process may sometimes be files that were created from a previous process. For the EPP/automation script, such process input files supply additional information, or raw data, for processing. Input files display in the process summary tab in the Operations Interface:

This section describes parameters for input files. In the following list, parameters are categorized according to their purpose in the EPP/automation script.

- *Parameters That Copy Files*
 - *inputFiles*
 - *inputFileN*
- *Parameters That Identify Files*
 - *inputFileLuids*
 - *inputFileLuidN*

- *Parameters That Print Path and File Names*

- *originalInputFileNames*
- *originalInputFileNameN*

Parameters That Copy Files

Copy File parameters are only available for use when both of the following conditions exist:

- The Run each time a file attachment event occurs option is deselected (Operations Interface, in the Add/Modify Process Parameter dialog).
- The EPP/automation is triggered by a process completion event.

The following parameters are used to automatically transfer input files from Clarity LIMS to the processing EPP/AI node. Note the following:

- Input files are named with their unique LIMS ID and the original file extension.
- Input files can be accessed directly by your script.
- Any file-related commands used by your script can assume these files are located in its current working directory.

inputFiles

{inputFiles}

Purpose—Transfers all of the input files to the temporary working directory.

Example: In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {inputFiles}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat "BAR103A1MA9.txt" "BAR103A3MA9.txt"
```

inputFileN

{inputFileN}

Purpose—Transfers the specified input file to the temporary working directory.

- All input files for this process are treated as an array based list.
- **N** specifies the array list index position [0..n] of the file desired.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {inputFile0}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat BAR103A1MA9.txt
```

Similarly, in the configured parameter string:

```
cmd /c "C:\ai\ai.bat {inputFile1}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat BAR103A3MA9.txt
```

Parameters That Identify Files

The following parameters are used to supply the external script with the unique LIMS IDs of the input files, without transferring the input files to the temporary working directory.

This mechanism allows the input files to be identified by the external script without actually transferring those files over the network.

This is particularly useful when the files are very large and stored in a location already known and accessible by the external script.

inputFileLuids

{inputFileLuids}

Purpose—Supplies the LIMS IDs for all input files.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {inputFileLuids}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat "BAR103A1MA9" "BAR103A3MA9"
```

inputFileLuidN

{inputFileLuidN}

Purpose—Supplies the LIMS ID for the specified input file.

- All input files for this process are treated as an array based list.
- **N** specifies the array list index position [0..n] of the file desired.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {inputFileLuid0}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat BAR103A1MA9
```

Similarly, in the configured parameter string:

```
cmd /c "C:\ai\ai.bat {inputFileLuid1}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat BAR103A3MA9
```

 Within the LIMS, generated IDs are most often referred to as 'LIMS IDs', but may occasionally be referred to as LUIDs (Lims Unique IDs).

Parameters That Print Path and File Names

The following parameters are used to supply the external script with the original path and file name of the input files, without transferring the input files to the temporary working directory.

This mechanism allows the input files to be identified by the external script without actually transferring those files over the network. This is particularly useful when the files are very large and stored in a location already known and accessible by the external script.

If you would like external scripts to access files directly using the supplied path and name:

- Be sure that your standard operating procedures for attaching files through the Operations Interface require attaching the files from universal network paths – rather than from a locally referenced directory (i.e., a network path that any machine on your network could access to locate the file).

originalInputFileNames

{originalInputFileNames}

Purpose—Supplies the original path and file names for all input files.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {originalInputFileNames}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat "\\nas\testdata\if-1.txt" "\\nas\testdata\if-3.txt"
```

originalInputFileNameN

{originalInputFileNameN}

Purpose—Supplies the original path and file name for the specified input file.

- All input files for this process are treated as an array based list.
- **N** specifies the array list index position [0..n] of the file desired.

Example:

In the configured parameter string:

```
cmd /c "C:\ai\ai.bat {originalInputFileName0}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat \\nas\testdata\if-1.txt
```

Similarly, in the configured parameter string:

```
cmd /c "C:\ai\ai.bat {originalInputFileName1}"
```

The parameter is replaced with:

```
cmd /c C:\ai\ai.bat \\nas\testdata\if-3.txt
```

 Check that the EPP/AI node machine can resolve the base network path as it is returned from the API.

Automation Testing

Use the `setExitStatus.py` Python script, attached to this page, to test and simulate the use of the automation triggers within BaseSpace Clarity LIMS.



The `setExitStatus.py` script is designed to illustrate concepts for API training purposes. Do not use in a production environment.

The `setExitStatus.py` script relies on the presence of the `glsapiutilv2.py` script. Typically, both scripts are located in the same directory.

The `setExitStatus.py` script uses the following command-line parameters:

-u {user}	LIMS username
- {password}	LIMS password
-l {stepURI}	LIMS stepURI—the URI of the transient step API resource that invokes the script.
-s {status}	The status the script is reporting (OK, WARNING, or ERROR).
-m {message}	The descriptive message displayed to the user.

An example of a parameter string that invokes this script from Clarity LIMS is provided in the following. Note the use of the stepURI token in the -l parameter.

```
python /opt/gls/clarity/customextensions/setExitStatus.py -l {stepURI:v2:http} \  
-u {username} -p {password} -s "OK" -m "successful"
```

Attachments

[glsapiutilv2.py.txt](#)

[setExitStatus.py.txt](#)

The latest `glsapiutil` (and `glsapiutil3`) Python libraries can be found on the [GitHub](#) page.

Troubleshooting Automation

Compatibility

BaseSpace Clarity LIMS v4 and later

Automation is powerful and simple in design. However, its applications can quickly become complex. We recommend you keep your scripts simple. When troubleshooting, the best practice is to isolate the issue to determine the source. The Automated Informatics (AI)/automation worker log file, `automatedinformatics.log`, is useful for isolating system components and diagnosing problems.

Isolate System Components

Isolate the behavior of each component in the system. In particular, determine if the following components can be ruled out as causing of the problem:

- **The script or program**—Running the custom logic, REST calls, and file handling.
- **The AI nodes/automation workers**—Calling the command line and invoking the script or program.
- **The network**—Providing reliable and timely TCP/IP packet transfers.
- **The client**—Completing the process/step and notifying the server.
- **The server**—Responding to client notifications and dispatching to AI nodes/automation workers.

Start with the Script

The script provides many options for troubleshooting. For example, increase logging to rule out unexpected behavior.

- Printing to `stderr` in the script writes a line to the [Error Handling](#) `automatedinformatics.log` file. This file is a great source of information.
- The records in the log file allow for emulating the command-line call for unit testing the script, and calling it manually on the command-line prompt (see [Review Automated Informatics Log File](#)).
- [Validate AI Node/Automation Worker](#) helps verify the automation program on the AI node/automation worker. If the script and the AI node/automation worker are functioning, review the log file entries for any warning (WARN) or error (ERR) lines near the time-stamp of the process completion event sent from the client.
- If the issue is related to the client or server software, contact the Illumina Support team, providing:
 - The `automatedinformatics.log` file
 - The server log
 - The results of the isolation tests (in the previous section).

Validate AI Node/Automation Worker

Use the following steps to test and verify the setup.

Test and Verify Setup on a Windows AI Node/Automation Worker

1. Create a process/step that generates a result file.
2. Configure an automation on the process/step. Associate it with the channel on which the AI node/automation worker is configured to communicate.

3. Add the following command line string.

```
cmd /c "C:\ai\ai.bat {outputFile0}"
```

4. On the AI node/automation worker machine, create an `ai` folder in `C:\` so the system has an `C:\ai` path. Create a new file named `ai.bat`.

5. Edit the `ai.bat` file and add the following line:

```
echo Data for Output File LIMS ID %1 > %1.txt
```

6. Run the process/step created on an existing attached result file.

- The step passes the LIMS ID of its output file placeholder to the script.
- The script creates a file in the working directory.
- When the script exits, this file transfers to the LIMS and associated with the step. The file contains a single line of text that includes the LIMS ID of the output file for easy verification.

Test and Verify Setup on a Linux AI Node/Automation Worker

1. Create a process/step that generates a result file.

2. Configure an automation on the process/step. Associate it with the channel on which the AI node/automation worker is configured to communicate.

3. Add the following command line string:

```
bash -c "echo Automation Test > {outputFile0}.txt"
```

4. Run the process/step created on an existing attached result file.

- The step passes the LIMS ID of its output file placeholder to the script.
- The script creates a file in the working directory with a file name that contains this LIMS ID.
- When the script exits, this file transfers to the LIMS and associated with the step. The file contains the text "Automation Test". When open, the file opens in the default program associated with *.txt files.

With automation, the command-line information is important. The actual values sent on the command line are recorded in the AI log file as the AI node/automation worker receives them. Copy the parameters/tokens from the log and use them on the command line to troubleshoot. If scripting in Groovy, the `cli` class handles command-line tokens well. See the example *.groovy files used with automation and the utility class section in [Work with EPP/Automation and Files](#).

Review Automated Informatics Log File

AI nodes/automation workers are installed using the Automated Informatics (Automation Worker for LIMS v5 and later) software package.

In the installation directory:

1. Find the `/log` directory, which contains an `automatedinformatics.log` file.
2. Use this file to locate log lines near the time of the process/step completion event.

- a. Locate the log line containing the parameter/token command string, and manually run and test the script.
- b. Locate the working directory to review temporary files created.

Step 1: Locate log lines near time of the process completion event

This step can indicate if the cause of the error lies with a network or other issue external to the computer running the AI node / automation worker.

- If no records are found, use the [Validate AI Node/Automation Worker](#) procedure to confirm that logging is functional.

Step 2: Locate log line containing parameter/token command string to manually run/test script

Running the script manually forms a unit test. The script is run on the command line, without being invoked by automation.

To locate the line containing the command-line string, search for the Command string, or externalprogram.runExternalProgram.

An example is shown in the following abridged log file section. Copy the line to a text file and modify it for script testing.

```
2014-02-28 21:24:48,688 INFO ...
definitions.behaviour.automatedinformatics.plugins.externalprogram.runExternalProham as ...
2014-02-28 21:24:49,323 INFO ... (ExternalProgramBehaviour.java:133)... Command string: bash
-c "~/scripts/HelloWorld.sh http://###.###.###.###/api/v2/processes/A30-MXX-110228-24-2320 >
92-2869.txt"
```

Step 3: Locate working directory to review temporary files created

Used when temporary files are left by the script, this automation script removes the temporary working directory, unless there was an error. In case of an error, the directory provides clues to the root cause of the error.

To locate the working directory, search for "Working directory:" in the log file.

An example is shown in the following abridged log file section. Use the recorded directory to list and review temporary files.

```
2014-02-28 21:24:49,324 INFO ... Working directory:
/home/gls/GenoLogicsAutomatedInformatics/temp/runExternalProgram-28022014-
432350866632555775.A30-MXX-110228-24-2320
2014-02-28 21:24:49,324 INFO ... Retrieved files. Executing command.
```



As of BaseSpace Clarity LIMS v5.0, several terms have been deprecated:

- Automation replaces External Program Integration Plug-in (EPP). In LIMS v4.x and earlier, the Operations Interface still uses the term EPP.
- Automation worker/AW node replaces AI node.
- Token replaces Parameter.
- Step replaces Process in the web interface.

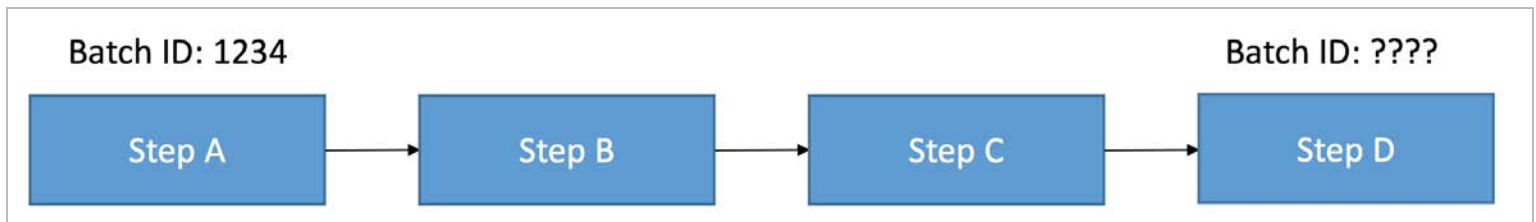
Tips and Tricks

This section provides tips and tricks to help you work efficiently with the API. For example, learn how to copy and update field values, create and rename samples, work with files and QC flags, and automate BCL conversion.

Accessing Step UDFs From a Different Step

This section outlines several strategies to enable this feature.

In all cases, assume that a UDF called Batch ID that was on Step A, and you want to access it on Step D:



If the samples in Step D do not have a homogeneous lineage, expect multiple values for the Batch ID.

Scenario 1: Crawl Back

This method involves crawling backwards from Step D to Step A.

The general form is as follows.

1. Examine the inputs to Step D.
Each input (I) has a parent-process element with a URI to the step that created the artifact. In this case, it is the URI to Step C.
2. Get the input-output maps for Step C (from the /details resource) and find the input (I') that produced output I.
Each input (I') has a parent-process element with a URI to the step that created the artifact. In this case, it is the URI to Step B.
3. Get the input-output maps for Step B (from the /details resource) and find the input (I'') that produced the output I'.
Each input (I'') has a parent-process element with a URI to the step that created the artifact. In this case, it is the URI to Step A.
4. Get the value of the UDF (Batch ID) from Step A: 1234.

This method is computationally slow, but it is safe. As the number of steps that need to be crawled back through increases, so does the duration of the script to retrieve the value.

Scenario 2: Jump Back

This method tried to jump straight to Step A, without passing through Steps B and C.

The general form is as follows.

1. Examine the inputs to Step D.

Each input (I) has a sample element that contains the limsid (S) of the related submitted sample.

2. Query the artifacts API end using the samplelimsid and process-type parameters:

`https://<your_hostname>/api/v2/artifacts?samplelimsid=Sandprocess-type=Step%20A`

This query should give an XML response containing the URI to Step A. From there, get the value of the UDF (Batch ID): 1234.

This method makes two assumptions:

1. That Step A produces analytes (derived samples). Thus, if Step A is a QC process, or does not produce analyte outputs, this method fails.
2. That the analytes (derived samples) resulting from S only passed through Step A one time. If this assumption is not true, you receive multiple URIs to the individual instances of Step A that relate. Also, you cannot be certain which Batch ID to rely upon.

This method is computationally fast, and its duration is not reduced if there are many steps between Step A and Step D.

Scenario 3: Pay it Forward

This method works well, but it involves making configuration changes to the steps. As such, this method is useless for legacy data resulting from samples that passed through the steps before the configuration was applied.

Its general form involves:

- In Step A: Add a script that copies the value of the Batch ID UDF (1234) to every input and output of type analyte in the step.
- In Step B: Add a script that copies the value of the Batch ID UDF (1234) to every output of type analyte in the step.
- In Step C: Add a script that copies the value of the Batch ID UDF (1234) to every output of type analyte in the step.
- In Step D: The inputs contain the value of the Batch ID.

This method relies on propagating the Step UDF through Steps A, B, and C to Step D. It is safe and fast. However, if the protocol is edited and a new step is inserted between B and C, add the script that propagates the value. This addition is so the chain does not break. This method is safe if any of the steps are QC steps or do not produce analyte outputs.

Scenario 4: Along for the Ride

This method is a niche solution, but it works well. It assumes that the samples from Step A proceed to Step D as an intact group, and they are joined by a control sample.

This method involves making configuration changes to the steps. As such, this method is useless for legacy data resulting from samples that passed through the steps before the configuration was applied.

- In Step A: Identify the control sample for the group, then copy the value of the Batch ID to the control sample.
- In Step D: Identify the control sample for the group, then retrieve the value of the Batch ID from it.

This method is the least work, but it does make several assumptions that might make it impracticable.

Obfuscating Sensitive Data in Scripts

If a BaseSpace Clarity LIMS script is run in an automation context, it is easy to obfuscate usernames and passwords by choosing the appropriate tokens (`{username}` or `{password}`) to be passed in as run-time arguments.

However, this type of functionality is not easily available outside of automations, and it is often necessary to store various credentials on machines that need to interact with the LIMS API, database, or some other protected resource. This article explains how to use cryptography in Python to protect and obfuscate these important authentication tokens.

Background

Many of the API Cookbook examples use a simple `auth_tokens.py` file that has usernames and passwords stored in plain text. This file can be compiled in Python, simply by importing it at a Python console:

- Import `auth_tokens`.
- Print `auth_tokens.username` to verify.

Importing this file creates an `auth_tokens.pyc` file—a byte-compiled version of the source file. The source file can now be deleted, providing the first rudimentary level of security. However, the credentials can still quite easily be retrieved. Even if the permissions on this file are restricted, this solution does not present a suitable level of security for most IT administrators. It does, however, allow us to easily prototype our code, hence its use in Cookbook examples.

Assumptions

- You have `pycrypto` installed (either through the OS package manager or `pip`).
- You have generated a secret key of random ASCII characters (create this key by randomly pressing keys on a US-layout keyboard and include numerous symbols).
- You already have a plain-text `auth_tokens.py` file. An example is attached at the bottom of this article.
- You have access to the Python or iPython command-line console.

Cryptography in Python

Python provides the `pycrypto` library that can easily be installed using the operating system package manager or the `pip` installation tool. It contains myriad different encryption algorithms and gives us a straightforward interface to wrap our own encryption objects and accessor functions.

Towards a Better `auth_tokens.py`

The goal is to be able to create a flat text file containing obfuscated usernames, passwords, hostnames, and so on. To do this, use a utility class called `ClarityCred` that provides encryption and decryption functionality using the ARC4 cipher from `pycrypto`. The `ClarityCred` class is provided in `cred.py`, attached at the bottom of this article.

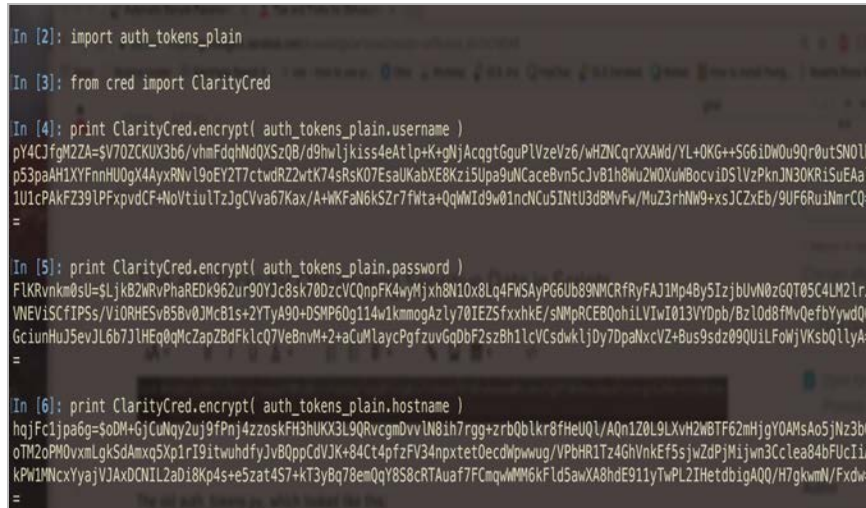
While the use of ARC4 is considered deprecated in favor of stronger encryption algorithms, such as AES, the ARC4 example lends itself to easier understanding. ARC4 simply requires a secret key and a salt size to be specified. The secret key can be generated at random using any preferred method and is hard-coded in `cred.py`, along with the salt size purely for ease of demonstration. Ideally, the secret key and salt size should be stored externally.

After applying the ARC4 encryption, the `ClarityCred` class wraps base64 encoding around it to obfuscate the data further.

Assume that you need to store a username, password, and hostname inside our `auth_tokens.py`, and we have this information in plain-text stored in another file called `auth_tokens_plain.py`. The usage is as follows.

1. Open a Python console, and import `ClarityCred` from `cred.py`.
2. Call the `ClarityCred.encrypt()` static function on the plain text username, password, and hostname strings.
3. Copy and paste these values into `auth_tokens.py`.

The following image illustrates steps 1 and 2, using an existing `auth_tokens_plain.py` file:



```
In [2]: import auth_tokens_plain

In [3]: from cred import ClarityCred

In [4]: print ClarityCred.encrypt( auth_tokens_plain.username )
pY4CjfgM2ZA=$70ZCKUX3b6/vhmFdqNdQXSz0B/d9hwLjkiss4eAtlp+K+gNjAcqgtGguPLVzeVz6/wHZNCqrXXAWd/YL+OKG++SG6iDw0u9Qr0utSN01h
p53paAH1XYFnnHU0gX4AyxRNVl9oEY2T7ctwdRZ2wtK74sRsK07EsaUKabXE8Kzi5Upa9uNCaceBvnScJvB1h8Wu2W0XuNBocv1DS1VzPknJN30KRiSuEAaL
1U1cPAKFZ39lPFxpvdCF+NoVtiulTzJgCvva67Kax/A+WKfaN6kSZr7fWta+QqWmId9w01ncNCu5INTu3dBMvFw/MuZ3rhNw9+xsJCZxEb/9UF6Ru1NmrCQ=
=

In [5]: print ClarityCred.encrypt( auth_tokens_plain.password )
F1KRvnmk0sU=$ljK8ZWRvPhaREDk96Zur90YJc8sk70DzcVCQnpFK4wyMjxh8N10x8Lq4FWSAyPG6Ub89NMCRfRyFAJ1Mp4By5IzjbUvN0zGQT05C4LM2lrJ
VNEViSCfIPSS/ViORHESv85Bv0JMc81s+2YTyA90+DSMP60g114w1kmmogAzLy70IEZSfxhkE/slWpRCEBQohiLViW1013VYDpb/Bz10d8fmvQefbYywdQ6
GciunHuJ5evJL6b7JlHEq0qMcZapZ8dFk1cQ7VeBnvM+2+aCuMlaycPgfvzuvGqDbF2szBh1lcVCsdwkljDy7DpaNxcVZ+Bus9sdz09QUiLFowjVKsbQllyA=
=

In [6]: print ClarityCred.encrypt( auth_tokens_plain.hostname )
hqJfC1jpa6g=$oDM+GjCuNqy2uj9fPnj4zzoskFH3hUKX3L9QrvcmDvvUN81h7rgg+zrbQblkr8fHeUQl/AQn1Z0L9LXvH2WBTf62mHjgY0AMsAo5JNz3b0
oTM2oPM0vxmlGkSdAmxQXp1r19itwuhdfyJvBQppCdVJK+84Ct4pfzFV34npxtet0ecdWpwwug/VPbHR1Tz4GhVnkEf5sjwZdPjMijwn3Cclea84bFucIiA
kPW1MncxYyajVJAxDcN1L2a0i8Kp4s+5e2at457+kT3yBq78emQqY8S8cRTAuaf7FCmqwMM6kFLd5awXA8hdE911yTwPL2IHetdbigAQQ/H7gkwmN/Fxdw=
=
```

The old `auth_tokens_plain.py` looked like this:

```
username = 'testuser'
password = 'testpass'
hostname = 'https://encryptiontest.claritylims.com'
```

The new `auth_tokens.py` looks like this:

```
username =
'zq1AwnqIkfA=$YFY1Uu01r6edu7qPnN9/l3kMI15ZG1JAsH7IhnxnNvYulMndhYh61xjVBfFwjN9sZEqPM0Q1x6kjq
3fbht/F1Rrgk1DL79H7NiUP6uYM2qVltPloRA4g8SiphF3KHx4gVTE93Ku58sFCgulrnh5u6tkCz98v0R7PsuIOW1CD
Mi9zSToIu+IkcyDPPYcD1b4z8ojez/7lczunaDfrmPhwopyyUiETu9BR49Bwp5fz4XSWICZFGCd9AjoEg/FTE+/X18f
+0pIz0viXQyN+JjE3vJkpNsRY2Z3d72sPgQmFFZhd48m+POUtD1UXLXhaijdxp78QTcEp7AHY+TiM8hsXT7BX1Q=='

password =
'9qW5BftGyXY=$6GL1t/Z11CbSmB7Qq54uf2TJ5fI8GU1W9NdBnumkTtF/X27WLEsr1+C0ilXQX6jnLm4kzR+5pCVgn
z4xz6/80/dMLM1T116tOvCJgPU4ZkRpkUYmcPVbrp+X3azR7I02408UjV/JeJYV869h3kvdPyWJGXRH4oJgs5NTJKI2
y6URBs0wlr1rgBuZ2YkO855ZGPw9J07UMM606q9xERRzQ+LT1XLRzSCuFnuSoDVEhshhYqZ/jpYWDHvA6Z5+YTYI/i09
9iYZ+WQdJAiU9hcgkUnWCybjcwivvHG6vAIROroLq1Oefo+hrJsVFBA3uDaPS8pkgMVskMPUGeft6vx4NgN/jaw=='

hostname =
'Q+oyq2m9Nv8=$rhgeJOMdm/M+dDN1SbBA3RCsUoo0Ts65G7lePvuajRmsLSNC5Qo5bwagRuyat0ztpeZrUmD8xTxTv
hUBvZYDlM6GBLsq5drBP6PFh/lplxb608YiSRXrboFov8tRnu6GbaTfGR8WV7s8vBZsXhrhlPn67p7yalJLnHWb9VOK
hx8AgCTtytQkkEwmpm2vbDwDha9kMdK63IrOSp2jmRaI/9X3xsd4upqaxvX7zrEJ8ruGU/szN0ITxTK1rprnowpyXfB
RiOEcrI7uh1bg73oqOETn3pB/uTrGkhGETKYB2aHaewwWmccbeZTgEPT0kDmuJdpoGYy+p+gxSoR9Arh3JtREIA=='
```

Examples of the plain-text `auth_tokens_plain.py` and encrypted `auth_tokens.py` are attached at the bottom of this article.

Using the New `auth_tokens.py` in Your Scripts

Now that the new `auth_tokens.py` is ready to use, you can import it and create the corresponding PYC file to provide that extra level of security, as previously discussed. You can remove the PY file and ship the PYC file everywhere it is required.

It may also be a good idea to restrict the read/write/execute permissions on the file to the system user that is calling the file (usually `glsai` in Clarity LIMS installations).

To use the values in this file in the code, we need to use the `decrypt()` function in `ClarityCred`. Look at the simple example of initializing a `glsapiutil` api object. For reference, the example current directory listing looks like this:

```
total 72
-rw-r--r--  1 vramraj  32840   871B 22 Nov 02:31 auth_tokens.pyc
-rw-r--r--  1 vramraj  32840   3.8K 10 Dec 11:28 cred.pyc
lrwxr-xr-x  1 vramraj  32840    27B 10 Dec 11:36 glsapiutil.py@ -> ../glsapiutil/glsapiutil.py
-rw-r--r--  1 vramraj  32840   21K 10 Dec 11:36 glsapiutil.pyc
```

Notice the `.py` source files are removed wherever possible.

Using a Python console, the normal api invocation (using a plain-text `auth_tokens` file) would look as follows.

```
import glsapiutil
import auth_tokens_plain

api = glsapiutil.glsapiutil2()
```

```
api.setHostname( auth_tokens_plain.hostname )
api.setVersion( 'v2' )
api.setup( auth_tokens_plain.username, auth_tokens_plain.password )
```

Now, however, with the encrypted tokens, we decrypt the values on the fly (changes shown in italicized text):

```
import glsapiutil
import auth_tokens
from cred import ClarityCred

api = glsapiutil.glsapiutil2()
api.setHostname( ClarityCred.decrypt( auth_tokens.hostname ) )
api.setVersion( 'v2' )
api.setup( ClarityCred.decrypt( auth_tokens.username ), ClarityCred.decrypt( auth_
tokens.password ) )
```

This method provides a relatively robust solution for encrypting and obfuscating sensitive data and can be used in any Python context, not just for Clarity LIMS API initialization. By further ensuring that only the `auth_tokens.pyc` file is shipped and copied with restricted read/write/execute permissions, this method should help satisfy IT security requirements.

However, the matter of storing the secret key externally remains. One idea is to store the secret key in a separate file and encrypt that file using openssl or an OpenPGP key. While the problem of storing each piece of information in encrypted format likely never fully goes away, the use of multiple methods of encryption can offer better protection and peace of mind.

Attachments

[auth_tokens.py.txt](#)

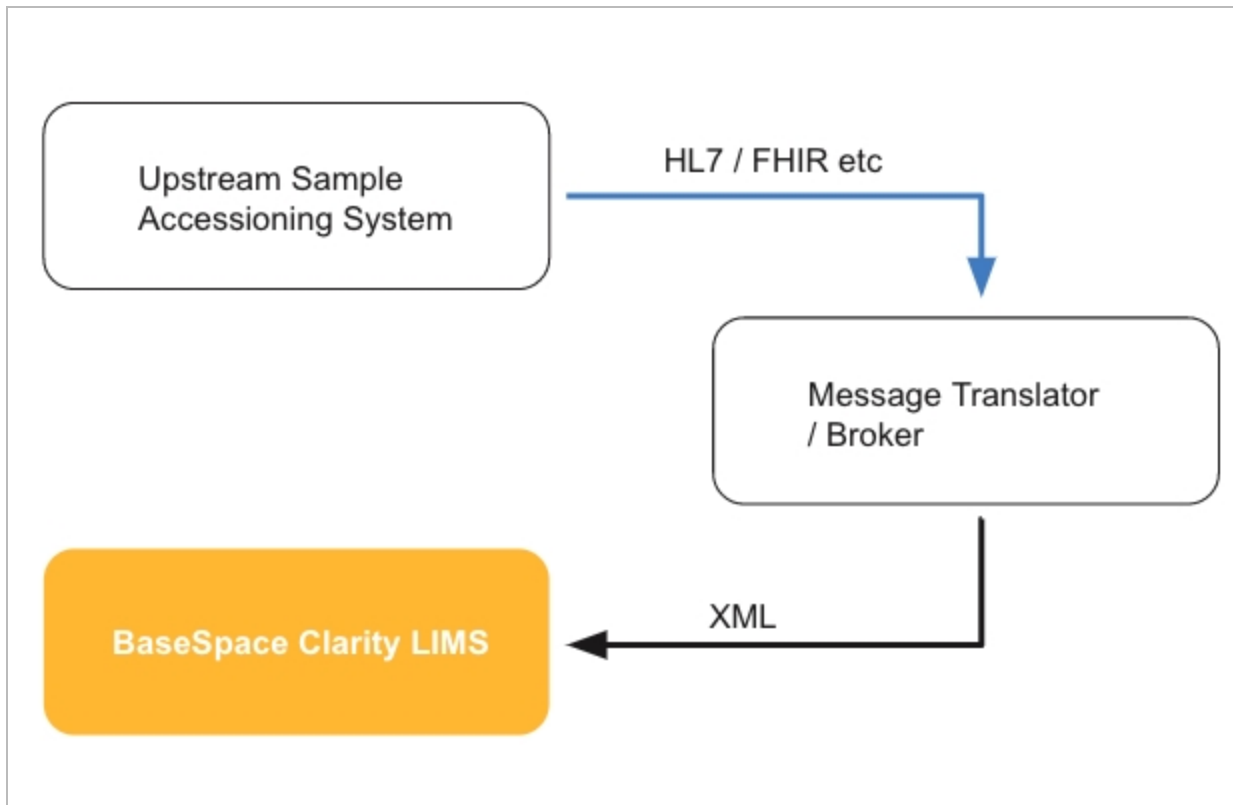
[auth_tokens_plain.py.txt](#)

[cred.py.txt](#)

Integrating BaseSpace Clarity LIMS with Upstream Sample Accessioning Systems

This section discusses methods for integrating BaseSpace Clarity LIMS with upstream sample accessioning systems.

The following illustration shows a typical architectural overview:



Creating a Sample in BaseSpace Clarity LIMS

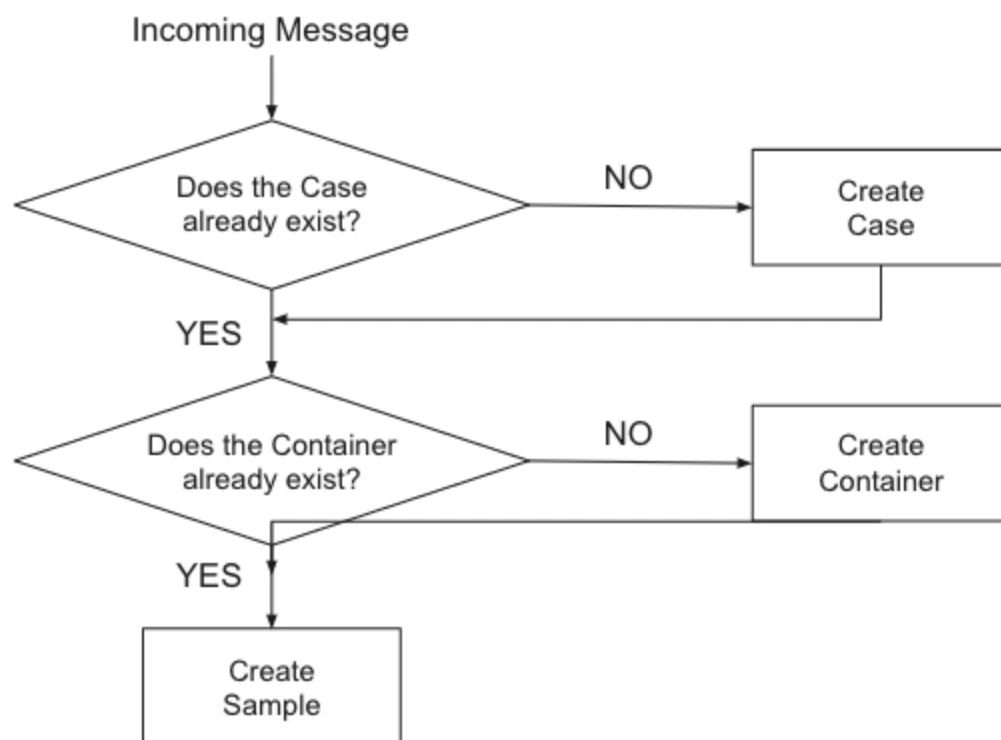
Required:

- A sample must have a Name/ID.
- A sample must be associated with a Case/Patient/Study/Project.
- A sample must be associated with a Container (Tube/Plate, and so on).

Optional (but expected):

User-defined fields (UDFs)/custom fields (defined by your LIMS configuration)

Typical flowchart of actions within the broker:



The following animation illustrates the elements of an XML sample-creation message to Clarity LIMS.

What does a XML Sample-creation message to Clarity look like?

```
<?xml version="1.0" encoding="UTF-8"?>
<smp:samplecreation xmlns:smp="http://genologics.com/ri/sample"
xmlns:udf="http://genologics.com/ri/userdefined">
  <name>NA12878</name>
  <project uri="http://your_server:8080/api/v2/projects/DIB601"></project>
  <location>
    <container uri="http://your_server:8080/api/v2/containers/27-195"></container>
    <value>1:1</value>
  </location>
  <udf:field name="Reference Genome">GRCh37</udf:field>
</smp:samplecreation>
```

Sample Name



1/6

Common Options for the Broker

Build your own:

- Pro: Not too difficult
- Con: Stability as number of messages increases
- ?: Maintainable over the long term

Use a commercial/open-source offering (eg, Mirth Connect)

- Pro: Quicker than build
- Pro: Robust, multithreaded support for millions of messages per day
- ?: May prove to be an excessive or overcomplicated means to accomplish something relatively simple

Does the broker need to carry out other business logic?

For example, one customer added logic to their broker that dealt with medical billing and was able to distinguish between physicians ordering duplicate tests for a subject (not reimbursable, therefore the duplicate sample wasn't submitted to Clarity LIMS), versus a temporal study that was reimbursable.

The best practice is to take advantage of as many legacy systems as possible, rather than creating samples in Clarity LIMS, then reinventing business logic to remove unwanted ones.

Creating Samples and Projects via the API

Assumptions

The incoming message contains the following:

- Project ID or Name
- Sample ID or Name
- Container ID or Name
- Container type (plate/tube type)
- Container well position if sample is on a plate (eg, G:2)
- Sample user-defined fields (UDFs)/custom fields

Main Logic

```
Does the project exist?  
  if NO: create it  
Does the container exist?  
  if NO: create it  
Create sample
```

Create a Sample

POST to https://your_server/api/v2/samples:

```
<?xml version="1.0" encoding="UTF-8"?>  
<smp:samplecreation xmlns:smp="http://genologics.com/ri/sample"  
xmlns:udf="http://genologics.com/ri/userdefined">  
  <name>20140909-1</name>  
  <project uri="https://your_server/api/v2/projects/ROS210"></project>  
  <location>  
    <container uri="https://your_server/api/v2/containers/27-195"></container>  
    <value>1:1</value>  
  </location>  
  <udf:field name="Reference Genome">Cane Toad</udf:field>  
</smp:samplecreation>
```

We receive something like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>  
<smp:sample uri="https://your_server/api/v2/samples/ROS210A14" limsid="ROS210A14">
```

```

<name>20140909-1</name>
<date-received>2014-09-10</date-received>
<project limsid="ROS210" uri="https://your_server/api/v2/projects/ROS210" />
<submitter uri="https://your_server/api/v2/researchers/1">
  <first-name>System</first-name>
  <last-name>Administrator</last-name>
</submitter>
<artifact limsid="ROS210A14PA1" uri="https://your_
server/api/v2/artifacts/ROS210A14PA1?state=4262" />
<udf:field type="String" name="Reference Genome">Cane Toad</udf:field>
</smp:sample>

```

Create a Project

POST to https://your_server/api/v2/projects

```

<?xml version="1.0" encoding="UTF-8"?>
<prj:project xmlns:udf="http://genologics.com/ri/userdefined"
xmlns:ri="http://genologics.com/ri" xmlns:file="http://genologics.com/ri/file"
xmlns:prj="http://genologics.com/ri/project">
  <name>Week 39</name>
  <open-date>2014-09-10</open-date>
  <researcher uri="https://your_server/api/v2/researchers/1"/>
</prj:project>

```

We receive something like the following:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<prj:project uri="https://your_server/api/v2/projects/ADM372" limsid=" ADM372 ">
  <name>Week 39</name>
  <open-date>2014-09-10</open-date>
  <researcher uri="https://your_server/api/v2/researchers/1" />
</prj:project>

```

Create a Container (tube)

POST to https://your_server/api/v2/containers:

```

<?xml version="1.0" encoding="UTF-8"?>
<con:container xmlns:con="http://genologics.com/ri/container">
  <name>Example Container 20140910</name>
  <type uri="https://your_server/api/v2/containertypes/2" name="Tube"/>
</con:container>

```

We receive something like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<con:container uri="https://your_server/api/v2/containers/27-1869" limsid=" 27-1869 ">
  <name>Example Container 20140910</name>
  <type uri="https://your_server/api/v2/containertypes/2" name="Tube" />
  <occupied-wells>0</occupied-wells>
  <state>Empty</state>
</con:container>
```

Create a Container (96 well plate)

POST to https://your_server/api/v2/containers:

```
<?xml version="1.0" encoding="UTF-8"?>
<con:container xmlns:con="http://genologics.com/ri/container">
  <name>Example Plate 20140910</name>
  <type uri="https://your_server/api/v2/containertypes/1" name="96 well plate"/>
</con:container>
```

We receive something like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<con:container uri="https://your_server/api/v2/containers/27-1870" limsid=" 27-1870 ">
  <name>Example Plate 20140910</name>
  <type uri="https://your_server/api/v2/containertypes/1" name="96 well plate" />
  <occupied-wells>0</occupied-wells>
  <state>Empty</state>
</con:container>
```

Create a Sample in the 96 Well Plate Created, and the Project Created

POST to https://your_server/api/v2/samples:

```
<?xml version="1.0" encoding="UTF-8"?>
<smp:samplecreation xmlns:smp="http://genologics.com/ri/sample"
xmlns:udf="http://genologics.com/ri/userdefined">
  <name>20140909-1</name>
  <project uri="https://your_server/api/v2/projects/ ADM372 "></project>
  <location>
    <container uri="https://your_server/api/v2/containers/ 27-1870 "></container>
    <value>G:2</value>
  </location>
  <udf:field name="Reference Genome">Cane Toad</udf:field>
</smp:samplecreation>
```

We receive something like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<smp:sample uri="https://your_server/api/v2/samples/ADM372A2" limsid="ADM372A2">
  <name>20140909-1</name>
  <date-received>2014-09-10</date-received>
  <project limsid="ADM372" uri="https://your_server/api/v2/projects/ ADM372 " />
  <submitter uri="https://your_server/api/v2/researchers/1"> <
    first-name>System</first-name>
    <last-name>Administrator</last-name>
  </submitter>
  <artifact limsid="ADM372A2PA1" uri="https://your_
server/api/v2/artifacts/ADM372A2PA1?state=4264" />
  <udf:field type="String" name="Reference Genome">Cane Toad</udf:field>
</smp:sample>
```

Confirm the Project Exists

GET: https://your_server/api/v2/projects?name=Week%2039

If the project exists, we receive something like the following:

```
<prj:projects xmlns:prj="http://genologics.com/ri/project">
  <project uri="https://your_server/api/v2/projects/ADM372" limsid="ADM372">
    <name>Week 39</name>
  </project>
</prj:projects>
```

If the project does not exist, we receive something like the following:

```
<prj:projects xmlns:prj="http://genologics.com/ri/project"/>
```

Confirm the Container Exists

GET: https://your_server/api/v2/containers?name=Example%20Container%2020140910

If the container exists, we receive something like the following:

```
<con:containers xmlns:con="http://genologics.com/ri/container">
  <container uri="https://your_server/api/v2/containers/27-1869" limsid="27-1869 ">
    <name>Example Container 20140910</name>
  </container>
</con:containers>
```

If the container does not exist, we receive something like the following:

```
<con:containers xmlns:con="http://genologics.com/ri/container"/>
```

Displaying Files From an Earlier Step

This article explains how to make files that were produced by/attached to Clarity LIMS in an earlier step and are visible in a subsequent step.

Solution

Consider a simplified workflow/protocol containing just two steps: Produce Files and Display Files.

- The first step, Produce Files, takes in analytes (derived samples) and generates individual resultfiles (one per input).
- The subsequent Display Files step allows us to view the files associated with the analytes from the previous step.

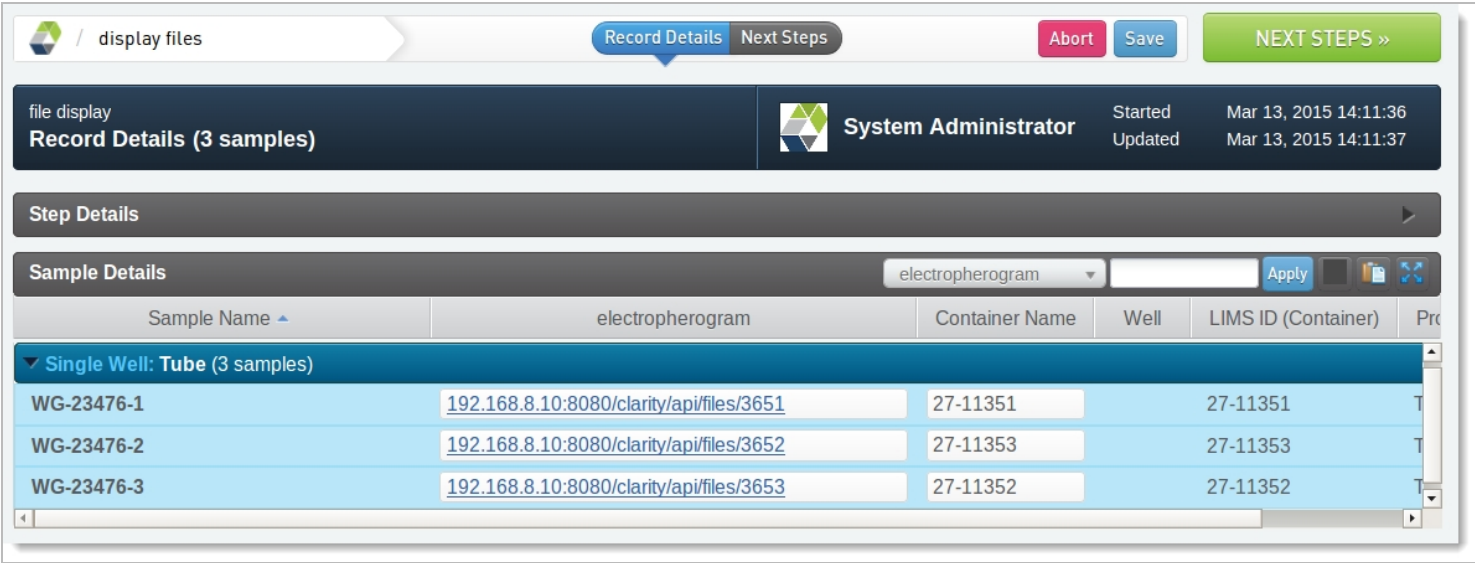
After the files have been generated by and attached to the Produce Files step, the Record Details screen of the step displays the files.

The key to displaying these files in any subsequent step involves producing a hyperlink to the file and displaying it as a user-defined field (UDF)/custom field in subsequent steps.

You may be familiar with creating and using text, numeric, and checkbox UDFs/custom fields. However, you may be less familiar with the hyperlink option. Fields of this type are used less frequently, but they are perfect for this solution.

i | As of BaseSpace Clarity LIMS v5.0, the term user-defined field (UDF) has been replaced with custom field in the user interface. However, the API resource is still called UDF.

This solution involves a script that runs on the Record Details screen on the subsequent Display Files step and populates the fields. See the following figure.



As you can see, the structure of the hyperlink is straightforward and includes:

- The IP address/hostname of the server
- The port
- A link to the LIMS ID of the file to be linked to

Script

To populate these fields, there are numerous methods available within an API-based script. The method discussed here works for the two-step protocol described earlier (namely that we want the files displayed in the next step of the protocol). It also works when the steps in which the files are uploaded and displayed are separated by several intermediate steps.

Assuming that the script runs just as the Record Details screen of the Display Files step is being displayed, use pseudocode to produce the hyperlinks.

For each output:

1. Determine the LIMS Unique ID (LUID) of the output artifact.
2. Determine the LUID of the submitted sample associated with the output artifact.
3. Determine the LUID of the resultfile artifact produced by the earlier process, derived from the common submitted sample.
4. Determine the LUID of the file associated with the resultfile artifact.
5. Update the hyperlink UDF/custom field on the output artifact (from Step 1) with the specific hyperlink value.

To illustrate these pseudocode steps, XML from a demo system is provided.

Step 1: Gather the LIMS Unique ID (LUID) of the Output Artifact

```
<prc:process uri="http://192.168.8.10:8080/api/v2/processes/24-24452" limsid="24-24452">
  <type uri="http://192.168.8.10:8080/api/v2/processtypes/1555">display files</type>
  <technician uri="http://192.168.8.10:8080/api/v2/researchers/1">
    <first-name>System</first-name>
    <last-name>Administrator</last-name>
  </technician>
  <input-output-map>
    <input post-process-
      uri="http://192.168.8.10:8080/api/v2/artifacts/ADM1301A2PA1?state=48636"
      uri="http://192.168.8.10:8080/api/v2/artifacts/ADM1301A2PA1?state=48636"
      limsid="ADM1301A2PA1"/>
    <output uri="http://192.168.8.10:8080/api/v2/artifacts/2-81806?state=49106" output-
      generation-type="PerInput" output-type="Analyte" limsid=" 2-81806 "/>
  </input-output-map>
  <input-output-map>
    <input post-process-
      uri="http://192.168.8.10:8080/api/v2/artifacts/ADM1301A3PA1?state=48632"
      uri="http://192.168.8.10:8080/api/v2/artifacts/ADM1301A3PA1?state=48632"
      limsid="ADM1301A3PA1"/>
```

```

        <output uri="http://192.168.8.10:8080/api/v2/artifacts/2-81805?state=49105" output-
        generation-type="PerInput" output-type="Analyte" limsid=" 2-81805 "/>
    </input-output-map>
    <input-output-map>
        <input post-process-
        uri="http://192.168.8.10:8080/api/v2/artifacts/ADM1301A1PA1?state=48650"
        uri="http://192.168.8.10:8080/api/v2/artifacts/ADM1301A1PA1?state=48650"
        limsid="ADM1301A1PA1"/>
        <output uri="http://192.168.8.10:8080/api/v2/artifacts/2-81804?state=49104" output-
        generation-type="PerInput" output-type="Analyte" limsid=" 2-81804 "/>
    </input-output-map>
</prc:process>

```

From the XML representation of the Display Files process/step. we see that we have three output artifact LUIDS: 2-81806, 2-81805 and 2-81804.

Step 2: Determine the LUID of the Submitted Sample Associated with the Output Artifact

By examining the XML representation of the first output artifact (2-81806), see that the LUID of the associated submitted sample is ADM1301A2:

```

<art:artifact uri="http://192.168.8.10:8080/api/v2/artifacts/2-81806?state=49106"
limsid=" 2-81806 ">
    <name>WG-23476-2</name>
    <type>Analyte</type>
    <output-type>Analyte</output-type>
    <parent-process uri="http://192.168.8.10:8080/api/v2/processes/24-24452" limsid="24-
    24452"/>
    <qc-flag>UNKNOWN</qc-flag>
    <location>
        <container uri="http://192.168.8.10:8080/api/v2/containers/27-11353" limsid="27-
        11353"/>
        <value>1:1</value>
    </location>
    <working-flag>true</working-flag>
    <sample uri="http://192.168.8.10:8080/api/v2/samples/ADM1301A2" limsid=" ADM1301A2 "/>
</art:artifact>

```

Step 3: Determine the LUID of the resultfile Artifact Produced by the Earlier Process/Step, Derived from the Common Submitted Sample

After the common ancestor is found, ask Clarity LIMS for the output artifacts produced by our step of interest (Produce Files) directly.

For example:

```
http://192.168.8.10:8080/api/v2/artifacts?samplelimsid= ADM1301A2 andprocess-type=
Produce %20 Files andtype=ResultFile
```

Yields the following XML:

```
<art:artifacts>
  <artifact limsid=" 92-81803 " uri="http://192.168.8.10:8080/api/v2/artifacts/92-
    81803"/>
</art:artifacts>
```

The resultfile with LUID 92-81803 is associated with the current output artifact (2-81806), even though these entities may be separated by several steps.

If the process/step produces multiple resultfiles, you may need to further constrain the search using the name= parameter. For example:

```
http://192.168.8.10:8080/api/v2/artifacts?samplelimsid= ADM1301A2 andprocess-type=
Produce %20 Files andtype=ResultFileandname=<name of ResultFile>
```

Step 4: Determine the LUID of the file associated with the resultfile artifact

By gathering the XML representation of artifact 92-81803, the associated file has LUID 40-3652:

```
<art:artifact uri="http://192.168.8.10:8080/api/v2/artifacts/92-81803?state=49103"
limsid=" 92-81803 ">
  <name>WG-23476-2</name>
  <type>ResultFile</type>
  <output-type>ResultFile</output-type>
  <parent-process uri="http://192.168.8.10:8080/api/v2/processes/24-24451" limsid="24-
    24451"/>
  <qc-flag>UNKNOWN</qc-flag>
  <sample uri="http://192.168.8.10:8080/api/v2/samples/ADM1301A2" limsid="ADM1301A2"/>
  <file:file limsid=" 40-3652 " uri="http://192.168.8.10:8080/api/v2/files/40-3652"/>
</art:artifact>
```

Step 5: Update the Hyperlink UDF/Custom Field on the Output Artifact (from Step 1) with the Specific Hyperlink Value

Now that you know the LUID of the file associated with output artifact 2-81803, set the value of its hyperlink field in the following form:

```
http://192.168.8.10:8080/clarity/api/files/3652
```

When constructing the value for the hyperlink, the 40- prefix should be removed from the LUID of the file.

Transitioning Output Artifacts into the Next Step

In a highly automated workflow, a lab gains little value from manually selecting samples into the ice bucket and then transitioning them through a step. Ideally, upon completion of one step, a following step could be automated such that the output analytes were transitioned through to the Record Details screen.

The Clarity LIMS External Program Plugin (EPP)/automation system cannot aid in this transition. The last point at which an automation can be triggered is before the step completion.

This scenario requires a stand-alone API application, which can be run by an automation at the end of a step.

Using this approach, a standalone app would poll the API until each of the output analytes from the previous step were queued for the next step. After they are queued, they can be walked through to the Record Details stage.

The steps are as follows:

1. EPP/automation triggers at step completion and launches an API app as a new Linux process and then finishes. The parameter for the API app is the URL for the current process.
2. API app polls to see if each output analyte is queued.
 - Use the artifacts batch endpoint (`/api/v2/artifacts/batch/retrieve`) to poll.
 - Check the last `workflow-stage` node within `workflow-stages` and look for `status="QUEUED"`.
3. API app moves the output analytes through the step to Record Details.
 - Use the `/api/v2/steps` endpoints to start the step and then move the analytes forward.

Determining the Workflow(s) to Which a Sample is Assigned

Within a script, you may sometimes need to know to which workflow the current sample is assigned.

However, in Clarity LIMS, the XML payload that relates to the sample does not provide information about the workflow associations of the sample.

For example, consider a sample (artifact), picked at random, from a demo system:

```
<art:artifact uri="http://192.168.8.10:8080/api/v2/artifacts/2-42201?state=25871" limsid="2-42201">
  <name>Sanger Sample 49</name>
  <type>Analyte</type>
  <output-type>Analyte</output-type> <parent-process
uri="http://192.168.8.10:8080/api/v2/processes/24-13704" limsid="24-13704"/>
  <qc-flag>UNKNOWN</qc-flag>
  <location>
    <container uri="http://192.168.8.10:8080/api/v2/containers/27-6899" limsid="27-6899"/>
    <value>E:1</value>
  </location>
  <working-flag>true</working-flag>
  <sample uri="http://192.168.8.10:8080/api/v2/samples/KUZ407A145" limsid="KUZ407A145"/>
</art:artifact>
```

It is evident that this XML payload does not provide the workflow information.

This following solution shows how to use the Clarity LIMS API to determine the association of a sample to one or more workflows.

Solution

The XML payload that corresponds to each sample artifact contains a link to the related submitted sample (or samples, if it is a pooled artifact).

Follow that link to see what it yields:

```
<smp:sample uri="http://192.168.8.10:8080/api/v2/samples/KUZ407A145" limsid="KUZ407A145">
  <name>Sanger Sample 49</name>
  <date-received>2013-07-15</date-received>
  <project limsid="KUZ407" uri="http://192.168.8.10:8080/api/v2/projects/KUZ407"/>
  <submitter uri="http://192.168.8.10:8080/api/v2/researchers/4">
    <first-name>Jill</first-name>
    <last-name>Hesse</last-name>
  </submitter>
  <artifact limsid="KUZ407A145PA1"
uri="http://192.168.8.10:8080/api/v2/artifacts/KUZ407A145PA1?state=25080"/>
  <udf:field type="String" name="Progress">Initial sample QC complete</udf:field>
  <udf:field type="String" name="Sanger Primer">ITS1</udf:field>
  <udf:field type="String" name="Template (uL)">6.9</udf:field>
</smp:sample>
```

The XML corresponding to the submitted sample has a link to an artifact. This artifact is special for several reasons:

- It is known as the root artifact.
- It has an unusual LIMS ID for an artifact. LIMS IDs start with 2- for analytes, and 92- for resultfiles. This one appears to be derived from the LIMS ID of the sample: KUZ407A145PA1
- A root artifact is created behind the scenes whenever a submitted sample is created in the system.
- The sample history in Clarity LIMS makes it appear as if the first step in the workflow is run on the submitted sample. However, it is actually the root artifact that is the input to the first process.
- When a submitted sample is assigned to the workflow, it is the root artifact that is assigned to that workflow.

Therefore, if gathering the XML payload corresponding to the root artifact, you should see the workflow assignment:

```
<art:artifact uri="http://192.168.8.10:8080/api/v2/artifacts/KUZ407A145PA1?state=25080"
limsid="KUZ407A145PA1">
  <name>Sanger Sample 49</name>
  <type>Analyte</type>
  <output-type>Analyte</output-type>
  <qc-flag>UNKNOWN</qc-flag>
  <location>
```

```

<container uri="http://192.168.8.10:8080/api/v2/containers/27-6754" limsid="27-6754"/>
  <value>1:1</value>
</location>
<working-flag>true</working-flag>
<sample uri="http://192.168.8.10:8080/api/v2/samples/KUZ407A145" limsid="KUZ407A145"/>
  <udf:field type="String" name="Conc. Units">ng/uL</udf:field>
  <udf:field type="Numeric" name="Concentration">100</udf:field>
  <artifact-group name="Sanger Sequencing"
    uri="http://192.168.8.10:8080/api/v2/artifactgroups/851"/>
</art:artifact>

```

The key element is as follows.

```

<artifact-group name="Sanger Sequencing"
  uri="http://192.168.8.10:8080/api/v2/artifactgroups/851"/>

```

The name of the `artifact-group` (Sanger Sequencing) should match the name of the workflow in which the root artifact (and artifacts derived from the root artifact) is assigned.

Troubleshooting

If you find that the `artifact-group` node is missing from some of the root artifacts, there are several potential reasons:

- The workflow has been completed, causing the root artifact to be unassigned from the workflow.
- The derived samples/artifacts have been removed from the workflow intentionally because of a sample processing issue.
- An API script has intentionally removed the derived samples/artifacts from the workflow.
- The assigned workflow has been marked as Archived.

Standardizing Sample Naming via the API

A lab may receive samples submitted from various sources. This can pose a problem with regards to sample names. There may be duplicate sample names, and/or various name formats, all of which make it hard for lab scientists to recognize a sample.

Clarity LIMS programmers often rename all incoming samples to a certain naming convention.

This section provides an example to address this problem.

Recommendations

- When accepting a project and its samples, the receiving lab scientist runs a Clarity LIMS step named Receive Samples.
- The underlying Receive Samples process type/master step is configured with analyte (sample) inputs and no analyte outputs.

- A shared result file output is configured to capture logging from the script.
- The sample name could be a derivative of the Sample LIMSID with a prefix:
 - Because the LIMSID is guaranteed to be unique, this approach mitigates any need to maintain an external sequence of numbers.
 - The Sample LIMSID is derived from the Project LIMSID, which is configurable.

Proposed Solution

- The Receive Samples process is configured to trigger a script that renames the samples that are input to the process.
- This trigger also passes the `OriginatingProcessURI` to the script. This example assumes that the original submitted sample name must be preserved, so it is saved in a sample UDF.

The following pseudocode shows how one might implement the sample renaming script:

1. Connect to the API, using the `OriginatingProcessURI`.
2. Retrieve the `OriginatingProcessXML` and store it in a variable.
3. Iterate through the `inputoutput` map of the `OriginatingProcessXML` and for each `InputArtifact`:
 - GET the `InputArtifactURI` and store the input `ArtifactXML` in a variable.
 - From this `ArtifactXML`, GET the `SourceSampleXML` and store it in another variable.
 - Modify the `SourceSampleXML`. To do this modification, copy the original `SampleName` to an `OriginalSampleNameUDF`.
 - Rename the `SampleName` to a desired name (see [Recommendations](#)).
 - Finally, PUT the `Sample XML` back.

Copying UDF Values from Source to Destination

How to copy the value of a UDF/custom field from source to destination (typically from the inputs of a process/step to the outputs) is a frequently asked question.

For example, suppose a process/step takes in libraries and tracks their normalization. In such a case, the input samples have a UDF/custom field that is used to track the library ID. Since the library ID changes, it is desirable for the output samples to also have this ID.

Solution

Use the API to gather the XML for the inputs, then copy the XML node relating to the UDF/custom field to the outputs.

Alternatively, use the out-of-the-box `copyUDFs` script, which Illumina provides as part of the NextGen Sequencing configuration.

Script

The `copyUDFs` script is available in the `ngs-extensions.jar` archive, and can be called from the EPP/automation parameter string.

The archive file may be named differently, depending upon the version you are running.

Usage:

```
java -jar ngs-extensions.jar -u {username} -p {password} -i {processURI} \  
script:copyUDFs -f <myUDF1>,<myUDF2>,<myUDF3>
```

Defining the UDF/Custom Field Values

The UDF/custom field values to be copied are defined in the `-f` portion of the syntax. These values must be present on both the inputs and outputs of a process.

For example, suppose you wanted to use this script to copy the value of a UDF called `Library ID`:

- The Library ID field must be defined on both inputs and outputs.
- The `-f` flag is defined as follows.

```
-f "Library ID"
```

To copy multiple UDF value from source to destination, list them in comma-separated form as part of the `-f` flag.

- To copy Library ID and Organism from source to destination, use the following example:

```
-f "Library ID", "Organism"
```

Updating Preset Value of a Step UDF through API

Use the API to update the preset value of a user-defined field (UDF)/custom field configured on a step.

From your test server:

1. GET a chosen UDF/custom field.
2. Do a PUT and include a new `<preset>` line.

For example, to add 'My new preset', insert the preset (`<preset>My new preset</preset>`), after your last `<preset>` value in your XML:

```
.....  
<preset>Your last preset</preset>  
<preset>My new preset</preset>  
<is-required>Your original value</is-required>  
<attach-to-category>Your original value</attach-to-category>  
</cnf:field>
```

This tool is powerful when integrating with external systems and combined with the Begin Work trigger. For example, it can be used to reach out to an external source with a script, initiated with the Begin Work trigger. The script makes sure that the presets for the Step Details UDFs/custom fields are always up to date and in sync with the server—before entering the Record Details screen.

Automating BCL Conversion

When a sequencing run is complete, it is often desirable to pass data to CASAVA for BCL conversion automatically rather than manually. This section proposes a method to configure this automation.

⚠️ | This solution is not tested end-to-end on an instrument.

The proposed approach involves adding an automation trigger to the Sequencing step, such that it invokes a script that launches the BCL Conversion step.

However, because the BCL Conversion step does not run immediately, it is launched in a dormant state until the Sequencing step is complete.

The key event here is the Run Report that is created and attached to the Sequencing step. As the last event to occur in the step, the creation of this report is used to prompt the BCL Conversion step to 'wake up' from its dormant state and begin processing.

The following pseudocode describes the work that must occur within the script:

```
Harvest the command line parameters
```

```
Use the API to convert the LIMS ID (that was passed to the script as the -r parameter) to a full URI
```

```
Craft the XML required to create the Input-Output Map for the BCL Conversion Process with -- the URI we just discovered as the input artifact to the Input-Output Map
```

```
POST the XML to the API
```

Solution Details

Step 1: Create Script to Launch BCL Conversion Process/Step

A required script that launches the BCL Conversion step via the API might be absent. The creation of such a script is covered in [Process Execution with EPP/Automation Support](#). This example only covers the functionality of the script rather than code.

In addition to the expected processURI, username, and password parameters/tokens, the script should accept another parameter (the LIMSID of the Run Report from the Sequencing step).

For example, the script can be invoked as follows:

```
/path/to/script/scriptname -i {processURI:v2:http} -u {username} \-p {password} -r {runreportLIMSID}
```

Use this syntax when configuring the command line on the Sequencing process/step.

Step 2: Configure the Clarity LIMS Web Interface

Configure the automation so the script is automatically triggered when exiting the Record Details screen.

Example Configuration Details on a Test System

1. The BCL Conversion process is configured:

- To take in a ResultFile input and generate a nonshared ResultFile output
 - With a process parameter of 'Standard,' which initiates the actual BCL conversion.
2. The script is passed the value '92-3771' as the -r parameter.
 3. The script is converted to a full URI and became the input element of the following XML, which is POSTed to the /processes API resource:

```
<?xml version="1.0" encoding="UTF-8"?>
<prx:process xmlns:prx="http://genologics.com/ri/processexecution">
  <type>BCL Conversion</type>
  <technician uri="http://localhost:8080/api/v2/researchers/1"></technician>
  <input-output-map shared="false">
    <input uri="http://localhost:8080/api/v2/artifacts/92-3771"></input>
    <output type="ResultFile"></output>
  </input-output-map>
  <process-parameter name="Standard"></process-parameter>
</prx:process>
```

Notes

- Update all URIs in the XML to point to the hostname and API version for the system.
- Provide a valid URI for the lab scientist. There might be a user in the system with LIMS ID of '1'.
- If the POST is successful, the API returns the valid XML for the created process.

Note: This scenario is one of the few occasions where the POST succeeds, yet returns XML that differs from the input XML. The results can be confusing, because a standard approach for validating whether POSTs are successful is to compare the output XML with the input. If they differ, assume that the POST failed. However, in this scenario it did not fail.

Finding QC Flags in Aggregate QC (Library Validation) via REST API

When running the Aggregate QC step in Clarity LIMS, the QC pass and fail flags for the samples display in the Record Details screen.

This section explains how to use the API instead to find the samples that passed or failed QC aggregation.

Query the API and filter the results list based on the qc-flag parameter value. For more on filtering, see [Filtering List Resources](#) section.

- To filter the list by QC flag with a value of PASSED, use the following example:

```
http[s]://<hostname/IP address>:<port>/api/<version of api>/artifacts?qc-flag=PASSED
```

- To find an individual QC flag result for an individual sample, use the LIMS ID of the sample:

```
http[s]://<hostname/IP address>:<port>/api/<version of api>/artifacts/<analyte artifact lims id>
```

- Then search for the value of the <qc-flag> element of the endpoint payload for the artifact.

The <qc-flag> element of the input analyte (sample) artifact is sent into the Aggregate QC step.

To demonstrate this detail, review the following steps:

1. In the API, find a single analyte artifact (derived sample) that has passed QC. The XML QC flag value is PASSED.
2. In Clarity LIMS, find the same sample and change the value of the <qc-flag> element from PASSED to FAILED. Save the change.
3. In the API, find the sample again. See that the XML QC flag value is set to FAILED.

Setting the Value of a QC Flag on an Artifact

The QC flag parameter qc-flag can be set on an input or output analyte (derived sample) or on an individual result file (measurement) with a few lines of Groovy code.

In the following example, the qc-flag value of the analyte artifact is set based on the value of the bp_size variable when compared to the threshold1 and threshold2 variables.

```
artifact = GLSRestApiUtils.httpGET(artifactURI, username, password)
Node foundQCNode = artifact.'qc-flag'[0] as Node
if((bp_size >= threshold1) && (bp_size <= threshold2)){qcFlag = "PASSED"} else{qcFlag = "FAILED"}
```

The following code determines whether a qc-flag value is previously set, such that a flag is only set if one does not exist.

```
foundQCNode ? foundQCNode.setValue(qcFlag) : addChild(new Node(null, 'qc-flag', qcFlag),
artifact, "qc-flag")
GLSRestApiUtils.httpPUT(artifact, artifactURI, username, password)
```

Creating Notifications When Files are Added via LabLink

External collaborators can use the LabLink Collaborations Interface (supported in LIMS v4.x and earlier) to upload and attach files to their projects and samples. Lab scientists might require a notification on these rare occasions, but 'Notification' is defined different things to different people.

Usually email notification is sufficient, but Twitter is increasing as a method for notification.

This topic explains how to:

1. Detect when files have been uploaded.
2. Extract the key information that might comprise a notification.

The Files API Resource

The key resource to investigate is the files resource, which provides a listing of files within the system.

Step 1: Access the Files Resource

On a test system accessing the files resource s follows:

```
http://192.168.9.123:8080/api/v2/files
```

produces the following output:

```
<file:files>
  <file limsid="92-932-40-16" uri="http://192.168.9.123:8080/api/v2/files/92-932-40-16"/ >
  <file limsid="92-944-40-18" uri="http://192.168.9.123:8080/api/v2/files/92-944-40-18"/>
  <file limsid="92-943-40-20" uri="http://192.168.9.123:8080/api/v2/files/92-943-40-20"/>
  <file limsid="92-945-40-22" uri="http://192.168.9.123:8080/api/v2/files/92-945-40-22"/>
  <file limsid="92-942-40-24" uri="http://192.168.9.123:8080/api/v2/files/92-942-40-24"/>
  <file limsid="92-941-40-26" uri="http://192.168.9.123:8080/api/v2/files/92-941-40-26"/>
  <file limsid="ADM102A1DN3-40-28" uri="http://192.168.9.123:8080/api/v2/files/ADM102A1DN3-40-28"/>
  ...
  <file limsid="92-2587-40-459" uri="http://192.168.9.123:8080/api/v2/files/92-2587-40-459"/>
  <file limsid="92-2590-40-460" uri="http://192.168.9.123:8080/api/v2/files/92-2590-40-460"/>
  <file limsid="92-2594-40-462" uri="http://192.168.9.123:8080/api/v2/files/92-2594-40-462"/>
  <file limsid="92-2595-40-463" uri="http://192.168.9.123:8080/api/v2/files/92-2595-40-463"/>
  <file limsid="92-2596-40-465" uri="http://192.168.9.123:8080/api/v2/files/92-2596-40-465"/>
  <file limsid="92-2597-40-466" uri="http://192.168.9.123:8080/api/v2/files/92-2597-40-466"/>
  <file limsid="40-501" uri="http://192.168.9.123:8080/api/v2/files/40-501"/ >
  <file limsid="40-558" uri="http://192.168.9.123:8080/api/v2/files/40-558"/>
  <file limsid="92-4555-40-609" uri="http://192.168.9.123:8080/api/v2/files/92-4555-40-609"/>
  <file limsid="92-4554-40-610" uri="http://192.168.9.123:8080/api/v2/files/92-4554-40-610"/>
  <file limsid="92-4556-40-611" uri="http://192.168.9.123:8080/api/v2/files/92-4556-40-611"/>
  <file limsid="92-4564-40-612" uri="http://192.168.9.123:8080/api/v2/files/92-4564-40-612"/>
  <file limsid="92-4563-40-613" uri="http://192.168.9.123:8080/api/v2/files/92-4563-40-613"/>
  <file limsid="92-4562-40-614" uri="http://192.168.9.123:8080/api/v2/files/92-4562-40-614"/>
  <file limsid="ACC151-40-651" uri="http://192.168.9.123:8080/api/v2/files/ACC151-40-651"/>
  <file limsid="ACC151A1-40-652" uri="http://192.168.9.123:8080/api/v2/files/ACC151A1-40-652"/>
</files>
```

Step 2. Filter the Files URI

Although not useful by itself, the files URI becomes more interesting when filtered to only include files uploaded after a specified date-time, and with a published status of 'true'.

For example, the following URI:

```
http://192.168.9.123:8080/api/v2/files?last-modified=2013-09-10T00:00:00-08:00
```

produces this output on a test system:

```
<file:files>
  <file limsid="ACC151-40-651" uri="http://192.168.9.123:8080/api/v2/files/ACC151-40-651"/>
  <file limsid="ACC151A1-40-652" uri="http://192.168.9.123:8080/api/v2/files/ACC151A1-40-652"/>
</file:files>
```

This outcome is much more manageable. Because they are uploaded via the Collaborations Interface, they inherently have a published status of 'true'. We use this status to exclude regular files uploaded to the LIMS via other methods and interfaces.

Step 3: Retrieve the XML Representations of the Files

By following the URIs to retrieve the full XML representations of these files, the output is similar to the following:

```
<file:file uri="http://192.168.9.123:8080/api/v2/files/ACC151-40-651" limsid="ACC151-40-651">
  <attached-to>http://192.168.9.123:8080/api/v2/projects/ACC151</attached-to>
  <content-location> sftp://192.168.9.123/home/glsftp/ACC151/ACC151-40-651.csv </content-location>
  <original-location>GLims.csv</original-location>
  <is-published>true</is-published>
</file:file>
```

and:

```
<file:file uri="http://192.168.9.123:8080/api/v2/files/ACC151A1-40-652" limsid="ACC151A1-40-652">
  <attached-to>http://192.168.9.123:8080/api/v2/samples/ACC151A1</attached-to>
  <content-location> sftp://192.168.9.123/home/glsftp/ACC151/ACC151A1/ACC151A1-40-652.png </content-location>
  <original-location>image001.png</original-location>
  <is-published>true</is-published>
</file:file>
```

Step 4: Retrieve Project/Sample Information

Retrieve the associated project/sample, and extract the names and/or IDs to embed into the notification, by following the URI in the 'attached-to' elements.

In this case, the following result is produced:

```
<prj:project uri="http://192.168.9.123:8080/api/v2/projects/ACC151" limsid="ACC151">
  <name>Scaffold POC</name>
  <open-date>2013-04-18</open-date>
  <researcher uri="http://192.168.9.123:8080/api/v2/researchers/3" />
  <file:file limsid="ACC151-40-651" uri="http://192.168.9.123:8080/api/v2/files/ACC151-40-651" />
</prj:project>
```

and:

```
<smp:sample uri="http://192.168.9.123:8080/api/v2/samples/ACC151A1" limsid="ACC151A1">
  <name>SG-5926 -1</name>
  <date-received>2013-04-18</date-received>
  <project limsid="ACC151" uri="http://192.168.9.123:8080/api/v2/projects/ACC151"/ >
  <submitter uri="http://192.168.9.123:8080/api/v2/researchers/1">
    <first-name>System</first-name>
    <last-name>Administrator</last-name>
  </submitter>
  <artifact limsid="ACC151A1PA1"
uri="http://192.168.9.123:8080/api/v2/artifacts/ACC151A1PA1?state=451" />
  <udf:field type="Boolean" name="Control?">false</udf:field>
  <udf:field type="String" name="Category">treated</udf:field>
  <file:file limsid="ACC151A1-40-652" uri="http://192.168.9.123:8080/api/v2/files/ACC151A1-40-652"/>
</smp:sample>
```

Proposed Solution

A script must be run periodically (hourly/daily) that queries the files resource for files that have a published status of true, and are last modified in the period of interest.

After this list of files is retrieved, the following pseudocode can be applied:

```
For each URI in files list:
  Retrieve the file object
  STORE the contents of the 'original-location' element
  IF the contents of the 'attached-to' element contains EITHER 'projects' OR 'samples':
    Retrieve the object pointed to by the 'attached-to' element
    STORE the contents of the 'name' element
```

```
Compile the stored elements into a notification
```

```
Publish all notifications produced in an atomic or molecular fashion, as required
```

An example derived from the previous XML can lead to the following notifications:

```
FILE: GLims.csv was recently uploaded to PROJECT: Scaffold POC
```

```
FILE: image001.png was recently uploaded to SAMPLE: SG-5926-1
```

Remote HTTP Filestore Setup

Server-side configuration allows for configuration of multiple filestores to be associated to entities (samples, projects, processes/steps, and so on) in BaseSpace Clarity LIMS.

This feature allows for linking to large data files on a different server, eliminating the need to move large files on the Clarity LIMS filestore. Large files can include results, images, and searches, and so on.

For example, sequencing instruments typically produce large result files. Attaching these files to the Sequencing step in Clarity LIMS results in the following drawbacks.

- Involves transferring the files to the Clarity LIMS filestore. The larger the file, the slower the transfer speed.
- Requires a large amount of space as runs build up.

An alternative solution is to set up a remote filestore to be used as the results directory from which Clarity LIMS accesses the files directly.

To do this setup, three steps are required:

1. Set up HTTP, HTTPS, FTP, or SFTP access to the files and folders you wish to share.
2. Configure the Clarity LIMS server to recognize the URI of a file on the remote filestore.
3. POST information to Clarity LIMS, via the REST API, to reference the file from a Clarity LIMS entity (project, sample, process/step, result file, and so on).

Setting Up Access to the Files

BaseSpace Clarity LIMS can operate with many different forms of file servers – HTTP, HTTPS, FTP, and SFTP access are all supported.

It is your responsibility to set up this access. For HTTP, you may be interested in [httpd](#) or [HFS](#) for HTTP file serving.

Configuring the LIMS Server

Properties Required for Tracking a New Remote Filestore

To track a new remote filestore, Clarity LIMS requires four database properties: directory, host, port, and scheme.

The four properties share a base name, but have different suffixes attached (dir, host, port, scheme). These suffixes are summarized in the following table.

- The base name can be anything. Clarity LIMS finds any base names that end in .scheme and uses that base name to find the other information.

- If necessary, add the last three properties listed in the table (with the .domain, .user, and .password suffixes) to specify a domain, username, and password to be used when accessing files.

Property Name	Usage	Example Value	Required	Description
Directory	\${baseName}.dir	/limsdata	True	The highest level directory in which it is valid to access files. Files outside this directory are not attached.
Hostname/IP	\${baseName}.host	YourHTTPHost	True	The hostname or IP address to use when accessing the files.
Port	\${baseName}.port	80	True	The port to use when accessing the files.
Scheme	\${baseName}.scheme	http	True	The scheme of the URI used to access the files. Examples are HTTP, HTTPS, FTP, and SFTP.
Domain	\${baseName}.domain	YourAuthDomain	False	The domain to use when authenticating access to the files.
Username	\${baseName}.user	fileUser	False	The username to use when authenticating access to the files.
Password	\${baseName}.password	filePassword	False	The password to use when authenticating access to the files. It is highly recommended that you encrypt your password. See the following section for details.

- **LIMS 4.x and earlier**—For the property changes to take effect, JBoss must be restarted.
- **LIMS v5 and later**—For the property changes to take effect, Tomcat must be restarted.

Working with the Database Properties

Use the `omxprops-ConfigTool.jar` to create, update, and retrieve values of the database properties. This tool is found at the following location: `/opt/gls/clarity/tools/propertytool`

To create a property, use the following examples:

```
java -jar omxprops-ConfigTool.jar addPropertyType <baseName><suffix> /<value>
<description>
```

i | These properties may not be global properties. Do not use the `-g` property here.

To get the value of an existing property:

```
java -jar omxprops-ConfigTool.jar get <baseName><suffix>
```

To update the value of an existing property:

```
java -jar omxprops-ConfigTool.jar set -y <baseName><suffix> <newValue>
```

To encrypt a password:

```
java -jar omxprops-ConfigTool.jar encryptPassword <password>
```

i | To set a property to the encrypted result, set the value as `ENC(<result>)`.

See also [Creating Encrypted Passwords](#).

Example: Mapping a Remote HTTP URI

The following example maps a remote HTTP URI:

```
http://YourHTTPHost:80/limsdata/LegacyFile.RAW
```

In this case, the base name for the properties is `http-lims-files`.

Steps

1. As the `glsglboss` user, access the `omxprops` property tool in `/opt/gls/clarity/tools/propertytool`.

2. Add the following `dir`, `host`, `port`, and `scheme` properties to the server from the command line:

```
java -jar omxprops-ConfigTool.jar addPropertyType http-lims-files.dir '/limsdata' 'Alternate
filestore server dir'
```

```
java -jar omxprops-ConfigTool.jar addPropertyType http-lims-files.host 'YourHTTPHost'
'Alternate filestore host'
```

```
java -jar omxprops-ConfigTool.jar addPropertyType http-lims-files.port '80' 'Alternate
filestore port'
```

```
java -jar omxprops-ConfigTool.jar addPropertyType http-lims-files.scheme 'http' 'Alternate
filestore server type'
```

In the example above, the `<http-lims-files.dir>` parameter value is `/limsdata`. Any file in `http://<YourHTTPHost>/limsdata/` is available to be referenced by BaseSpace Clarity LIMS.

For all files on the web server to be available, use the / parameter value, for example:

```
java -jar omxprops-ConfigTool.jar addPropertyType -y http-lims-files.dir / Alternate  
filestore server dir
```

Posting Information to the LIMS Referencing Files on the Remote Filestore

After the filestore properties are added to Clarity LIMS (and JBoss/Tomcat has been restarted, as applicable), you can attach the files to Clarity LIMS.

To attach the files to Clarity LIMS:

- POST to `http://hostname/api/v2/files`, with the `content-location` tag pointing to the remote filestore.

An example XML POST is provided, using the filestore created in the previous example:

```
<file:file xmlns:file="http://genologics.com/ri/file">  
  <content-location>http://YourHTTPHost:80/limsdata/LegacyFile.RAW</content-location>  
  <attached-to>http://hostname/api/v2/artifacts/92-148</attached-to>  
  <original-location>http://YourHTTPHost:80/limsdata/LegacyFile.RAW</original-location>  
  <is-published>false</is-published>  
</file:file>
```

Results

- The file is now downloadable directly from Clarity LIMS.
- Any entity that can have a file attached to it may be referenced in the `<attached-to>` parameter.
- For LIMS v4.x and earlier, if you would like the file to be accessible from the LabLink Collaborations Interface, change the `<is-published>` value to `true`.

For more information on working with files, see [Work with Files](#).

About the Cookbook

The Clarity LIMS Cookbook uses example scripts to help you learn how to work with REST and EPP automation scripts. Cookbook recipes are small, specific how-to articles designed to help you understand REST and automation script concepts. Each recipe includes the following:

- Explanations about a concept and how a particular programming interface is used in a script.
- A snippet of script code to demonstrate the concept.



The API documentation includes the terms External Program Integration Plug-in (EPP) and EPP node.

As of BaseSpace Clarity LIMS v5.0, these terms are deprecated. The term EPP has been replaced with automation. EPP node is referred to as the Automation Worker or Automation Worker node. These components are used to trigger and run scripts, typically after lab activities are recorded in the LIMS.

The best way to get started is to download the example script and try it out. After you have seen how the script works, you can dissect it and use the pieces to create your own script.

Get Started with the Cookbook

Before downloading your first script, do the following actions:

- Familiarize yourself with the API Cookbook prerequisites and key concepts found in [Development Prerequisites](#) and [REST General Concepts](#).
- Use a non-production server for script development.
- Familiarize yourself with the coding language.
- Use the GLSRestApiUtils file to assist with recipe development.
- Review [Tips and Troubleshooting](#).

Script Development with a Non-Production Server

The example script recipes really come to life when you change them and see what happens. Running the scripts often requires new custom fields and master steps to be added to the system. You need unrestricted access to development and test servers (licensed as non-production servers) with Groovy (a coding language). You also need an AI node/automation worker installed so that you can experiment freely.

For more information and recommendations for deploying and copying scripts in development, test, and product environments, refer to [Useful Tools](#).

Script Types

There are two coding languages used for the examples provided in Cookbook: Groovy and Python.

Groovy

The Cookbook Recipe Examples are written in Groovy. Many of our examples use the following Groovy concepts:

- Closures
- The each method

Groovy closures are essentially blocks of code that can be stored for later use.

The each method takes a closure as an argument. Then, the method iterates through each element in a collection, performing the closure on the element, which is stored in the `it` variable by default. The following example shows the each method:

```
outputNodes.each {  
    GLSRestApiUtils.setUdfValue(it, 'Library Size', '25')  
}
```

Python

The Cookbook also provides a few examples written in Python, which uses the minidom module. The following script shows how the minidom module is used:

```
dom = parseString(pXML)
elementList = dom.getElementsByTagName("udf:field")
for element in elementList:
    name = element.getAttribute("name")
    if name == udfName:
        udf = api.getInnerXml(element.toxml(), "udf:field")
```

This same functionality can be obtained using any programming language capable of interacting with the Web API. For more information on the minidom module, refer to [Python Minidom](#).

Use GLSRestApiUtils

In addition to the Groovy file example attached to each Cookbook recipe page, most recipes require the `glsapiutil.py` file, which is available on our GitHub repository. The mature `glsapiutil.py` library is strictly for Python 2. A newer version, `glsapiutil3.py`, works with Python 3.

For more information on these files, refer to [Obtain and Use the REST API Utility Classes](#).

Tips and Troubleshooting

When reading a recipe, look for file attachments. Almost all examples have an attached Groovy script to download. To use the scripts with a non-production server, edit the script to include your server network address and credentials.

For illustration purposes, most scripts use populated information. You must add your own sample, process (eg, a master step in Clarity LIMS v5 and later), and other data. The non-production server has a directory set up for this purpose at `/opt/gls/clarity/customextensions/`.

When using full production scripts, the following considerations must be taken:

- Cookbook scripts are written to explain concepts. They are not deeply engineered code written in a defensive programming style. Always think through the expected and unexpected input of your scripts when incorporating concepts or code from Cookbook recipe examples.
- Full production servers can require different configurations for scripting languages other than Groovy, and for the EPP/automation worker node. For example, your script directory can be accessible by the user account running the EPP/automation worker node for User Interface (UI) triggers.

Discuss the software deployment plans with your system administrator to coordinate between non-production and production servers. For more information on using production scripts, refer to [REST General Concepts](#) and [Automation Overview](#).

Version Compatibility

Each recipe was written with a specific API version. For information on how to check the version of the API on your system, refer to [Requesting API Version Information](#).

Apache Groovy is required for most Cookbook examples. It is open source and is available under an Apache license from groovy-lang.org/download.html. It is installed on non-production servers, but you can also install it to your desktop. The Cookbook examples were developed with Groovy v1.7.

Python is required for some Cookbook examples. It is available from www.python.org/download. The Cookbook examples were developed with Python v2.7.

Path to Groovy

The automation worker node executing the command uses the first instance of Groovy it finds in the executable search path for the limited shell. This is the `$PATH` variable.

If you have multiple versions of Groovy (or multiple users using different versions) and experience problems with your command-line calls, declare the full path to Groovy/Java in your command.

To see your executable search path, and other environment variables available to you, run the following command:

```
bash -c "env > {outputFileLuid0}.txt"
```

Compare this command to the full logon shell, which is `bash -l -c "env > {outputFileLuid0}.txt"`.

```
bash -l -c "env > {outputFileLuid0}.txt"
```

For more information on command-line actions, refer to [Supported Command Line Interpreters](#).

References

For details on the programming interface methods and data elements available, refer to the following documentation:

- [Resources and Reference](#)
- [Automation Tokens](#)
- [Automation/EPP Parameters \(LIMS v4.x & earlier\)](#)

Browser Plug-ins

Browsing for, and adjusting resources, in Firefox, Chrome, or other browsers is great for getting started or for troubleshooting.

The following plug-ins are available with Firefox:

- **Text Link**—Makes any URI in the XML a hyperlink.
- **Linkificator**—Converts text links into selectable links.
- **RESTClient**—Provides a simple interface to call HTTP methods on REST resources. It is useful for troubleshooting, checking error codes, and for getting comfortable with GET, PUT, and POST requests.

The following plug-ins are available with Chrome:

- **Advanced REST Client**—Provides similar functionality to Poster by Firefox.
- **XML Tree**—Displays XML data in a user-friendly way.

Obtain and Use the REST API Utility Classes

This page is maintained for posterity, but customers are encouraged to visit the [GitHub](#) repository for all subsequent updates to the library (including changelogs). Unless otherwise specified, changes are only made in the Python version of the library.

Change Log

Dec 19, 2017:

- glsapiutil v3 ALPHA (bleeding-edge library) released on GitHub. GitHub has the most current library.
- Links to library removed from this page.

Dec 15, 2016:

- reportScriptStatus() function had a bug that caused it not to work when a <message> node was unavailable. This bug has been fixed.
- deleteObject() functions now available for both v1 and v2 of the library.
- getBaseURI() now returns a trailing slash at the end of the URI string.
- getFiles() function added to batch retrieve files.



The Python `glsapiutil.py` and `glsapiutil3.py` classes are now available on GitHub. GitHub has the most current libraries. `glsapiutil3.py` works with both Python v2 and v3.

Legacy Overview

The GLSRestApiUtils utility class provides a consistent way to perform common REST operations, such as REST HTTP methods or common XML string manipulation. It is a utility class written in Python and Groovy for the API Cookbook examples. This utility class is specific to the Cookbook examples. The class is not required for the API with Groovy or Python, as there are many other ways to manipulate HTTP and XML in these languages. However, it is required if you want to run the Cookbook examples as written. It is also not part of REST or EPP/automation.

Use UtilityCalls in your Scripts

Almost all Cookbook example files use the HTTP methods from the GLSRestApiUtils class.

The HTTP method calls in Groovy resemble the following example:

```
returnNode = GLSRestApiUtils.httpGET(uri, username, password)
returnNode = GLSRestApiUtils.httpPUT(inputNode, uri, username, password)
returnNode = GLSRestApiUtils.httpPOST(inputNode, uri, username, password)
```

In this example, the returnNode and inputNode are Groovy nodes containing XML. The XML in the returnNode contains the XML available from the server after a successful method call. If the method call was unsuccessful, the XML contains error information. The following is an example of the XML manipulation functions in the utility:

```
println GLSRestApiUtils.nodeToXmlString(returnNode)
```

As you can see from these examples, the utility class is easy to include in your scripting. The code is contained in the GLSRestApiUtils files attached to this page.

Deploy a Groovy Script that Uses the Utility Class

To deploy a Groovy script that uses the utility class, you must include the directory containing `GLSRestApiUtils.groovy` in the Groovy class path.

Groovy provides several ways to package and distribute source files, including the following methods:

- Call Groovy with the `-classpath` (or `-cp`) parameter.
- Add to the `CLASSPATH` environment variable.
- Create a `~/.groovy/lib` directory for jar files for common libraries.

If you would like to experiment with the Cookbook examples, you can also copy the file into the same directory as the example script.

Use the Python Library

The Python library (`glsapiutil.py`) is functionally similar to the Groovy library. Python library v3 works for both Python 2 and 3, but the more mature `glsapiutil.py` must be used for production code.

Setting up the API in Python requires the following information:

- hostname
- API version
- LIMS API account username and password

The following example uses the platform library to get the hostname from the server directly. Some application examples require getting the hostname from an input parameter (usually a step or process URI).

```
import platform

HOSTNAME = platform.node() # get the system hostname
VERSION = 'v2' # API version 2
BASE_URI = HOSTNAME + '/api/' + VERSION + '/'

USERNAME = 'username'
PASSWORD = 'password'api = None

[...]

def main():
    global api
    api = glsapiutil.glsapiutil2() # initialize the API object, using version 2 of the utils
    api.setHostname( HOSTNAME ) # set the hostname, currently taken from the system
    api.setVersion( VERSION ) # the version, currently set to 'v2'
    api.setup( USERNAME, PASSWORD ) # authenticate with the Clarity API user credentials
```

Python Library Functions

The HTTP method calls for the Python version of the library resemble the following:

```
returnXML = api.GET( uri )
returnXML = api.PUT( xml, uri )
returnXML = api.POST( xml, uri )
```

Unlike with the Groovy library, the rest functions in the Python library require XML (text) as input (not DOM nodes). The return values of the GET, PUT, and POST functions are also XML text.

Deploy Scripts with the Python Library in Clarity LIMS (v4 and Later)

If a script must work with a running process or step, it is normal to use either the {processURI:v2} or the {stepURI:v2} tokens. The following example has the {stepURI:v2} token:

```
bash -l -c "/usr/bin/env python /opt/gls/clarity/customextensions/myScript.py
-u {username} -p {password} -s {stepURI:v2} -f {compoundOutputFileLuid0}"
```

In LIMS v4 and above, these tokens sometimes resolve to `https://localhost:9080/api/v2/...` instead of the expected HOSTNAME. Setting up the API object with a hostname other than `https://localhost:9080` can cause Access Denied errors. To avoid this issue, alter the API authentication code slightly as follows.

```
import platform

HOSTNAME = platform.node() # get the system hostname
VERSION = 'v2' # API version 2
BASE_URI = HOSTNAME + '/api/' + VERSION + '/'

USERNAME = 'username'
PASSWORD = 'password'
api = None

[...]

def setupGlobalsFromURI( uri ):
    global HOSTNAME
    global VERSION
    global BASE_URI

    tokens = uri.split( '/' )
    HOSTNAME = '/'.join( tokens[0:3] )
    VERSION = tokens[4]
    BASE_URI = '/'.join( tokens[0:5] ) + '/'
```



```
def main():
    global api
    api = glsapiutil.glsapiutil2() #initialise the API object, using version 2 of the API

    setupGlobalsFromURI( args.stepURI ) # assuming the step URI was passed by the EPP /
    automation trigger

    api.setHostname( HOSTNAME ) # set the hostname, currently taken from the system
    api.setVersion( VERSION ) # the version, currently set to 'v2'
    api.setup( USERNAME, PASSWORD ) # authenticate with the Clarity LIMS API user
    credentials
```

The changes are highlighted in red. This code takes the resolved {stepURI:v2} token (assumed to be stored in the args object) and resets the HOSTNAME variable to the new value (eg, <https://localhost:9080>) before authenticating.

These changes are fully backward-compatible with Clarity LIMS v4 or earlier. The EPP/automation URI tokens resolve to the expected hostname, and the setupGlobalsFromURI() function still parses it correctly.

 | On [GitHub](#), in addition to the libraries, a `basic_complete_recipe.py` script that contains the skeleton code is needed to get started with the Python API. This script also includes the modifications required to work with Clarity LIMS v4 and later. The legacy Groovy library can still be obtained using the attachment.

Attachments

[GLSRestApiUtils.groovy.txt](#)

Work with EPP/Automation and Files

You can configure the automation trigger and use automation to invoke any external program that runs from a command line. Refer to the following for details:

- [Automation Trigger Configuration](#)
- [Process Execution with EPP/Automation Support](#)

 | EPP automation/support is compatible with API v2 r21 and later.

The API documentation includes the terms External Program Integration Plug-in (EPP) and EPP node.

As of BaseSpace Clarity LIMS v5.0, these terms are deprecated. The term EPP has been replaced with automation. EPP node is referred to as the Automation Worker or Automation Worker node. These components are used to trigger and run scripts, typically after lab activities are recorded in the LIMS.

Automation Trigger Configuration

Automations (formerly referred to as EPP triggers or automation actions) allow lab scientists to invoke scripts as part of their workflow. These scripts must successfully complete for the lab scientist to proceed to the next step of the workflow.

i | EPP automation/support is compatible with API v2 r21 and above.

The API documentation includes the terms External Program Integration Plug-in (EPP) and EPP node.

As of BaseSpace Clarity LIMS v5.0, these terms are deprecated. The term EPP has been replaced with automation. EPP node is referred to as the Automation Worker or Automation Worker node. These components are used to trigger and run scripts, typically after lab activities are recorded in the LIMS.

Automations have various uses, including the following:

- **Workflow enforcement**—Makes sure that samples only enter valid protocol steps.
- **Business logic enforcement**—Validates that samples are approved by accounting before work is done on them. This automation can also make sure that selected samples are worked on together.
- **Automatic file generation**—Automates the creation of driver files, sample sheets, or other files specific to your protocol and instrumentation.
- **Notification**—Notifies external systems of lab progress. For example, you can notify Accounting of completed projects so that they can then bill for services rendered.

Configuration

You can enable automations on master steps in two configuration areas of Clarity LIMS:

- On the Automations tab, when adding or configuring an automation. For more information, refer to [Add and Configure Automations](#).
- On the Lab Work tab, on the master step configuration form. For more information, refer to [Configure Master Steps and Steps](#).

After it is enabled on a master step, the automation becomes available for use on all steps derived from that master step.

You can configure the automation trigger on the master step, or on the steps derived from that master step.

AUTOMATION

Step Automations are assigned to Master Steps. Triggers for each Automation can be enabled on either Master Steps or Steps.

Automation Name	Trigger Location	Trigger Style
Generate NanoDrop driver file	Record Details >	Automatic upon entry >
Parse NanoDrop NDT and assign QC flags	Record Details >	Manual button >

Script Messages

When you use scripts, various messages can display the status, progress, and any errors or issues that can occur. The following types of messages occur when executing scripts:

- Progress message
- Non-responsive script message
- Non-blocking success and warning message
- Blocking script failure message

Progress Message

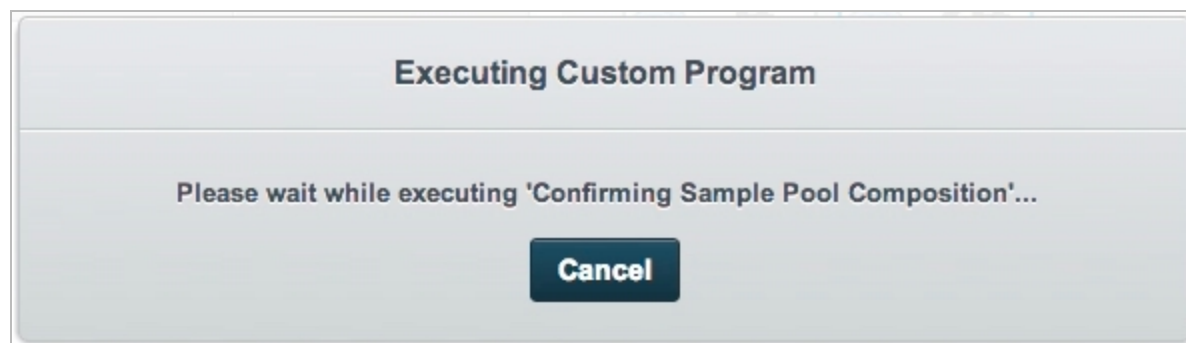
While executing a script, if more than one script would be triggered for a single user action, they are reported in sequence. This reporting continues until all scripts complete, or one of them fails.

An example scenario would be a step that is configured to execute the following:

- One script upon exit of the Placement screen.
- A second script upon entry of the Record Details screen.

In this scenario, when the lab scientist advances their protocol step from the Placement screen to the Record Details screen, the scripts are executed in sequence.

The parameter string/automation name configured on the master step is displayed in a progress message. You can use this feature by giving your parameter strings/automations meaningful names that provide you with context about what the script is doing. The following is an example of a progress message.



You cannot proceed until the script completes successfully.

Non-Responsive Scripts Message

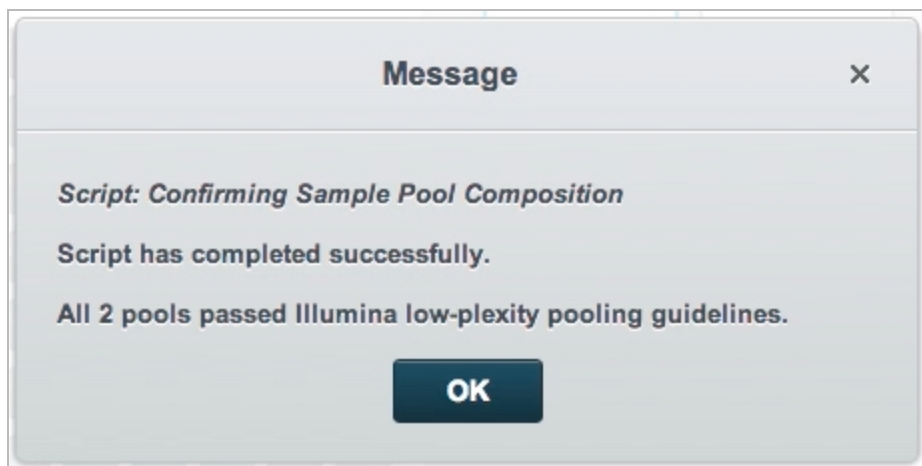
You can request to cancel a script that is not responsive. While canceling abandons the monitoring of script execution, it does not stop the execution of the script.

After canceling a script, follow up with the Clarity LIMS administrator to determine if the AI node/automation worker must be restarted.

Non-Blocking Success and Warning Message

The scientific programmers in your facility can provide you with a message upon successful execution of a script. There are two possible non-fatal messages: OK and WARNING. These messages can be set using the step program status REST API endpoint.

Message boxes display the script name, followed by a message that is set by the script using the step program status REST API endpoint. Line breaks are permitted in the custom message. The following is an example of a success message:



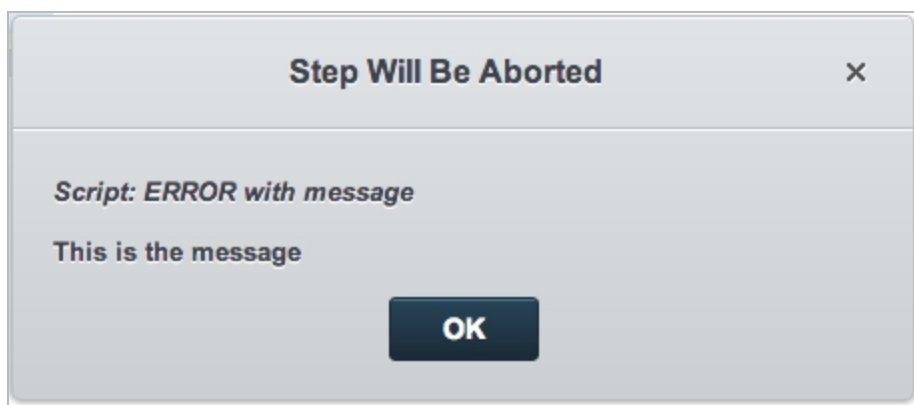
After you select **OK**, you are permitted to proceed in the workflow.

Blocking Script Failure Message

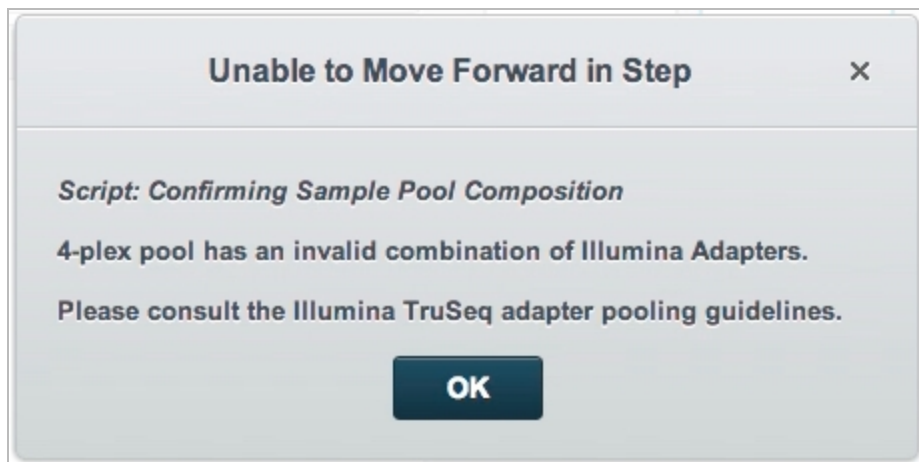
When a script fails, a message box displays. There are two ways to produce fatal messages:

- By using the step program status REST API endpoint (informing FAILURE as the status)
- By generating output to the console and returning a non-zero exit code

For example, when beginning a step, if the script does not allow you to work on the samples together in Ice Bucket, the samples will be returned to Ice Bucket after acknowledging the error message. In this case, the step is prevented from being tracked. The following is an example of a failure message:



If you attempt to advance a step from the Pooling screen, but an error is detected, the error state prevents you from continuing. The following is an example of this type of message:



After you select **OK**, you are prevented from proceeding in the workflow. Instead, you must return to the Pooling screen and address the problem before proceeding.

Process Execution with EPP/Automation Support

At the completion of a process (using API v2 r21 or later), EPP can invoke any external program that runs from a command line. In this example, a process with a reference to a declared EPP program is configured and executed entirely via the API.

i | EPP automation/support is compatible with API v2 r21 and above.

The API documentation includes the terms External Program Integration Plug-in (EPP) and EPP node.

As of BaseSpace Clarity LIMS v5.0, these terms are deprecated. The term EPP has been replaced with automation. EPP node is referred to as the Automation Worker or Automation Worker node. These components are used to trigger and run scripts, typically after lab activities are recorded in the LIMS.

Prerequisites

1. You have defined a process that has the following items:
 - An input of type analyte.
 - A single output per input.
 - A single shared result file.
2. The process type is associated with an external program that has the following requirements:
 - At least one process-parameter defined and named TestProcessParam.
 - The `bash -c "echo HelloWorld > {compoundOutputFileLuid0}.txt"` parameter string.
3. Samples have been added to Clarity LIMS.

Code Example

The example in [Run a Process/Step](#) is similar to this example. The differences are that this example has minimal input/output and posts a reference to a pre-defined EPP process-parameter.

Step 1: Identify the Sample

To run a process on a sample, you must first identify the sample to be used as the input to the process. For this example, run the process on the first highlighted sample.

[27-103202] 27-103202

Identifier: 27-103202

Type: 96 well plate

LIMS ID: 27-103202

Wells

Contents

Samples

Details

Contents				
QC	Well (row:column)	LIMS ID	Name	Submitted S
	A:1	LIT2693A1RO2	RED-qsK-1	RED-qsK-1
	A:2	LIT2693A2RO2	RED-qsK-2	RED-qsK-2

After you have identified the sample, you can use its LIMS ID to as a parameter for the script. The artifact URI is then used as the input in constructing the XML to POST and executing a process.

Step 2: Identify the Container

In addition, this example requires a container in which to store the results of the process execution. An example of how to do this is included in the Groovy script under [Process Execution with EPP/Automation Support](#).

The following code block outlines this action and obtains the URI of the container for the process execution POST.

```
// Create a new container using the StreamingMarkupBuilder
def containerDoc = builder.bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(con: 'http://genologics.com/ri/container')
    mkp.declareNamespace(udf: 'http://genologics.com/ri/userdefined')
    'con:container'{
        'name' ("HiSEQ POST${randomNumGen.nextInt()})")
        'type' (uri:"http://${hostname}/api/v2/containerTypes/1", name:"96 well plate")
    }
}

// Post the new container to the API
containerNode = GLSRestApiUtils.xmlStringToNode(containerDoc.toString())
returnNode = GLSRestApiUtils.httpPOST(containerNode, containersListURI, username, password)
container96WellsURI = returnNode.@uri
```



As shown in other examples, you can use StreamingMarkupBuilder to construct the XML needed for the POST.

Step 3: Construct and POST the XML to Execute the Process

You now have all the pieces of data to construct the XML for the process execution. The following is an example of what this XML looks like.

```
// Create the required URIs
processListURI = "http://${hostname}/api/v2/processes"
researcherURI = "http://${hostname}/api/v2/researchers/1"
analyteURI = "http://${hostname}/api/v2/artifacts/${inputAnalyteLIMSID}"

// Retrieve the Process-Type
processParamName = 'TestProcessParam'
processTypeNode = GLSRestApiUtils.httpGET(processTypeURI, username, password)

// Create a new process using the StreamingMarkupBuilder
def processDoc = new StreamingMarkupBuilder().bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(prx:'http://genologics.com/ri/processexecution')
    'prx:process' {
        'type' (processTypeNode.@name)
        'technician' (uri:researcherURI)
        'input-output-map' {
            'input' (uri:analyteURI)
            'output' (type:'Analyte') {
                'location' {
                    'container' (uri:container96WellsURI)
                    'value' ("A:1")
                }
            }
        }
        'input-output-map' (shared:'true') {
            'input' (uri:analyteURI)
            'output' (type:'ResultFile')
        }
        'process-parameter' (name:processParamName)
    }
}

// Post the process to the API
unresolvedProcessNode = GLSRestApiUtils.xmlStringToNode(processDoc.toString())
```

```
returnNode = GLSRestApiUtils.httpPOST(unresolvedProcessNode, "${processListURI}",
username, password)
println GLSRestApiUtils.nodeToXmlString(returnNode)
```

Elements

Executing a process uses the `processexecution (prx)` namespace. The following elements are required for a successful POST:

- **type**—Name of the process being run.
- **technician uri**— URI for the technician that is listed as running the process.
- **input-output-map**—An input-output-map element for each pair of inputs and outputs.
- **input uri**—URI for the input artifact.
- **output type**— Type of artifact of the output.

If the outputs of the process are analytes, then the following elements are also required:

- **container uri**— URI for the container the output is placed in.
- **value**—Well placement for the output.

To use the configured EPP process, the `process-parameter` element is required. This element is the name of the configured EPP that is executed when this process is posted.

Requirements

The following elements that match the `processParamName` variable must exist in the system before the process can be executed:

- process type
- technician
- input artifact
- container
- EPP parameter

With analyte outputs, if there are no containers with empty wells in the system, you must create one before running the process.

The XML constructed must match the configuration of the process type. For example, if the process is configured to have both analytes and a shared result file as outputs, you must have the following:

- An input-output-map for each pair of analyte inputs and outputs.
- An additional input-output-map for the shared result file.

The name on the process execution XML must match one of the possibly declared EPP parameter names. This requirement is true for any EPP parameters.

Expected Output and Results

If the POST is Successful, then the process XML is returned.

In the following example, there are two <input-output-map> elements. The second instance has the output-generation-type of PerAllInputs. This element indicates that the result file is shared and only one is produced, regardless of the number of inputs.

```
<prc:process uri="http://localhost/api/v2/processes/ROB-QSK-110525-24-109084" limsid="ROB-QSK-110525-24-109084">
  <type uri="http://localhost/api/v2/processtypes/1979">robtype1</type>
  <date-run>2014-11-14</date-run>
  <technician uri="http://localhost/api/v2/researchers/11651">
    <first-name>John-Luck</first-name>
    <last-name> Pikkard </last-name>
  </technician>
  <input-output-map>
    <input post-process-
      uri="http://localhost/api/v2/artifacts/LIT2693A1SAM1?state=8564"uri="http://localhost
      /api/v2/artifacts/LIT2693A1SAM1?state=8562" limsid="LIT2693A1SAM1"/>
    <output uri="http://localhost/api/v2/artifacts/LIT2693A1R010?state=8563" output-
      type="Analyte" limsid="LIT2693A1R010"/>
  </input-output-map>
  <input-output-map>
    <input post-process-uri="http://localhost/api/v2/artifacts/ADM793A1PA1?state=3896"
      uri="http://localhost/api/v2/artifacts/ADM793A1PA1?state=3893" limsid="ADM793A1PA1"/>
    <output uri="http://localhost/api/v2/artifacts/ADM793A1R02?state=3894" output-
      generation-type="PerAllInputs" output-type="ResultFile" limsid="ADM793A1R02"/>
  </input-output-map>
  <process-parameter name="TestProcessParam"/>
</prc:process>
```

If the POST is Not Successful, then the XML that is returned contains the error that occurred when the POST completed. The following example shows this error:

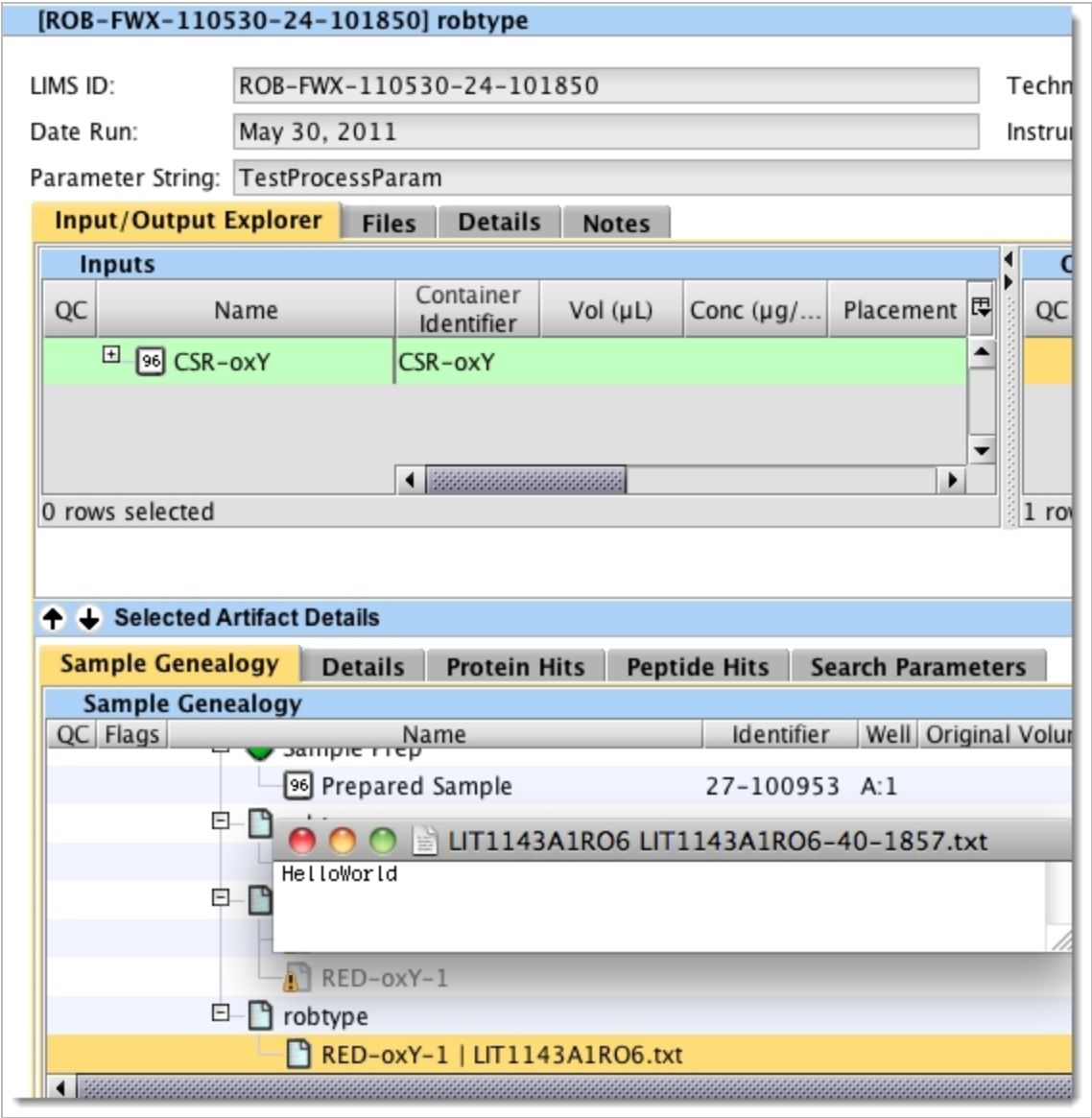
```
<exc:exception xmlns:exc="http://genologics.com/ri/exception">
  <message>The process type named 'HiSEQ POST' cannot produce the following types of shared
  outputs: 'ResultFile'.</message>
</exc:exception>
```

EPP Confirmation

The response XML contains the `<process-parameter>` tag. The `automatedInformatics.log` file will contain an entry similar to `Command string: bash -c "echo HelloWorld > LIT1143A1RO6.txt"`.

In the Operations Interface of Clarity LIMS , the process summary view shows the parameter string name.

In addition, the file generated by the EPP is also visible. The text file also has HelloWorld in it when it is opened.



Attachments

- [ExecuteProcessWithEPP.groovy.txt](#)
- [autocomplete-process.py.txt](#)

Work with Submitted Samples

When working with submitted samples, you can do the following:

- [Add Samples to the System](#)
- [Rename Samples](#)
- [Assign Samples to Workflows](#)
- [Update Sample Information](#)
- [Show the Relationship Between Samples and Analyte Artifacts \(Derived Samples\)](#)

Add Samples to the System

You can add samples to the system using API (v2 r21 and later). This example assumes that you have sample information in a file that is difficult to convert into a format suitable for importing into Clarity LIMS. The aim is to add the samples, and all associated data, into Clarity LIMS without having to translate the file manually. You can use the REST API to add the samples.

Prerequisites

Follow the instructions provided in the following examples:

- [Add a New Project to the System with UDF/Custom Field Value](#)
- [Add an Empty Container to the System](#)

Code Example

To add a sample in Clarity LIMS, you must assign it to a project and place it into a container. This example assumes that you are adding a new project and container for the samples being created.

Step 1: Create a New Project

As shown in [Add a New Project to the System with UDF/Custom Field Value](#), you define a project by using `StreamingMarkupBuilder`. `StreamingMarkupBuilder` is a built-in Groovy data structure designed to build XML structures. This structure creates the XML that is used in a POST to the projects resource:

```
def builder = new StreamingMarkupBuilder()
builder.encoding = "UTF-8"
String openDate = new SimpleDateFormat("yyyy-MM-dd").format(new Date()) // Create a new
project using the Markup Builder
def projectDoc = builder.bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(prj: 'http://genologics.com/ri/project')
    'prj:project' {
        'name' ("${projectName}")
        'open-date' ("${openDate}")
    }
}
```

```

        'researcher' (uri:researcherURI)
    }
}

// Post the new project to the API
createdProjectNode = GLSRestApiUtils.xmlStringToNode(projectDoc.toString())
createdProjectNode = GLSRestApiUtils.httpPOST(createdProjectNode, "${projectListURI}",
username, password)
println GLSRestApiUtils.nodeToXmlString(createdProjectNode)

```

If the POST to projects is successful, the following XML is returned:

```

<prj:project xmlns:prj="http://genologics.com/ri/project"
uri="http://yourIPaddress/api/v2/projects/EXA2248" limsid="EXA2248">
  <name>Cookbook Project POST 8.1</name>
  <open-date>2010-09-14</open-date> <researcher
uri="http://yourIPaddress/api/v2/researchers/1"/>
</prj:project>

```

Step 2: Create a New Container

As shown in the [Add an Empty Container to the System](#) example, you can add a container by using `StreamingMarkupBuilder` to create the XML for a new container. This creates the XML that is used in a POST to the containers resource:

```

// Determine the URI of the containers list
containersListURI = "http://${hostname}/api/v2/containers"
// Create a new container using the Markup Builder
def containerDoc = builder.bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(con: 'http://genologics.com/ri/container')
    mkp.declareNamespace(udf: 'http://genologics.com/ri/userdefined')
    'con:container' {
        'name' ("${containerName}")
        'type' (uri:"http://${hostname}/api/v2/containertypes/1", name:"96 well plate")
    }
}

// Post the new container to the API
createdContainerNode = GLSRestApiUtils.xmlStringToNode(containerDoc.toString())
println GLSRestApiUtils.nodeToXmlString(createdContainerNode)
createdContainerNode = GLSRestApiUtils.httpPOST(createdContainerNode, containersListURI,
username, password)
println GLSRestApiUtils.nodeToXmlString(createdContainerNode)

```

If the POST to containers is successful, the following XML is returned:

```
/** NOTE: if the system has unique container name constraints enabled, change this value
every time script runs*/
<con:container xmlns:con="http://genologics.com/ri/container"
uri="http://yourIPAddress/api/v2/containers/27-4075" limsid="27-4075">
  <name>Example 8.1 container</name>
  <type uri="http://yourIPAddress/api/v2/containertypes/1" name="96 well plate"/>
</con:container>
```

Step 3: Create a New Sample

Now that you have the project and container, you can use `StreamingMarkupBuilder` to create the sample. The XML created to add the sample uses the URIs for the project and container that were created in the previous steps.

```
// Determine the samples list URI
sampleListURI = "http://${hostname}/api/v2/samples"
// Create a sample using the Markup Builder
def sampleDoc = builder.bind {
  mkp.xmlDeclaration()
  mkp.declareNamespace(smp: 'http://genologics.com/ri/sample')
  'smp:samplecreation' {
    'name' ("${sampleName}")
    'project' (uri:"${createdProjectNode.'@uri'}")
    'location' {
      'container' (limsid:"${createdContainerNode.'@limsid'}",
        uri:"${createdContainerNode.'@uri'}")
      'value' ('A:1')
    }
  }
}

// Post the sample to the API
createdSampleNode = GLSRestApiUtils.xmlStringToNode(sampleDoc.toString())
createdSampleNode = GLSRestApiUtils.httpPOST(createdSampleNode, "${sampleListURI}",
username, password)
println GLSRestApiUtils.nodeToXmlString(createdSampleNode)
```

This POST to the samples resource creates a sample in Clarity LIMS, adding it to the project and container specified in the POST.

Expected Output and Results

In Clarity LIMS Projects and Samples dashboard, open the project to find the new sample in its container.

Attachments

[PostSample.groovy.txt](#)

Rename Samples

You can rename samples in the system using API (v2 r21 and later). The amount of information provided in the sample name is sometimes minimal. After the sample is in the system, you can add additional information to the name. For example, you can help lab scientists understand what they must do with a sample, or where it is in processing.

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

Code Example

Clarity LIMS displays detailed information for each sample, including its name, container, well, and date submitted.

In this example, the sample name is Colon-1. To help keep context when samples are processed by default, the submitted sample name is used for the downstream samples (or derived samples) generated by a step in Clarity LIMS.

Step 1. Retrieve the Sample

Before you rename a sample, you must first request the resource via a GET. As REST resources are self-contained entities, always request the full XML representation before editing the portion that you wish to change.

 | The XML representations of individual REST resources are self-contained entities. Always request the full XML representation before editing any portion of the XML. If you do not use the complete XML when you update the resource, you can inadvertently change data.

The following GET method returns the full XML structure for the sample:

```
// Retrieve the sample sampleURI = "http://${hostname}/api/v2/samples/${sampleLIMSID}"
sample = GLSRestApiUtils.httpGET(sampleURI, username, password)
```

The variable `sample` now holds the complete XML structure returned from the `sampleURI`.

The following example shows the XML for the sample, with the `name` element on the second line. In this particular case, the Clarity LIMS configuration has expanded the sample with 18 custom fields that provide sample information.

```
<smp:sample uri="http://yourIPaddress/api/v2/samples/HAM754A1"
limsid="HAM754A1"><name>Colon-1</name><date-received>2010-03-25</date-received><project
uri="http://yourIPaddress/api/v2/projects/HAM754" limsid="HAM754"/><artifact
uri="http://yourIPaddress/api/v2/artifacts/HAM754A1PA1?state=11822"
limsid="HAM754A1PA1"/><udf:field type="String" name="Organism">Homo
sapiens</udf:field><udf:field type="String" name="Gender">Unknown</udf:field><udf:field
type="Numeric" name="Concentration (ng/uL)">200.0</udf:field><udf:field type="Numeric"
name="Total Volume (uL)">50</udf:field><udf:field type="String" name="Investigator Sample
Name">F3_5016</udf:field><udf:field type="String" name="Sample Group Phase">Pick
Plate</udf:field><udf:field type="String" name="Date of Sample Group Phase">2010-05-
19</udf:field><udf:field type="String" name="Sample QC">FAILED</udf:field><udf:field
type="String" name="Sample QC Date">2010-04-12</udf:field><udf:field type="String"
name="Sample Picked">Yes</udf:field><udf:field type="String" name="Sample Latest Pick
Date">2010-05-19</udf:field><udf:field type="Numeric" name="Sample Picked
Iteration">3</udf:field><udf:field type="String" name="Sample Library Requeue
Status"/><udf:field type="String" name="Sample Library Requeue Date">2010-05-
19</udf:field><udf:field type="Numeric" name="Paired End Read Sequencing Lane
Iteration">2</udf:field><udf:field type="String" name="Paired End Read Sequencing Last
Date">2010-01-15</udf:field><udf:field type="String" name="Investigator Last
Name">Hershberger</udf:field><udf:field type="String" name="Cohort">Not
Applicable</udf:field></smp:sample>
```

Step 2. Rename the Sample

Renaming the sample consists of the following:

- Changing the name in the XML
- Using the PUT method to update the sample resource

The name change is executed with the nameNode XML element node, which references the XML element containing the name of the sample.

```
// Rename the sample nameNode = sample.name[0] nameNode.setValue(newName) returnNode =
GLSRestApiUtils.httpPUT(sample, sample.@uri, username, password)
```

The PUT method updates the individual sample resource using the complete XML representation, which includes the new name. Such complete updates provide a simple interaction between client and server.

Expected Output and Results

The updated sample view displays the new name. You can also view the results in a web browser via the URI at <http://<YourIPaddress>/api/v2/samples/<SampleLIMSID>>.

```
<smp:sample uri="http://yourIPaddress/api/v2/samples/HAM754A1"
limsid="HAM754A1"><name>Colon-1 Updated</name><date-received>2010-03-25</date-
received><project uri="http://yourIPaddress/api/v2/projects/HAM754"
limsid="HAM754"/><artifact
uri="http://yourIPaddress/api/v2/artifacts/HAM754A1PA1?state=11822"
limsid="HAM754A1PA1"/><udf:field type="String" name="Organism">Homo
sapiens</udf:field><udf:field type="String" name="Gender">Unknown</udf:field><udf:field
type="Numeric" name="Concentration (ng/uL)">200.0</udf:field><udf:field type="Numeric"
name="Total Volume (uL)">50</udf:field><udf:field type="String" name="Investigator Sample
Name">F3_5016</udf:field><udf:field type="String" name="Sample Group Phase">Pick
Plate</udf:field><udf:field type="String" name="Date of Sample Group Phase">2010-05-
19</udf:field><udf:field type="String" name="Sample QC">FAILED</udf:field><udf:field
type="String" name="Sample QC Date">2010-04-12</udf:field><udf:field type="String"
name="Sample Picked">Yes</udf:field><udf:field type="String" name="Sample Latest Pick
Date">2010-05-19</udf:field><udf:field type="Numeric" name="Sample Picked
Iteration">3</udf:field><udf:field type="String" name="Sample Library Requeue
Status"/><udf:field type="String" name="Sample Library Requeue Date">2010-05-
19</udf:field><udf:field type="Numeric" name="Paired End Read Sequencing Lane
Iteration">2</udf:field><udf:field type="String" name="Paired End Read Sequencing Last
Date">2010-01-15</udf:field><udf:field type="String" name="Investigator Last
Name">Hershberger</udf:field><udf:field type="String" name="Cohort">Not
Applicable</udf:field></smp:sample>
```

Attachments

[RenamingSample.groovy.txt](#)

Assign Samples to Workflows

You can use the API (v2 r21 and later) to automate the process of assigning samples to a workflow. This example shows how to create the required XML. The example also provides a brief introduction on how to use the route/artifacts endpoint, which is the endpoint used to perform the sample assignment.

The example takes two samples that exist in Clarity LIMS and assigns each of them to a different workflow.

Code Example

Use the following steps to assign two samples to two different workflows.

Step 1: Define Assignment Endpoint URI

Define the assignment endpoint URI using the following example. The assignment endpoint allows you to assign the artifacts to the desired workflow.

```
// Define the assignment endpoint URI
assignmentOrderURI = "http://${hostname}/api/v2/route/artifacts/"
```


Step 2: Retrieve the Base Artifact URIs of the Samples

You can also retrieve the base artifact URIs of the samples using the following example:

```
// Retrieve the samples' base artifact URIs
sampleAArtifactURI = GLSRestApiUtils.httpGET(sampleURIs[0], username,
password).`artifact`.@uri
sampleBArtifactURI = GLSRestApiUtils.httpGET(sampleURIs[1], username,
password).`artifact`.@uri
```

Step 3: Gather Workflow URIs

Use the following example to gather the workflow URIs:

```
// Gather the required workflow URIs
workflowAURI = workflowURIs[0]
workflowBURI = workflowURIs[1]
```

Step 4: Construct and POST the XML

Next, you can construct the XML that is posted to perform the workflow assignment. You can do this construction by using the `StreamingMarkupBuilder` and the following example.

Assign the analyte (derived sample) artifact of the sample to a workflow as follows.

1. Create an assign tag with the URI of the destination workflow as an attribute.
2. Create an artifact tag inside the assign tag with the URI of the analyte as an attribute.
3. After the assignment XML is defined, you can POST it to the API. This POST performs the sample assignment.

```
// Create a new routing assignment using the Markup Builder
def assignmentOrder = builder.bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(rt: 'http://genologics.com/ri/routing')
    'rt:routing' {
        'assign'('workflow-uri': workflowAURI) {
            'artifact'('uri': sampleAArtifactURI)
        }
        'assign'('workflow-uri': workflowBURI) {
            'artifact'('uri': sampleBArtifactURI)
        }
    }
}

// Post the commands to the API
assignmentOrderNode = GLSRestApiUtils.xmlStringToNode(assignmentOrder.toString())
```

```
assignmentOrderNode = GLSRestApiUtils.httpPOST(assignmentOrderNode, assignmentOrderURI,
username, password)
println GLSRestApiUtils.nodeToXmlString(assignmentOrderNode)
```

Expected Output and Results

After the script has run, the samples display in the first step of the first protocol in the specified workflows.

Attachments

[AssigningArtifactsToWorkflows.groovy.txt](#)

Update Sample Information

The most important information about a sample is often recorded in custom fields in API (v2 r21 and later). These fields often contain information that is critical to the processing of the sample, such as species or sample type.

When samples come into the lab, you can provide lab scientists with information about priority or quality. You can provide this information by changing the value of specific sample custom fields.

This example shows how to change the value of a sample custom field called Priority after you have entered a submitted sample into the system.

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

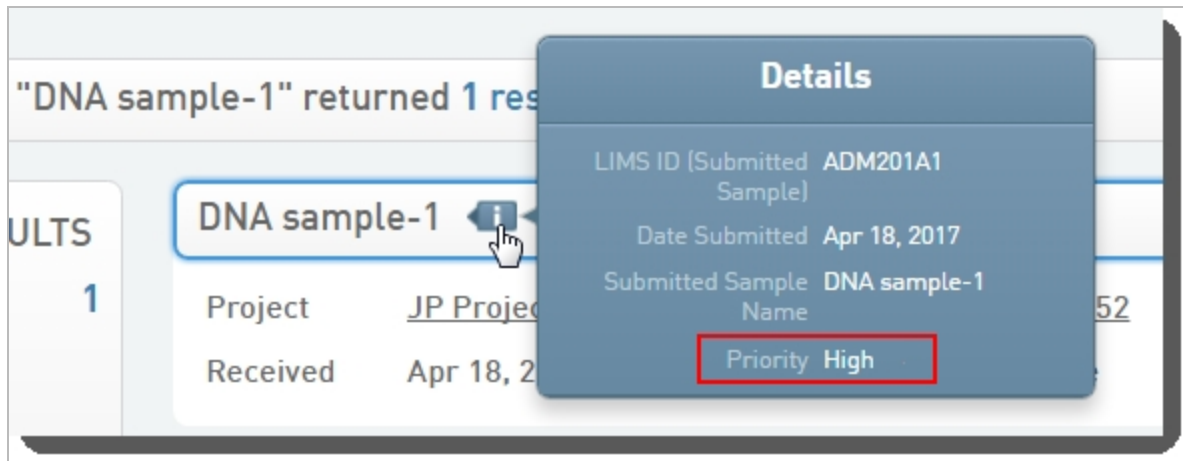
Code Example

In Clarity LIMS, you can display detailed information for a sample, including the following:

- Name
- Clarity LIMS ID
- Custom fields

In the following figure, you can see that the sample name is DNA Sample-1 and the field named Priority has the value High.

In this example, change the value of the Priority custom field to Critical.



Step 1: Retrieve the Resource

Before you can change the value of the field, you must first request the resource via a GET method.

To change a sample submitted in Clarity LIMS, use the individual sample resource. The XML returned from a GET on the individual sample resource contains the information about the sample.

The following GET method returns the full XML structure for the sample:

```
// Retrieve the sample
sampleURI = "http://{hostname}/api/v2/samples/${sampleID}"
sample = GLSRestApiUtils.httpGET(sampleURI, username, password)
```

The sample variable now holds the complete XML structure returned from the sample GET request.

- ⚠ The XML representations of individual REST resources are self-contained entities. Always request the complete XML representation before editing any portion of the XML. If you do not use the complete XML when you update the resource, you can inadvertently change data.

The following shows XML returned for the sample, with the Priority field shown in red in the second to last line. In this example:

- The Clarity LIMS configuration has added three fields to the expanded sample information.
- The custom fields are named Sample Type, PhenotypicInformation, and Priority .

```
<smp:sample uri="http://yourIPaddress/api/v2/samples/ADM201A1" limsid="ADM201A1">
  <name>DNA sample-1</name>
  <date-received>2017-04-18</date-received>
  <project uri="http://yourIPaddress/api/v2/projects/ADM201" limsid="ADM201"/>
  <artifact uri="http://yourIPaddress/api/v2/artifacts/ADM201A1PA1?state=2995"
    limsid="ADM201A1PA1"/>
  <udf:field type="String" name="Sample Type">DNA</udf:field>
  <udf:field type="String" name="Phenotypic Information">Human</udf:field>
  <udf:field type="String" name="Priority">High</udf:field>
```

```
</smp:sample>
```

Step 2: Update the Sample Custom Field

When updating the Priority field, you need to do the following:

- Change the value in the XML.
- Use a PUT method to update the sample resource.

You can change the value for Priority to Critical by using the utility files `setUdfValue` method.

The subsequent PUT method updates the sample resource at the specified URI using the complete XML representation, which includes the new custom field value for XML.

```
// Update the sample's Priority UDF value
sample = GLSRestApiUtils.setUdfValue(sample, 'Priority', 'Critical')
returnNode = GLSRestApiUtils.httpPUT(sample, sample.@uri, username, password)
println GLSRestApiUtils.nodeToXmlString(returnNode)
```

A successful PUT returns the new XML in the `returnNode`. The results can also be reviewed in a web browser at `<YourIPAddress>/api/v2/samples/<SampleLIMSID>` URI.

An unsuccessful PUT returns the HTTP response code and message in the `returnNode` XML .

i | The values for the other two fields, Sample Type and Phenotypic Information, did not change. These values did not change because they were included in the XML used in the PUT (eg, they were held in the sample variable as part of the complete XML structure).

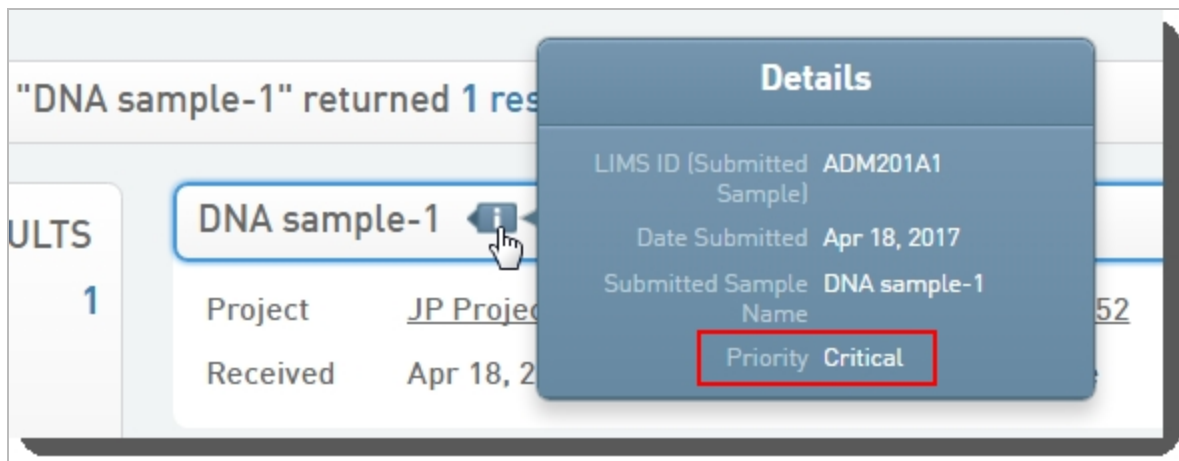
If those custom fields had not been included in the XML, they would have been updated to have no value.

Expected Output and Results

The following XML from our example shows the expected output:

```
<smp:sample uri="http://yourIPAddress/api/v2/samples/ADM201A1" limsid="ADM201A1">
  <name>DNA sample-1</name>
  <date-received>2017-04-18</date-received> <project
uri="http://yourIPAddress/api/v2/projects/ADM201" limsid="ADM201"/>
  <artifact uri="http://yourIPAddress/api/v2/artifacts/ADM503A1PA1?state=2995"
limsid="ADM201A1PA1"/>
  <udf:field type="String" name="Sample Type">DNA</udf:field>
  <udf:field type="String" name="Phenotypic Information">Human</udf:field>
  <udf:field type="String" name="Priority">Critical</udf:field>
</smp:sample>
```

In Clarity LIMS, the updated sample details now show the new Priority value.



Attachments

[UpdateSampleUDF.groovy.txt](#)

Show the Relationship Between Samples and Analyte Artifacts (Derived Samples)

In the Clarity LIMS API (v2 r21 or later), the initial submitted sample is referred to as a sample (or root artifact). Any derived sample output from a process/step is referred to as an analyte, or artifact of type analyte. This example demonstrates the relationship between samples and analyte artifacts. You must have a sample in the system and one or more processes/steps done that output analyte (derived sample) artifacts.

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

Code Example

The code example does the following when it is used:

- Retrieves the URI of an arbitrary analyte artifact
- Retrieves the corresponding sample of the artifact
- Retrieves the original root analyte artifact from the sample, as shown in the following example:

```
// Determine the analyte's URI and retrieve it
```

```

arbitraryAnalyteArtifactURI = "http://${hostname}/api/v2/artifacts/${artifactLIMSID}"
arbitraryArtifactNode = GLSRestApiUtils.httpGET(arbitraryAnalyteArtifactURI, username,
password)
println GLSRestApiUtils.nodeToXmlString(arbitraryArtifactNode)

// Retrieve the analyte's sample
def rootSampleURI = arbitraryArtifactNode.'sample'[0]. '@uri'
rootSampleNode = GLSRestApiUtils.httpGET(rootSampleURI, username, password)
println GLSRestApiUtils.nodeToXmlString(rootSampleNode)

// Retrieve the sample's root analyte
def rootAnalyteArtifactURI = rootSampleNode.'artifact'[0]. '@uri'
rootAnalyteArtifactNode = GLSRestApiUtils.httpGET(rootAnalyteArtifactURI, username,
password)
println GLSRestApiUtils.nodeToXmlString(rootAnalyteArtifactNode)

```

Expected Output and Results

You can generate XML for an arbitrary analyte artifact. The analyte artifact is downstream and has a parent-process element (as shown in line 5). The sample artifact is an original artifact. Downstream artifacts relate to at least one sample, but can also relate to more than one sample, like with pooling or shared result files. The following is an example of XML generated for an analyte artifact:

```

<art:artifact xmlns:art="http://genomics.com/ri/artifact"
uri="http://yourIPaddress/api/v2/artifacts/EXA2241A1DN3?state=18882" limsid="EXA2241A1DN3">
  <name>Example Analyte Artifact</name>
  <type>Analyte</type>
  <output-type>Analyte</output-type>
  <parent-process uri="http://yourIPaddress/api/v2/processes/DNA-SA1-100914-24-1917"
limsid="DNA-SA1-100914-24-1917"/>
  <qc-flag>UNKNOWN</qc-flag>
  <location>
    <container uri="http://yourIPaddress/api/v2/containers/27-3543" limsid="27-3543"/>
    <value>A:2</value>
  </location>
  <working-flag>false</working-flag>
  <sample uri="http://yourIPaddress0/api/v2/samples/EXA2241A1" limsid="EXA2241A1"/>
</art:artifact>

```

You can also generate XML for a submitted sample. Every submitted sample has exactly one corresponding original root artifact. A sample representation does not link to downstream artifacts, but you can find them using query parameters in the artifacts list resource. The following is an example of XML generated for a submitted sample:

```
<smp:sample xmlns:smp="http://genologics.com/ri/sample"
uri="http://yourIPAddress/api/v2/samples/EXA2241A1" limsid="EXA2241A1">
  <name>Example Sample</name>
  <project uri="http://yourIPAddress/api/v2/projects/EXA2241" limsid="EXA2241"/>
  <artifact uri="http://yourIPAddress/api/v2/artifacts/EXA2241A1SAM1?state=18879"
limsid="EXA2241A1SAM1"/>
  <biosource/>
  <udf:field xmlns:udf="http://genologics.com/ri/userdefined" type="String" name="String
Type UDF">Updated String UDF Value</udf:field>
  <udf:field xmlns:udf="http://genologics.com/ri/userdefined" type="Text" name="Text Type
UDF">Updated Text UDF Value</udf:field>
  <udf:field xmlns:udf="http://genologics.com/ri/userdefined" unit="u" type="Numeric"
name="Numeric Type UDF">3.140</udf:field>
  <udf:field xmlns:udf="http://genologics.com/ri/userdefined" type="Boolean" name="Boolean
Type UDF">true</udf:field>
  <udf:field xmlns:udf="http://genologics.com/ri/userdefined" type="Date" name="Date Type
UDF">2010-09-14</udf:field>
  <udf:field xmlns:udf="http://genologics.com/ri/userdefined" type="URI" name="URI Type
UDF">http://www.genologics.com</udf:field>
  <udf:field xmlns:udf="http://genologics.com/ri/userdefined" type="Executable"
name="Executable Type UDF">/usr/bin/yes</udf:field>
</smp:sample>
```

Lastly, you can generate XML for an original sample artifact called a root artifact. The following is an example of XML generated from an original sample artifact. In this case, both the downstream artifact and the original root artifact point to the same original sample (eg, LIMS ID EXA2241A1).

```
<art:artifact xmlns:art="http://genologics.com/ri/artifact"
uri="http://yourIPAddress/api/v2/artifacts/EXA2241A1SAM1?state=18879"
limsid="EXA2241A1SAM1">
  <name>Example Analyte Artifact</name>
  <type>Analyte</type>
  <output-type>Analyte</output-type>
  <qc-flag>UNKNOWN</qc-flag>
  <location>
    <container uri="http://yourIPAddress/api/v2/containers/27-3543" limsid="27-3543"/>
    <value>A:1</value>
  </location>
```

```
<working-flag>true</working-flag> <sample  
uri="http://yourIPaddress/api/v2/samples/EXA2241A1" limsid="EXA2241A1"/>  
</art:artifact>
```

Attachments

[SampleAndAnalyteRelations.groovy.txt](#)

Work with Containers

When working with containers, you can do the following:

- [Add an Empty Container to the System](#)
- [Find the Contents of a Well Location in a Container](#)
- [Filter Containers by Name](#)

Add an Empty Container to the System

When a lab processes samples, the samples are always in a container of some sort (eg, a tube, a 96-well plate, or a flow cell). In Clarity LIMS, this processing is modeled by placing all samples into containers. Because the Clarity LIMS interface relies on container placement for the display of many of its screens, adding containers is a critical step when running a process or adding samples through the API (v2 r21 or later).

The following example demonstrates how to add an empty container, of a predefined container type, to Clarity LIMS through the API.

 If you would like to add a batch of containers to the system, you can increase the script execution speed by using batch operations. For more information, refer to the [API Portal](#) and the articles in the [Batch Resources Overview](#) section.

Code Example

The following actions must be done when adding an empty container to the system:

- Define the new container.
- Post the container.

Step 1: Define the New Container

Before you can add a container to the system, you must first define the container to be created. You can construct the XML that defines the container using `StreamingMarkupBuilder`, a built-in Groovy data structure designed to build XML structures.

To construct the XML, you must declare the container namespace because you are building a container. The minimum information that can be specified to create a container are the `container name` and `container type`.

If you also want to add custom field values to the container you are creating, you must declare the `userdefined namespace`.



As of Clarity LIMS v5, the term **user-defined field (UDF)** has been replaced with **custom field** in the user interface. However, the API resource is still called **udf**.

```
// Determine the containers list URI
String containersListURI = "http://${hostname}/api/v2/containers"
def builder = new StreamingMarkupBuilder()
builder.encoding = "UTF-8"

// Create a new container using the Markup Builder
def containerDoc = builder.bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(con: 'http://genologics.com/ri/container')
    mkp.declareNamespace(udf: 'http://genologics.com/ri/userdefined')
    'con:container'{
        'name' (containerName)
        'type' (uri:"http://${hostname}/api/v2/containertypes/1", name:"96 well plate")
    }
}
```

Step 2: Post the New Container

The POST command posts the XML constructed by `StreamingMarkupBuilder` to the containers resource of the API. The POST command also adds a link from the containers URI (the list of containers) to the new container.

```
// Post the new container to the API
def containerNode = GLSRestApiUtils.xmlStringToNode(containerDoc.toString())
containerNode = GLSRestApiUtils.httpPOST(containerNode, containersListURI, username,
password)
println GLSRestApiUtils.nodeToXmlString(containerNode)
```

Expected Output and Results

The XML for the new container is as follows.

```
<con:container xmlns:con="http://genologics.com/ri/container">
  <name>containerPOSTScript</name>
  <type uri="http://yourIPAddress/api/v2/containertypes/1" name="96 well plate"/>
</con:container>
```

The XML for the list of containers, with the newly added container shown at the end of the list, is as follows.

```
<con:containers>
<container uri="http://yourIPAddress/api/v2/containers/27-1858" limsid="27-1858">
  <name>27-1858</name>
```

```

</container>
<container uri="http://yourIPaddress/api/v2/containers/27-2182" limsid="27-2182">
  <name>27-2182-1</name>
</container>
<container uri="http://yourIPaddress/api/v2/containers/27-2202" limsid="27-2202">
  <name>testingContainerPOST</name>
</container>
</con:containers>

```

For Clarity LIMS v4.2 and earlier, in the Operations Interface, the updated Containers view displays the new container.

QC	Icon	Name	Container Identifier	Vol (µL)	Conc (µg/...	Wells Avail.	
	96	27-1374	27-1374			96	
	96	27-1375	27-1375			96	
	96	27-1376	27-1376			96	
	96	27-1377	27-1377			96	
	96	27-1378	27-1378			96	
	96	27-1379	27-1379			96	
	96	27-1380	27-1380			96	
	96	27-1381	27-1381			96	
	96	testingContainerP...	testingCon...			96	

For Clarity LIMS v5 and above, the Operations Interface Java client has been deprecated, and there is no equivalent Containers view screen in which to view empty containers added via the API. However, if you intend to add samples to Clarity LIMS through the API, this example is still relevant, as you must first add containers in which to place those samples.

Attachments

[PostContainer.groovy.txt](#)

Find the Contents of a Well Location in a Container

As samples are processed in the lab, they are kept in a container. Some of these containers hold multiple samples, and lab scientists often must switch between container tracking and sample tracking.

If you process several containers each day and track them in a list, you would need to find which samples are in those containers. This way, you can record specifics from these container-based activities in relation to the samples from Clarity LIMS.

The example finds which sample is in a given well of a multi-well container using Clarity LIMS and API (v2 r21 or later).

Prerequisites

Before you follow the example, make sure that you have the following items:

- Several samples that are in Clarity LIMS.
- A step that has been run on the samples.
- The outputs of the step that have been placed in a 96-well plate.

Code Example

Clarity LIMS captures detailed information for a container (eg, its name, LIMS ID, and the names of the sample in each of its wells). Information about the container and what it currently contains is available in the individual XML resource for the container.

The individual container resource contains a placement element for each sample placed on the container. Each placement element has a child element named value that describes one position on the container (eg, the placement elements for a 96-well plate include A:1, B:5, E:2).

Step 1: Retrieve the Container Information

In the script, the GET request retrieves the container specified by the container LIMS ID provided as input to the {containerLIMSID} parameter. The XML representation returned from the API is stored as the value of the container variable:

```
// Determine the specified container's URI and retrieve it
containerURI = "http://${hostname}/api/v2/containers/${containerLIMSID}"
container = GLSRestApiUtils.httpGET(containerURI, username, password)
```

The following example shows the XML format returned for a container. The XML includes a placement element for each artifact that is placed in a well location in the container.

```
<con:container uri="http://yourIPAddress/api/v2/containers/27-1259" limsid="27-1259">
  <name>PC0006</name>
  <type uri="http://yourIPAddress/api/v2/containerTypes/1" name="96 well plate"/>
  <occupied-wells>96</occupied-wells>
  <placement uri="http://yourIPAddress/api/v2/artifacts/HAM751A495PA1"
    limsid="HAM751A495PA1">
    <value>B:3</value>
  </placement>
  <placement uri="http://yourIPAddress/api/v2/artifacts/HAM751A496PA1"
    limsid="HAM751A496PA1">
    <value>B:4</value>
  </placement>
  <placement uri="http://yourIPAddress/api/v2/artifacts/HAM751A493PA1"
    limsid="HAM751A493PA1">
    <value>B:1</value>
```

```

</placement>
<placement uri="http://yourIPaddress/api/v2/artifacts/HAM751A497PA1"
lmsid="HAM751A497PA1">
  <value>B:5</value>
</placement>
</con:container>

```

Step 2: Find the Artifact URI

When you look for the artifact at the target location, the script searches through the placement elements for one with a value element that matches the target. If a match is found, it is stored as the value of the contents variable.

The `>uri` attribute of the matching placement element is the URI of the artifact that is in the target well location. This is stored as the value of the `artifactURI` variable, and printed as the output of the script:

```

// Print the artifact that is located at the specified placement
contents = container.placement.find { it.value.text() == targetPlacement }
artifactURI = contents?.@uri
println artifactURI

```

Expected Output and Results

Running the script in a console produces the artifact at

`http://yourIPaddress/api/v2/artifacts/HAM751A496PA1.`

Attachments

[GetContentsOfWellLocation.groovy](#)

Filter Containers by Name

Samples in the lab are always in a container (eg, a tube, plate, or flow cell). When a container holds more than one sample, it is often easier to track the container rather than the individual samples. These containers can be found in API (v2 r21 or later).

In Clarity LIMS, containers are identified the LIMS ID or by name. The best way to find a container in the API is with the LIMS ID. However, the API also supports searching for containers by name by using a filter.

- **LIMS ID**—This is a unique ID. The container resource with LIMS ID 27-42 can be found at `http://<YourIPaddress>/api/containers/27-42.`
- **Name**—Container names can be unique, depending on how the server software was set up. In some labs, container names are reused to show when a container is recycled or when samples are submitted in containers.

The following example shows a container list filtered by name. Your system contains a series of containers, named with a specific naming convention.

Code Example

In this example, the queried containers are named Smith553 and 001TGZ.

The request for a container with a specific name is structured in the same way as the request for all containers, but also includes a parameter to filter by name:

```
http://<YourIPAddress>/api/<apiversion>containers?name= <yourcontainername>
```

The name parameter is repeatable, and the results returned match any of the names queried:

```
// Determine the containers URIs and retrieve them
containersURI = "http://${hostname}/api/v2/containers?name=" + URLEncoder.encode('Smith553')
+ "andname=" + URLEncoder.encode('001TGZ')
containers = GLSRestApiUtils.httpGET(containersURI, username, password)
// For each container, print its limsid
containers.'container'.each {
    println it.@limsid
}
```

The GET method returns the full XML structure for the list of containers matching the query. In this case, the method returns the XML structure for containers with the names Smith553 and 001TGZ.

The XML contains a list of container elements. The `.each` method goes through each container node in the list and prints the container LIMS ID.

The XML returned is placed in the variable `containers`:

```
<con:containers>
  <container uri="http://yourIPAddress/api/v2/containers/27-505" limsid="27-505">
    <name>Smith553</name> </container>
  <container uri="http://yourIPAddress/api/v2/containers/27-511" limsid="27-511">
    <name>001TGZ</name>
  </container>
</con:containers>
```

If the system has no containers named Smith553 or 001TGZ, then `containers.container` is an empty list. The `.each` method does nothing, as expected.

Expected Output and Results

When execution completes, the code returns the list of LIMS IDs associated with the container names Smith553 and 001TGZ. The name and LIMS IDs are different in this case (eg, 27-505 27-511).

Attachments

[GetContainerNameFilter.groovy.txt](#)

Work with Derived Sample Automations

When working with containers, you can do the following:

- [Remove Samples from Workflows](#)

- [Requeue Samples](#)
- [Rearray Samples](#)

Remove Samples from Workflows

In Clarity LIMS, derived sample automations are automations that users can run on derived samples directly from the Projects Dashboard.

The following example uses an automation to initiate a script that removes multiple derived samples from workflows. The example also describes the main functions included in the script, and shows how to configure the automation in Clarity LIMS and run it from the Projects Dashboard.

For more information on creating automations, refer to the following documents:

- For Clarity LIMS v4.1 or v4.2, refer to the *Samples* section of *Adding Automation Actions for Derived Samples*.
- For Clarity LIMS v5 and later, refer to the *Clarity LIMS Core Functionality* documentation.

Prerequisites

Before removing samples from the workflows, make sure you have the following items:

- A project containing at least one sample assigned to a workflow.
- A step has been run on a sample, resulting in a derived sample.
- The derived sample is associated with one or more workflows.

Code Example

The attached `UnassignSamplesFromWorkflows.groovy` script uses the derived sample automations feature to remove selected derived samples from their associated workflows. The following actions must be done when removing samples from these workflows.

Step 1: Create Sample Node List

This `getSampleNodes` function is passed a list of derived sample LIMS IDs (as a command-line argument) to build a list containing the XML representations of the samples. A for-each loop on the derived sample list makes a GET call for each sample and creates the sample node list. The following command-line example shows how the `getSampleNodes` function works:

```
def getSampleNodes(sampleList, hostname, username, password) {
    sampleURLList = sampleList.collect{"${hostname}v2/artifacts/${it}"}
    return sampleNodeList = GLSRestApiUtils.batchGET(sampleURLList, username, password)
}
```

This list is used to retrieve the sample URIs and the workflow-stage URIs. These URIs are required to build the unassignment XML.

Step 2: Retrieve Workflow URIs

A sample can be associated with one or many workflows, and each derived sample has a list of the `workflow-stages` to which it is assigned. Making a GET call on each `workflow-stage` URI retrieves its XML representation, from which the workflow URI can be acquired and added to a list. The `getWorkflowURIs` function calls for each sample node included in the list (eg, `sampleURIList`, `username`, and `password` from [Step 1: Create Sample Node List](#)).

The for loop does the following actions:

1. Makes a GET call for each `workflow-stage` to which the passed sample is assigned.
2. Retrieves the associated workflow URIs.
3. Returns a list containing all URIs for the workflows with which the sample is associated.

```
def getWorkflowURIs(sampleNode, username, password){
  return sampleNode.'workflow-stages'?.'workflow-stage'.collect\
    {GLSRestApiUtils.httpGET(it.@uri, username, password).workflow.@uri}
}
```

Step 3: Build and POST the Unassignment XML

Now that the functions used to retrieve both the derived sample URIs and the workflow URIs have been built, you can use `StreamingMarkupBuilder` to create the XML and then POST to the unassignment URI. This process can be done with the `unassignSamplesFromWorkflows` and `unassignSamplesXML` functions.

To unassign the derived samples, you can POST to the artifacts URI at `${hostname}v2/route/artifacts`. Nested loops create the declaration for each sample and their associated workflows. The following example shows the declaration built in the format of the workflow URI, with the unassign flag followed by the URI of the sample being unassigned.

```
def unassignSamplesFromWorkflows(sampleNodeList, username, password) {
  def builder = new StreamingMarkupBuilder()
  builder.encoding = "UTF-8"
  def unassignSamplesXML = builder.bind{
    mkp.xmlDeclaration()
    mkp.declareNamespace(rt: 'http://genologics.com/ri/routing')
    'rt:routing' {
      //Dynamically build XML using a for loop for each sampleNode
      sampleNodeList.each{ sampleNode ->
        //Generates the URI list for each sample
        workflowUriList = getWorkflowURIs(sampleNode, username, password)
        workflowUriList?.each{
          'unassign'('workflow-uri': it){
            'artifact'(uri: sampleNode.@uri)
          }
        }
      }
    }
  }
}
```

```

    }
  }
}
}
return GLSRestApiUtils.xmlStringToNode(unassignSamplesXML.toString())
}

```

Now that the XML is built, convert the XML to a node and post it as follows.

1. Use `GLSRestApiUtils` to convert the XML to a node.

2. POST the node using the following command:

```

unassignSampleNode = unassignSamplesFromWorkflows(sampleNodeList, username, password)
//Try the post
try {
    unassignSampleNode = GLSRestApiUtils.httpPOST(unassignSampleNode, unassignmentURI,
    username, password)
}

```

Configure and Run the Automation

Automations can be configured and run using Clarity LIMS (v4.x and earlier to v5 and later).

For Clarity LIMS v4.2 or earlier, add and configure the automation as follows.

1. In the Clarity LIMS Web Interface, select **Configure**.
2. Select **Automation**.
3. Select **Add Action** and enter the following information:
 - **Action Name**—This is the name that displays to the user running the automation from the Projects Dashboard. Choose a descriptive name that reflects the functionality/purpose (eg, Remove From Workflows).
 - **Channel Name**—Enter the channel name. For more information, refer to [Automation Channels](#).
 - **Command Line**—Enter the command line required to invoke the script.

Action Name	Remove From Workflo... 	Channel Name	limsserver
Remove From Workflows			
Command Line			
<pre>bash -c "\${groovy_bin_location}/groovy \${script_location}/UnassignSamplesFromWorkflows.groovy -h {baseURI} -u {username} -p {password} -s {derivedSampleLuids}"</pre>			

4. Select **Save Action**.

For Clarity LIMS v5 or later, add and configure the automation as follows.

1. In Clarity LIMS, under Configuration, select the **Automation** tab.
2. Select the **Derived Sample Automation** tab.
3. Select **New Automation** and enter the following information:
 - **Automation Name**—This is the name that displays to the user running the automation from the Projects Dashboard. Choose a descriptive name that reflects the functionality/purpose (eg, Remove from Workflows).
 - **Channel Name**—Enter the channel name. For more information, refer to [Automation Channels](#).
 - **Command Line**—Enter the command line required to invoke the script.

Derived Sample Automation

AUTOMATION DETAILS

NEW AUTOMATION +

Delete Cancel Save

Remove From Workflow

Channel Name

Command Line

```
bash -c "${groovy_bin_location}/groovy
${script_location}/UnassignSamplesFromWorkflows.groovy -h {baseURI} -u {username} -p
{password} -s {derivedSampleLuids}"
```

4. Select **Save**.

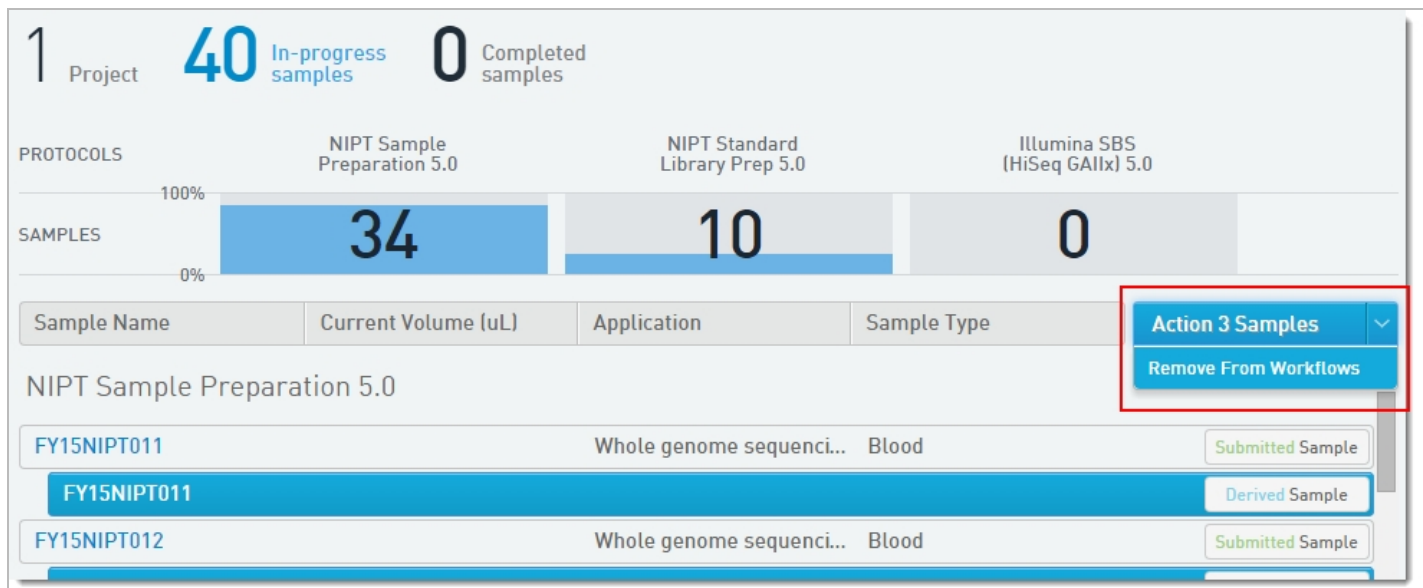
Run the automation as follows.

i | These steps are applicable to Clarity LIMS v4.x or earlier and v5 or later.

1. Open the Projects Dashboard.
2. Select a project containing in-progress samples. Select **In-progress samples**.
In the sample list, you see the submitted and derived samples that are currently in progress for this project.



3. Select one or more derived samples.
Selecting samples activates the Action button and drop-down list.
4. In the Action drop-down list, select the **Remove From Workflows** automation created in the previous step.



Expected Output and Results

The API of the selected samples now shows an additional workflow stage with a status of REMOVED.

```
<workflow-stages>
  <workflow-stage status="QUEUED" name="NIPT sample receiving 5.0" uri="http://192.168.8.10:8080/api/v2/configuration/workflows/161/stages/1057"/>
  <workflow-stage status="REMOVED" name="NIPT sample receiving 5.0" uri="http://192.168.8.10:8080/api/v2/configuration/workflows/161/stages/1057"/>
</workflow-stages>
```

Attachments

[UnassignSamplesFromWorkflows.groovy.txt](#)

Requeue Samples

Derived sample automations are automations that users can run on derived samples directly from the Projects Dashboard in Clarity LIMS.

The following example uses an automation to initiate a script that requeues samples to an earlier step in the workflow. The example also describes the main functions included in the script and demonstrates the configuration options that prompt the user for input. These options allow for greater flexibility during script runs. Before you follow the example, make sure that you have the following items:

- A project containing samples assigned to a multi-stage workflow.
- Samples that must be requeued. These samples must have completed at least one step in the workflow and must be available for requeue.

For more on creating automations in Clarity LIMS, refer to the following documents:

- For Clarity LIMS v4.1 or v4.2, refer to the *Samples* section of *Adding Automation Actions for Derived Samples*.
- For Clarity LIMS v5 and later, refer to the *Clarity LIMS Core Functionality* documentation.

Code Example

The purpose of the attached `RequeueSamples.groovy` script is to requeue selected derived samples to a previous step in the workflow with the derived sample automations feature.

Step 1: Get Sample Nodes

The `getSampleNodes` function is passed a list of derived sample LIMS IDs (as a command-line argument) to build a list containing the XML representations of the samples. The resulting sample URI list can then be used with a `batchGET` to return the sample nodes:

```
def getSampleNodes(sampleList, hostname, username, password) {  
    sampleURIList = sampleList.collect("${hostname}v2/artifacts/${it}")  
    return GLSRestApiUtils.batchGET(sampleURIList, username, password)  
}
```

Step 2: Retrieve Workflow and Stage URIs

To retrieve the workflow name, you can URL encode the workflow name and use the result to query and retrieve the workflow URI:

```
encodedWorkflowName = java.net.URLEncoder.encode(workflowName, "UTF-8")  
try {  
    workflowURI = GLSRestApiUtils.httpGET\  
        ("${hostname}v2/configuration/workflows?name=${encodedWorkflowName}", username,  
        password).workflow.@uri[0]  
    if (!workflowURI) {  
        throw new Exception()  
    }  
} catch (Exception e) {  
    println "Unable to find workflow: " + workflowName
```

```

    System.exit(-1)
}

```

The stage names are guaranteed to be unique for each workflow. However, they may not be unique in the Clarity LIMS system. As a result, the stage URI cannot be queried for in the same way as the workflow URI.

Instead, you can navigate through the workflow node to find the stage that matches the stage name specified using the `getStageURI` function. If a match is found, return the stage URI.

```

def getStageURI (workflowURI, stageName, username, password) {
  def stageURI
  GLSRestApiUtils.httpGET(workflowURI, username, password).stages.stage.find {
    if (it.@name.toString().equals(stageName)) {
      stageURI = it.@uri
      return true
    }
  }
  if (stageURI) {
    return stageURI
  }
}

```

Step 3: Verify Each Sample Meets Requeue Criteria

Next, you must make sure that each sample meets the criteria to be requeued using the `canRequeue` function. The following method checks all workflow stages for the samples:

- If a match is found between a workflow stage URI and the stage URI specified, the sample node is added to a list of samples that can be requeued using the `requeueList` function.
- If all the samples have this match and a status that allows for requeue, the list is returned. Otherwise, the script exits with an error message that states the first sample to cause failure.

```

def canRequeue(nodeList, stageURI){
  def Set conditions = ["REMOVED", "FAILED", "COMPLETE", "SKIPPED"]
  def requeueList = []
  def lastStageRun
  nodeList.each{ sampleNode ->
    sampleNode.'workflow-stages'?.'workflow-stage'.each {
      if (it.@uri.toString().equals(stageURI)) {
        lastStageRun = it
      }
    }
  }
  if (conditions.contains(lastStageRun?.@status.toString())) {
    requeueList.add(sampleNode)
  }
}

```

```

    } else {
      println "Sample: " + sampleNode.@limsid.toString() + " cannot be queued to stage " + stageName
      System.exit(-1)
    }
  }
}
return requeueList
}

```

Step 4: Check and Retrieve Workflow Stage for Sample Node

In this example, both unassignment from and assignment to a workflow stage must occur to complete the requeue. As the samples are requeuing to a previous stage in the workflow and can currently be queued for another stage, you must remove them from these queues.

The `getCurrentStageURI` and `lastStageRun` functions check the sample node for its most recent workflow stage. If the node is in a queued status, it returns that stage URI to be unassigned.

```

def getCurrentStageURI(sampleNode, workflowURI) {
  def lastStageRun
  sampleNode.'workflow-stages'?.'workflow-stage'.each {
    stageWorkflowURI = GLSRestApiUtils.httpGET(it.@uri, username,
      password).'workflow'.'@uri'[0]
    if (stageWorkflowURI.toString().equals(workflowURI.toString())) {
      lastStageRun = it
    }
  }
  return (lastStageRun.@status.toString().equals('QUEUED')) ? lastStageRun.@uri.toString() : null
}

```

Step 5: Build the XML Assignment

Using the previous methods and their results, the following code uses Streaming Markup Builder and the `assignmentXML` function to build the XML to be posted:

```

def assignmentXML = builder.bind {
  mkp.xmlDeclaration()
  mkp.declareNamespace(rt: 'http://genologics.com/ri/routing')
  'rt:routing' {
    requeueList.each { sampleNode ->
      currentURI = getCurrentStageURI(sampleNode, workflowURI)
      if (currentURI) {

```

```

        'unassign'('stage-uri': currentURI) {
            'artifact'(uri: sampleNode.@uri)
        }
    }
    'assign'('stage-uri': stageURI) {
        'artifact'(uri: sampleNode.@uri)
    }
}
}
return GLSRestApiUtils.xmlStringToNode(assignmentXML.toString())
}

```

The returned XML node is then posted using `httpPOST`.

Configure and Run the Automation

Automations can be configured using Clarity LIMS v4.1, v4.2, or v5 and later.

For Clarity LIMS v4.1 or v4.2, add and configure the automation as follows.

1. In the Clarity LIMS Web Interface, select **Configure**.
2. Select **Add Action** and enter the following information:
 - **Action Name**—This is the name that displays to the user running the automation from the Projects Dashboard. Choose a descriptive name that reflects the functionality/purpose (eg, Requeue Samples).
 - **Channel Name**—Enter the channel name. For more information, refer to [Automation Channels](#).
 - **Command Line**—Enter the command line required to invoke the script.

The screenshot shows the configuration form for an automation action. The 'Action Name' field contains 'Remove From Workflo...' with a pencil icon. The 'Channel Name' field contains 'limsserver'. Below these fields is a blue button labeled 'Remove From Workflows'. The 'Command Line' field contains the following text:

```
bash -c "${groovy_bin_location}/groovy
${script_location}/UnassignSamplesFromWorkflows.groovy -h {baseURI} -u {username} -p
{password} -s {derivedSampleLuids}"
```

i | In the example Command Line, note the `-w` and `-t {userinput}` options. The derived sample automation allows for user input as a command-line variable.

3. Select **Save Action**.

For Clarity LIMS v5 and later, add and configure the automation as follows.

1. In Clarity LIMS, under Configuration, select the **Automation** tab.

2. Select the **Derived Sample Automation** tab.
3. Select **New Automation** and enter the following information:
 - **Automation Name**—This is the name that displays to the user running the automation from the Projects Dashboard. Choose a descriptive name that reflects the functionality/purpose (eg, Requeue Samples).
 - **Channel Name**—Enter the channel name. For more information, refer to [Automation Channels](#).
 - **Command Line**—Enter the command line required to invoke the script.

Derived Sample Automation

AUTOMATION DETAILS

NEW AUTOMATION +

Delete Cancel Save

Requeue Samples

Channel Name

Command Line

```
{groovy_bin_location}/groovy{script_location}/RequeueSamples.groovy -h {baseURI} -u {username} -p {password} -w {userinput:workflowName} -t {userinput:stageName} -s {derivedSampleLuids}
```

4. Select **Save**.

Run the automation as follows.

i | These steps are applicable to Clarity LIMS v4.x or earlier and v5 or later.

1. Open the Projects Dashboard.
2. Select a project containing in-progress samples. Select **In-progress samples**.
In the sample list, you see the submitted and derived samples that are currently in progress for this project.

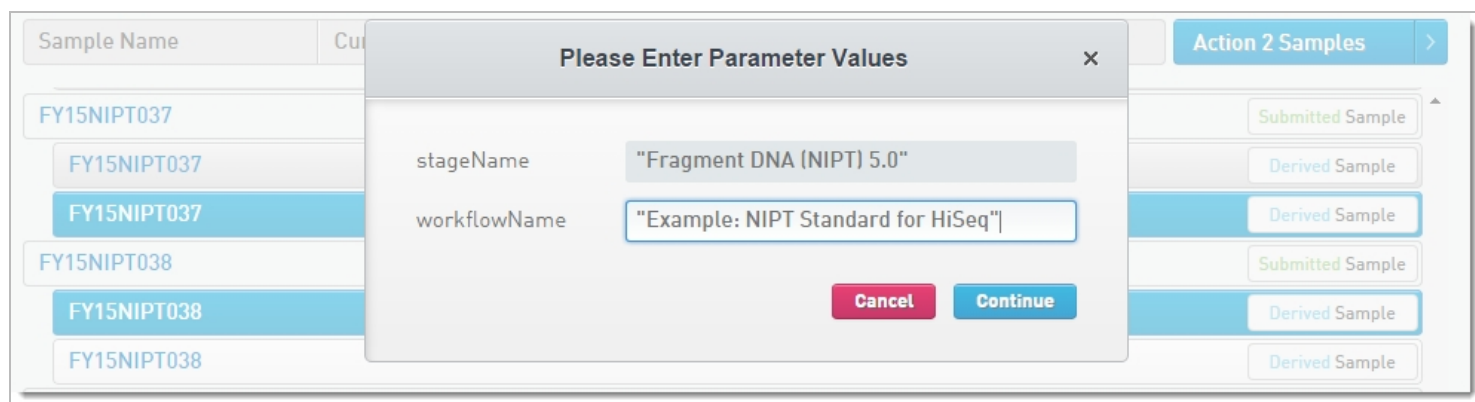


3. Select one or more derived samples.

Selecting samples activates the Action button and drop-down list.

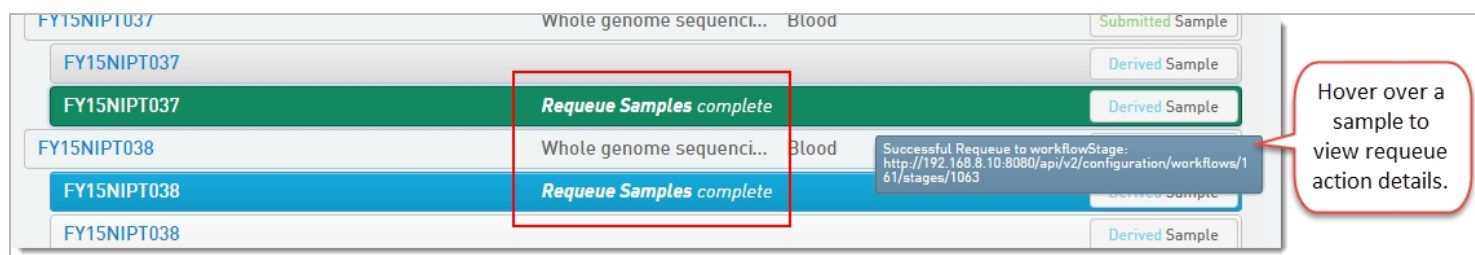
4. In the **Action** drop-down list, select the **Requeue Samples** automation.

In this example, the `-w` and `-t {userinput}` options invoke a dialog box on automation trigger. The user is required to enter two parameters: the full name of the stage and the workflow for which selected samples are to be requeued. The names must be enclosed in quotation marks.



Expected Output and Results

If the requeue is successful, each requeued sample is marked with a complete tag. Hovering over a sample shows a more detailed message.



Attachments

[RequeueSamples.groovy.txt](#)

Rearray Samples

In high throughput labs, samples are worked on in batches and some work is executed by a robot. Sometimes, a set of plates must be rearranged to one larger plate before the robot can begin the lab step.

This example accomplishes this using two scripts. One script is configured on a derived sample automation, while the second script is included in a command line that is either configured on a step automation (in Clarity LIMS v5 and later) or on a process parameter (in Clarity LIMS v4.2 and earlier).

For more on creating automations in Clarity LIMS, refer to the following documents:

- For Clarity LIMS v4.1 or v4.2, refer to the *Samples* section of *Adding Automation Actions for Derived Samples*.

- For Clarity LIMS v5 and later, refer to the *Clarity LIMS Core Functionality* documentation.

Prerequisites

Before you follow the example, make sure that you have the following items:

- A project containing samples assigned to a workflow in Clarity LIMS.
- The workflow name.
- Given samples that are assigned to the same workflow stage.

Code Example

This example demonstrates the following scripts:

- **AssignToRearrayWf.groovy**—Executed as a derived sample automation, this script assigns selected samples to the rearray step.
- **AssignToLastRemoved.groovy**—Executed after the rearray step, this script assigns the samples to the stage to which they were originally assigned. The script is included in a command line configured on a step automation (in Clarity LIMS v5 and later), and on a process parameter in (Clarity LIMS v4.2 and earlier).

Configure the Assign To Rearray Automation

Automations can be configured using Clarity LIMS v4.2 and earlier or v5 and later.

For Clarity LIMS v4.2 and earlier, configure the Assign to Rearray automation as follows.

1. In Clarity LIMS Web Interface, select **Configure**.
2. Select **Automation**.
3. Select **Add Action** and create a derived sample automation that prompts the user for the workflow stage name to be used.



In the example, note the following:

- The `{groovy_bin_location}` and `{script_location}` parameters must be customized to reflect the locations on your computer.
- The `-woption` allows for user input to be passed to the script as a command-line variable.

The screenshot shows the 'LIMS Configuration' window with the 'Automation' tab selected. The configuration details are as follows:

Field	Value
Action Name	Assign to Rearray
Channel Name	limsserver
Command Line	<code>{groovy_bin_location}/groovy {script_location}/AssignToRearrayWf.groovy -h {baseURI} -u {username} -p {password} -w "{userinput:WorkflowName}" -s {derivedSampleLUIDS}</code>

Buttons at the top right: Delete, Cancel, Save Action.

For Clarity LIMS v5 and later, configure the Assign to Rearray automation as follows.

1. In Clarity LIMS, under Configuration, select the **Automation** tab.
2. Select the **Derived Sample Automation** tab.
3. Select **New Automation** and create an automation that prompts the user for the workflow stage name to be used.

i | In the example, note the following:

- The `{groovy_bin_location}` and `{script_location}` parameters must be customized to reflect the locations on your computer.
- The `-w` option allows for user input to be passed to the script as a command-line variable.

The screenshot shows the 'NEW AUTOMATION' dialog box with the following configuration:

Field	Value
Automation Name	Assign to Rearray
Channel Name	limsserver
Command Line	<code>{groovy_bin_location}/groovy {script_location}/AssignToRearrayWf.groovy -h {baseURI} -u {username} -p {password} -w "{userinput:WorkflowName}" -s {derivedSampleLUIDS}</code>

Buttons at the top: NEW AUTOMATION +, Delete, Cancel, Save.

AssignToRearrayWf Script Methods

Step 1: Retrieve the Artifact Nodes Using the Artifact LIMS IDs

The `AssignToRearrayWf` script has a list of artifact (sample) LIMS IDs given on the command line. To begin, use this script to build a list of artifact nodes.

The following code example builds a list of artifact URIs using the artifact LIMS ID list and the `getArtifactNodes` function. The resulting artifact URI list can then be used for a `batchGET` call to return the artifact nodes.

```
def getArtifactNodes (artifactList, hostname, username, password) {  
    artifactURLList = artifactList.collect{"${hostname}v2/artifacts/${it}"}  
    return GLSRestApiUtils.batchGET (artifactURLList, username, password)  
}
```

Step 2: Retrieve Workflow Name and URI

In this example, you can assume that the workflow name is known by the user and is passed to the script by user input when the automation is initiated.

The workflow can then be queried for using the passed workflow name. The workflow name is first encoded, and from this, you can retrieve the workflow URI.

```
encodedWorkflowName = URLEncoder.encode (workflowName, "UTF-8")  
workflowURI = GLSRestApiUtils.httpGET  
("${hostname}v2/configuration/workflows?name=${encodedWorkflowName}", username,  
password)?.workflow?.getAt (0)?.@uri
```

Step 3: Make Sure that the Samples Belong to the Same Workflow Stage

For the samples to be placed in the same container, they must all belong to the same workflow and be currently queued to the same stage in that workflow.

Using the workflow name passed in by the user, do the following:

1. Search the workflow stage list of the first artifact and store the URI of the most recent stage that is part of the workflow, if it is queued. Otherwise, the script exits with an error message.
2. After storing the workflow stage URI of the first artifact, use the `checkMatch` function check against the remaining artifacts in the list to verify they are all currently queued to the same stage.

If all artifacts are queued for the stage, they are removed from the queue of the stage under the `lastWfStageURI` function.

```
def checkMatch (artifactNodeList, workflowURI, username, password) {  
    def lastWfStageURI  
    def valid = false  
    firstArtifactStageList = artifactNodeList[0].workflow-stages.workflow-stage'  
    for (i = 0; i < firstArtifactStageList.size(); i++) {  
        if (firstArtifactStageList[i].@status.toString().equals('QUEUED')) {
```

```

        stageWfURI = GLSRestApiUtils.httpGET(firstArtifactStageList[i].@uri,
        username, password)?.workflow?.@uri[0].toString()
        if (stageWfURI.equals(workflowURI.toString())) {
            valid = true
            lastWfStageURI = firstArtifactStageList[i].@uri
            if (firstArtifactStageList[i+1]?.@status.toString().equals('REMOVED')) {
                valid = false
            }
        }
    }
}

if (!valid) {
    println "artifact " + artifactNodeList[0].name.text() + " is not currently
    queued to this workflow"
    System.exit(-1)
}

for (i = 1; i < artifactNodeList.size(); i++) {
    valid = false
    artifactNodeList[i].'workflow-stages'.'workflow-stage'.each {
        if (it.@uri.toString().equals(lastWfStageURI.toString())) {
            if (it.@status.toString().equals('QUEUED')) {
                valid = true
            } else {
                valid = false
            }
        }
    }
}

if (!valid) {
    println "Artifact " + artifactNodeList[i].name.text() + " is not currently
    queued to this workflow"
    System.exit(-1)
}
}
}

```

Step 4: Build the XML for Stage Assignment Using StreamingMarkupBuilder

In this example, all the artifacts are unassigned from the previous workflow stage returned and assigned to the rearray stage using the `queuePlacementStep` function. The previous methods have verified that the artifacts in the list can be rearranged together.

```

def queuePlacementStep (artifactNodeList, unassignStageURI, workflowToAssign) {
  def builder = new StreamingMarkupBuilder()
  builder.encoding = "UTF-8"
  def placementStepAssignment = builder.bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(rt: 'http://genologics.com/ri/routing')
    'rt:routing' {
      'unassign'('stage-uri': unassignStageURI) {
        artifactNodeList.each { artifactNode ->
          'artifact'(uri: artifactNode.@uri)
        }
      }
      'assign'('workflow-uri': workflowToAssign) {
        artifactNodeList.each { artifactNode ->
          'artifact'(uri: artifactNode.@uri)
        }
      }
    }
  }
  return GLSRestApiUtils.xmlStringToNode(placementStepAssignment.toString())
}

```

The returned XML node is then posted using `httpPOST`.

Step 2: Configure the Placement Workflow, Protocol, and Step

Workflows, protocols, and steps can be configured using Clarity LIMS v4.2 and earlier or v5 and later.

For Clarity LIMS v4.2 and earlier, configure the placement of objects as follows.

1. In the Operations Interface, create a process with analyte inputs and individual per-input analyte outputs.
2. On the External Programs tab, set up a process parameter for the `AssignToLastRemoved.groovy` script.



The `{groovy_bin_location}` and `{script_location}` parameters must be customized to reflect the locations on your machine.

Name:

Abbreviation: ☐ Open the process tab after the process wizard completes

Icon:  ☐ Display in Explorer Tree ☐ Add to Button Bar

Input Types Output Types Output Details Output Generation Attributes **External Programs**

☒ Process is associated with an external program

Modify Process Parameter [X]

Channel Name:

Trigger Mode:

Name:

Command Line Call:

Process Parameter Name: **Array to last Removed Stage**

3. In the Clarity LIMS Web Interface, select **Configure**.
4. Create a protocol (eg, Array Samples to New Container).
5. Assign the protocol to the process you created in step 1.
To configure the step, hover over it and select the Edit button that displays.
6. Save the protocol.



7. On the Properties tab, set 96 well plates and 384 well plates as the permitted destination containers.

Items permitted for use in this step

Permitted Destination Containers

✕ 96 well plate ✕ 384 well plate

8. On the Automation tab, make sure that the process parameter configured in step 1 is available for selection.
9. Make the following adjustments:
 - For Behavior, select **Automatically initiated**.
 - For Stage of Step, select **at end of step**.

Place Samples in Container » Array Samples to new Container

« Back Properties **Automation** Next Steps Queue View Record Details Next »

Configure trigger timing of automation actions

Name	Behaviour	Stage of Step	Timing
Array to last Removed Stage	Automatically initi...	at end of step	

10. Create a workflow that includes the new protocol.

i | The `AssignToRearrayWf` script operates on the assumption that this is the only protocol in the workflow and that the protocol contains only the step that was configured.

For Clarity LIMS v5 and later, configure the placement of objects as follows.

1. In Clarity LIMS, under **Configuration**, select the **Lab Work** tab.
2. Create a master step of Standard step type.
3. From Configuration, select the **Automation** tab.
4. Select the **Step Automation** tab.
5. Create an automation for the `AssignToLastRemoved.groovy` script.
The `{groovy_bin_location}` and `{script_location}` parameters must be customized to reflect the locations on your computer.

AUTOMATION DETAILS

NEW AUTOMATION + Delete Cancel Save

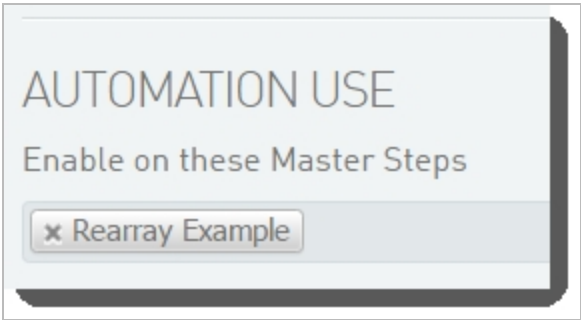
Array to Last Removed Stage

Channel Name limsserver

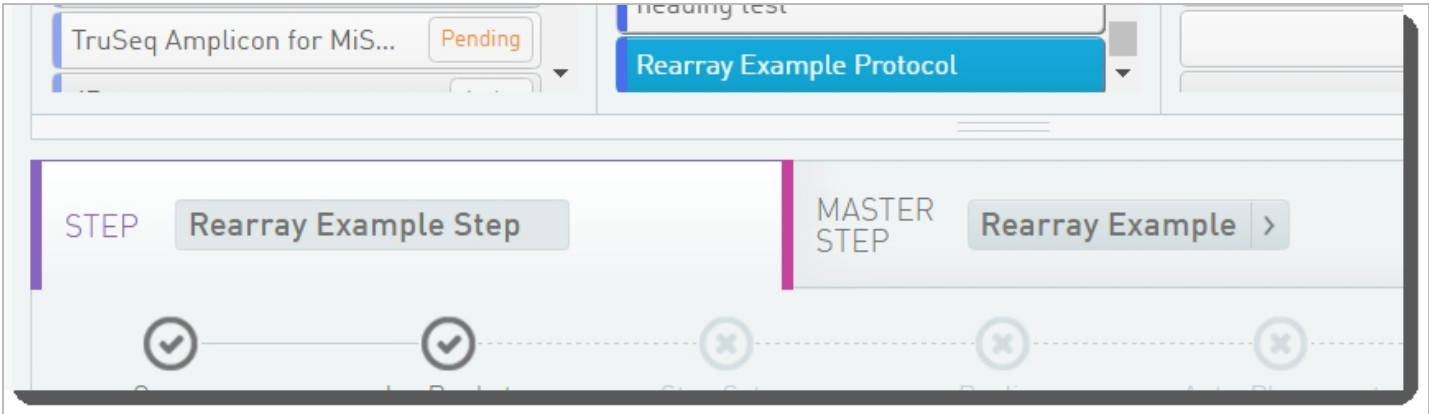
Command Line

```
bash -c "{groovy_bin_location}groovy {script_location}/AssigntoLastRemoved.groovy
-h {baseURI} -u {username} -p {password} -i {stepURI:v2:http}"
```

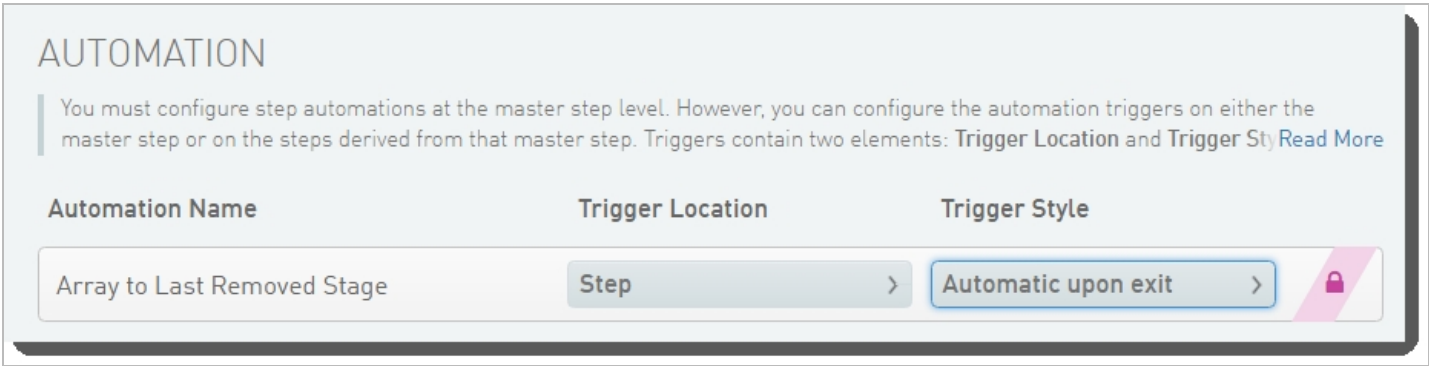
6. Enable the automation on the master step you created in step 2.



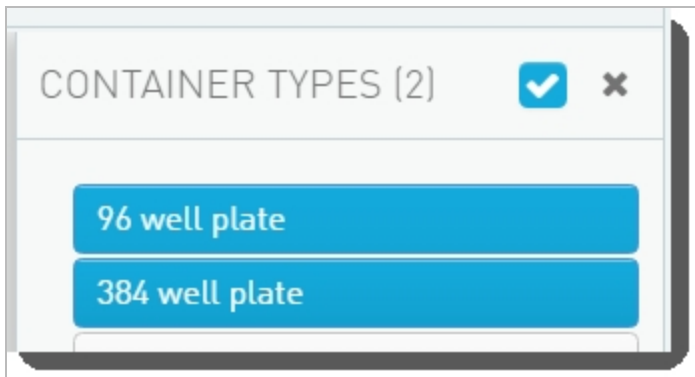
7. Configure a new protocol and step as follows.
- a. On the Lab Work tab, create a non-QC protocol.
 - b. In the Protocols list, select the new protocol and then add a new step to it.
Base the new step on the master step you created in step 2.



- c. On the Step Settings form, in the Automation section, you see the step automation you configured. Configure the automation triggers as follows.
 - **Trigger Location**—Step
 - **Trigger Style**—Automatic upon exit



- d. On the Placement milestone, Add 96 well plate and 384 well plate as the permitted destination container types for the step.



- e. Remove the default Tube container type.



- f. Save the step.
8. Configure a new workflow as follows:
 - a. On the Lab Work tab, create a workflow.
 - b. Add the protocol you created to the workflow.

AssignToLastRemovedStage Script Methods

Step 1: Build List of Artifact Nodes from the Step Input-Output Map

The first step of `AssignToLastRemovedStage` script is the same as for the `AssignToRearrayWf` script: return the artifact node list.

However, in this script, you are not directly given the artifact LIMS IDs. Instead, because you receive the step URI from the process parameter command line, you can collect the artifact URIs from the inputs of the step details input-output map using the `getArtifactNodes` function.

An example step details URI might be `{hostname}/api/v2/steps/{stepLIMSID}/details`.

```
def getArtifactNodes (stepDetails, username, password) {  
    artifactURIList = GLSRestApiUtils.httpGET(stepDetails, username, password) \  
        .'input-output-maps'. 'input-output-map'.collect{it.input.@uri[0]}.toSet()  
    return GLSRestApiUtils.batchGET(artifactURIList, username, password)  
}
```

Step 2: Assign Artifacts Back to the Original Stage

Each artifact in the list was removed from this stage before going through the rearray step.

With this in mind, and because the Clarity LIMS API stores artifact history by time (including stage history), the stage to which you now want to assign the samples to be the second-to-last stage in the `workflow-stage` list.

The following method finds the stage from which the artifacts were removed using the `getLastRemoved` function:

```
def getLastRemoved (artifactNodeList) {  
    def stageList = artifactNodeList[0]. 'workflow-stages'. 'workflow-stage'  
    stageToQueue = (stageList[(stageList.size()) - 2].@status.toString().equals  
        ('REMOVED')) ? stageList[(stageList.size()) - 2].@uri : null
```

Step 3: Make Sure that Artifacts are Assigned to the Same Stage

You can then check to make sure all artifacts originated in this stage. This helps you avoid the scenario where the `AssignToRearrayStage.groovy` script was run on two groups of artifacts queried while in different workflow stages.

```
if (!stageToQueue) {  
    println "Failed to requeue, ${artifactNodeList[0].name.text()} may have been  
    assigned to a new workflow"  
    System.exit(-1)  
} else {  
    artifactNodeList.each {  
        nodeStageList = it. 'workflow-stages'. 'workflow-stage'  
        lastRemoved = nodeStageList[(nodeStageList.size() - 2)].@status.toString  
            ().equals('REMOVED') ? nodeStageList[(nodeStageList.size()) - 2].@uri : null  
        if (!lastRemoved.toString().equals(stageToQueue.toString())) {  
            println "Sample ${it.name.text()} started in a separate stage, verify all  
            samples are being arrayed for the same workflow"  
            System.exit(-1)  
        }  
    }  
}
```

```
}
```

Step 4: Build the XML to Assign the Samples Back to the Returned Stage

This returned stage URI is then used to build the assignment XML to assign all the samples back to this stage with the `assignStage` function.

```
def assignStage (artifactNodeList, assignStageURI) {  
  def builder = new StreamingMarkupBuilder()  
  builder.encoding = "UTF-8"  
  def assignmentXML = builder.bind {  
    mkp.xmlDeclaration()  
    mkp.declareNamespace(rt: 'http://genomics.com/ri/routing')  
    'rt:routing' {  
      'assign'('stage-uri': assignStageURI) {  
        artifactNodeList.each { artifactNode ->  
          'artifact'(uri: artifactNode.uri)  
        }  
      }  
    }  
  }  
  return GLSRestApiUtils.xmlStringToNode(assignmentXML.toString())  
}
```

After posting this XML node, the samples are assigned back to the stage in which they began.

Step 3: Running the Assign to Rearray Automation

1. In the Projects Dashboard, select the samples to be rearranged. Run the Assign to Rearray automation. On the automation trigger, the `{userinput}` phrase invokes a dialog that prompts for the full name of the workflow.

The screenshot shows a table with columns: Sample Name, Current Volume (uL), Application, Sample Type, and Action. The table lists six samples, all with 'SampleForRobot' names and 'Derived Sample' types. A modal dialog titled 'Please Enter Parameter Value' is open, prompting for 'WorkflowName' with the example text 'Example: NIPT Standard for HiSeq'. The dialog has 'Cancel' and 'Continue' buttons.

Sample Name	Current Volume (uL)	Application	Sample Type	Action
SampleForRobot-25			Derived Sample	Derived Sample
SampleForRobot-26			Submitted Sample	Submitted Sample
SampleForRobot-26			Derived Sample	Derived Sample
SampleForRobot-27			Submitted Sample	Submitted Sample
SampleForRobot-27			Derived Sample	Derived Sample
SampleForRobot-28			Submitted Sample	Submitted Sample
SampleForRobot-28			Derived Sample	Derived Sample

Step 4: Placing the Samples and Running the Rearray Step

1. In Clarity LIMS, under Lab View, select the protocol you created in [Step 2: Configure the Placement Workflow, Protocol, and Step](#).

The samples assigned by the Assign to Rearray automation is available to assign to a new container.

Place Samples in Container Total 3

Step 1 > Array Samples to new Container 3

2. Add the samples to the Ice Bucket and begin work.
3. The placement screen opens, allowing you to place the samples into the new container in your desired placement pattern.

Samples To Be Placed

96 well plate 27-1942 1 of 1

Grid highlights: B8, C3, C4

#	Sample Name	Well	Container Na...
---	-------------	------	-----------------

Placed Samples

96 well plate 27-7205 1 of 1

Grid highlights: C2, C3, C4

Well	Sample Name	Container Na...	Project Name
C:2	SampleForRobo...	27-7205	Samples to Re...
C:3	SampleForRobo...	27-7205	Samples to Re...
C:4	SampleForRobo...	27-7205	Samples to Re...

4. Proceed to the Record Details screen, then on to Next Steps. Do not perform any actions on these screens.
5. In the next step drop-down list, select **Mark Protocol as Complete** and select **Apply**.
6. Select **Next**. This initiates the 'Assign to last removed' trigger, which assigns the samples back to the step from which they were removed.

Attachments

- [AssignToRearrayWf.groovy.txt](#)

- [AssignToLastRemoved.groovy.txt](#)

Work with Process/Step Outputs

When working with process and step outputs, you can do the following:

- [Update UDF/Custom Field Values for a Derived Sample Output](#)
- [Rename Derived Samples Using the API](#)
- [Find the Container Location of a Derived Sample](#)
- [Traverse a Pooled and Demultiplexed Sample History/Genealogy](#)
- [View the Inputs and Outputs of a Process/Step](#)

Update UDF/Custom Field Values for a Derived Sample Output

As processing occurs in the lab, associated processes and steps are run in Clarity LIMS. Often, key data must be recorded for the derived samples (referred to as analytes in the API) generated by these steps.

The following example explains how to change the value of an analyte UDF/global custom field.

If you would like to update a batch of output derived samples (analytes), you can increase the script execution speed by using batch operations. For more information, refer to the [Batch Resources Overview](#) documentation.

Prerequisites

In LIMS v4.2 or earlier, the key data fields are configured as user-defined fields (UDFs) on analytes. If you are using Clarity LIMS v4.2 and earlier, make sure you have the following items:

- A defined UDF named Library Size on Analyte.
- A configured Library Prep process to apply Library Size to output analytes (derived samples).
- A Library Prep process that has been run and has generated derived samples.

In Clarity LIMS v5 or later, the key data fields are configured as global custom fields on derived samples. If you are using Clarity LIMS v5 or later, make sure you have the following items:

- A defined global custom field named Library Size on the Derived Sample object.
- A configured Library Prep step to apply Library Size to generated derived samples.
- A Library Prep process that has been run and has generated derived samples.

Terminology

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:

- The master step on which the fields are configured.
- The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

Code Example

In Clarity LIMS v4.2 and earlier, the Operations Interface contains the process summary view, which displays detailed information about a process and its input and output artifacts. You can view the artifact UDFs associated with the process type by selecting outputs and then opening the Details tab below the Input/Output Explorer.

In the following process summary view, the Details tab displays the Library Size values for the selected output analytes.

-  An analyte resource generally contains specific details recorded in lab processes. Those specific details are often stored in analyte UDFs. Analyte UDFs must also be associated with a process. When you update an analyte, only the analyte UDFs that are configured for the process that created the analyte can be updated.

[A14-BMJ-100830-24-1472] API Cookbook Example 1.4

LIMS ID: A14-BMJ-100830-24-1472

Technician: Brandon Johnson

Date Run: Aug 30, 2010

Instrument/Software:

Input/ Output Explorer

Files

Details

Notes

Inputs

QC	Name	Container Identifier	Vol
+	Colon-1	27-1316	0.0
+	Colon-2	27-1317	0.0
+	Colon-3	27-1318	0.0
+	Colon-4	27-1319	0.0
+	Colon-5	27-1320	0.0
+	Colon-6	27-1321	0.0

Outputs

QC	Name	Container Identifier	Vol
+	Colon-1	27-2317	0.0
+	Colon-2	27-2318	0.0
+	Colon-3	27-2319	0.0
+	Colon-4	27-2320	0.0
+	Colon-5	27-2321	0.0
+	Colon-6	27-2322	0.0

Adapter Ligation for Tra...

Run Process...

Selected Artifact Details

Sample Genealogy

Details

Details for

Output per Input: Analyte

Show Non-Detail Fields

Detail Comparison

QC	Name	LIMS ID	Library Size
	Colon-1	HAM754A1AP8	3.00
	Colon-2	HAM754A2AP8	1.00
	Colon-3	HAM754A3AP8	4.00
	Colon-4	HAM754A4AP8	1.00
	Colon-5	HAM754A5AP8	5.00
	Colon-6	HAM754A6AP8	9.00

In Clarity LIMS v5 and later, the Record Details screen displays the information about the derived samples generated by a step. You can view the global fields associated with the derived samples in the Sample Table.

The following screenshot shows the Library Size values for the derived samples.

i | Derived sample information is stored in the API in the analyte resource. Step information is stored in the process resource. Each global field value is stored as an udf.

An analyte resource contains specific derived sample details that are recorded in lab steps. Those details are typically stored in global custom fields (configured in Clarity LIMS on the Derived Sample object) and then associated with the step.

When you update the information for a derived sample by updating the analyte API resource, only the global fields that are associated with the step can be updated.

JP Cookbook Example

Record Details (8 samples)

System Administrator

Started

Updated

Apr 21, 2017 10:35:21

Apr 21, 2017 10:35:22

Step Data Table

No Step Details to Record

Sample Table

Container Name

Apply

Sample Name

Container Name

Well

LIMS ID (Container)

Project Name

Library Size

Single Well: Tube (8 samples)

Kidney-1060Z5

27-204

27-204

JP Project

3.00

Kidney-1176E2

27-202

27-202

JP Project

1.00

Kidney-1332L4

27-203

27-203

JP Project

4.00

Kidney-1400L5

27-206

27-206

JP Project

1.00

Kidney-1618C6

27-209

27-209

JP Project

5.00

Kidney-1680A4

27-208

27-208

JP Project

9.00

Step 1. Request the Process Resource

To update the derived samples generated by a step, you must first request the process resource through a GET method.

The following GET method provides the full XML structure for the step:

```
// Retrieve Process processURI = "http://${hostname}/api/v2/processes/${processLIMSID}"  
process = GLSRestApiUtils.httpGET(processURI, username, password)
```

The process variable now holds the complete XML structure returned from the GET request.

The XML returned from a GET on the process resource contains the URIs of the process output artifacts (the derived samples generated by the step). You can use these URIs to query for each individual artifact resource.

The process resource contains many input-output-map elements, where each element represents an artifact. The following snippet of the XML shows the process:



Because processes with multiple inputs and outputs tend to be large, many of the input-output-map nodes have been omitted from this example.

```
<prc:process uri="http://yourIPaddress/api/v2/processes/A14-BMJ-100830-24-1472"
limsid="A14-BMJ-100830-24-1472"> <type>API Cookbook Example 1.4</type> <date-run>2010-08-
30</date-run> <technician uri="http://yourIPaddress/api/v2/researchers/305"> <first-
name>Brandon</first-name> <last-name>Johnson</last-name> </technician> <input-output-map>
<input uri="http://yourIPaddress/api/v2/artifacts/HAM754A4PA1?state=15633" post-process-
uri="http://yourIPaddress/api/v2/artifacts/HAM754A4PA1?state=15641"
limsid="HAM754A4PA1"/> <output
uri="http://yourIPaddress/api/v2/artifacts/HAM754A4AP8?state=15644" output-type="Analyte"
limsid="HAM754A4AP8"/> </input-output-map> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/HAM754A1PA1?state=15622" post-process-
uri="http://yourIPaddress/api/v2/artifacts/HAM754A1PA1?state=15646"
limsid="HAM754A1PA1"/> <output
uri="http://yourIPaddress/api/v2/artifacts/HAM754A1AP8?state=15642" output-type="Analyte"
limsid="HAM754A1AP8"/> </input-output-map> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/HAM754A3PA1?state=15626" post-process-
uri="http://yourIPaddress/api/v2/artifacts/HAM754A3PA1?state=15637"
limsid="HAM754A3PA1"/> <output
uri="http://yourIPaddress/api/v2/artifacts/HAM754A3AP8?state=15639" output-type="Analyte"
limsid="HAM754A3AP8"/> </input-output-map></prc:process>
```

Step 2. Request the Artifact Resource and Update the Analyte UDF/Custom Field

After you have retrieved each individual artifact resource, you can use this information to update the UDFs/custom fields for each output analyte after you request its resource.

Request the analyte output resource and update the UDF/custom field as follows.

1. If the output-type is analyte, then run through each input-output-map and request the output artifact resource.
2. Use a GET to return the XML for each artifact and store it in a variable.
3. When you have the analytes stored, change the analyte UDF/custom field through the following methods:
 - The UDF/custom field change in the XML.
 - The http PUT call to update the artifact resource.

The UDF/custom field change can be achieved with the Library Size UDF/custom field XML element defined in the following code. In this example, the Library Size value is updated to 25.

The PUT method updates the artifact resource at the specified URI using the complete XML representation, including the UDF/custom field. The setUdfValue method of the util library is used to perform this in a safe manner.

```
// For each input-output-map, if its output is an Analyte, set its Library Size UDF
process.'input-output-map'.each { if (it.output.@'output-type'[0] == "Analyte") {
  analyteURI = it.output.@uri[0] analyte = GLSRestApiUtils.httpGET(analyteURI, username,
  password) analyte = GLSRestApiUtils.setUdfValue(analyte, 'Library Size', '25')
  GLSRestApiUtils.httpPUT(analyte, analyte.@uri, username, password) } }
```

The output-type attribute is the user-defined name for each of the output types generated by a process/step. This is not equivalent to the type element of an artifact whose value is one of several hard-coded artifact types. If you must filter inputs or outputs from the input-output-map based on the artifact type, you will must GET each artifact in question to discover its type.

 It is important that you remove the state from each of the analyteURIs before you GET them, to make sure that you are working with the most recent state.

Otherwise, when you PUT the analyteURI back with your UDF/custom field changes, you can inadvertently revert information, such as QC, volume, and concentration, to previous values.

Expected Output and Results

The results can be reviewed in a web browser through the following URI:

[http://yourIPaddress/api/v2/artifacts/HAM754A4AP8\](http://yourIPaddress/api/v2/artifacts/HAM754A4AP8)

```
<art:artifact uri="http://yourIPaddress/api/v2/artifacts/HAM754A4AP8?state=15644"
limsid="HAM754A4AP8"> <name>Colon-4</name> <type>Analyte</type> <output-
type>Analyte</output-type> <parent-process uri="http://yourIPaddress/api/v2/processes/A14-
BMJ-100830-24-1472" limsid="A14-BMJ-100830-24-1472"/> <volume unit="uL">0.0</volume>
<concentration unit="ug/mL">10.0</concentration> <qc-flag>UNKNOWN</qc-flag> <location>
<container uri="http://yourIPaddress/api/v2/containers/27-2320" limsid="27-2320"/>
<value>1:1</value> </location> <working-flag>true</working-flag> <sample
uri="http://yourIPaddress/api/v2/samples/HAM754A4" limsid="HAM754A4"/> <udf:field
type="Numeric" name="Library Size">25</udf:field></art:artifact>
```

In Clarity LIMS v4.2 and earlier, in the Operations Interface, the process summary view now shows the updated Library Size.

[A14-BMJ]-100830-24-1472] API Cookbook Example 1.4

LIMS ID: A14-BMJ-100830-24-1472

Technician: Brandon Johnson

Date Run: Aug 30, 2010

Instrument/Software:

Input/ Output Explorer

Files

Details

Notes

Inputs

QC	Name	Container Identifier	Vol
+	Colon-1	27-1316	0.0
+	Colon-2	27-1317	0.0
+	Colon-3	27-1318	0.0
+	Colon-4	27-1319	0.0
+	Colon-5	27-1320	0.0
+	Colon-6	27-1321	0.0

Outputs

QC	Name	Container Identifier	Vol
+	Colon-1	27-2317	0.0
+	Colon-2	27-2318	0.0
+	Colon-3	27-2319	0.0
+	Colon-4	27-2320	0.0
+	Colon-5	27-2321	0.0
+	Colon-6	27-2322	0.0

Adapter Ligation for Tra...

Run Process...

Selected Artifact Details

Sample Genealogy

Details

Details for

Output per Input: Analyte

Show Non-Detail Fields

Detail Comparison

QC	Name	LIMS ID	Library Size
	Colon-1	HAM754A1AP8	25.00
	Colon-2	HAM754A2AP8	25.00
	Colon-3	HAM754A3AP8	25.00
	Colon-4	HAM754A4AP8	25.00
	Colon-5	HAM754A5AP8	25.00
	Colon-6	HAM754A6AP8	25.00

In Clarity LIMS v5 or later, in the Record Details screen, the Sample table now shows the updated Library Size.

Sample Table					
Container Name		Apply			
Sample Name	Container Name	Well	LIMS ID (Container)	Project Name	Library Size
▼ Single Well: Tube (8 samples)					
Kidney-1060Z5	27-204		27-204	JP Project	25.00
Kidney-1176E2	27-202		27-202	JP Project	25.00
Kidney-1332L4	27-203		27-203	JP Project	25.00
Kidney-1400L5	27-206		27-206	JP Project	25.00
Kidney-1618C6	27-209		27-209	JP Project	25.00
Kidney-1680A4	27-208		27-208	JP Project	25.00

Attachments

[UpdateUDFAnalyteOutput.groovy.txt](#)

Rename Derived Samples Using the API

Lab scientists must understand the priority of the samples they are working with. To help them prioritize their work, you can rename the derived samples generated by a step so that they include the priority assigned to the original submitted sample.

If you would like to rename a batch of derived samples, you can increase the script execution speed by using batch operations. You can also use a script to rename a derived sample after a step completes.

Prerequisites

If you are using Clarity LIMS v4.2 and earlier, make sure that you have done the following actions:

- Added samples to the system.
- Defined the sample UDF/custom field named Priority. It should have sp1, sp2, and sp3 preset values.
- Run a step on samples with the Priority value set to sp1, sp2, or sp3.

If you are using Clarity LIMS v5 and later, make sure that you have done the following actions:

- Added samples to the system.
- Defined a global custom field named Priority on the Submitted Sample object. The field should have default values sp1, sp2, and sp3, and it should be enabled on a step.
- Run samples through the step with the Priority of each sample set to sp1, sp2, or sp3.

Code Example

In this example, six samples have been added to a project in Clarity LIMS. The submitted samples names are Heart-1 through Heart-6. The samples are run through a step that generates derived samples, and the priority of each sample is set.

By default, the name of the derived samples generated by the step would follow the name of the original submitted samples as shown in the Assign Next Steps screen of the step.

JP Cookbook Example
Assign Next Steps (6 samples)

System Administrator

Started
Updated
Apr 24, 2017 16:57:09
Apr 24, 2017 16:58:11

Samples					
Sample Name	*Next Step	Container Name	Well	LIMS ID (Container)	Priority
▼ Single Well: Tube (6 samples)					
Heart-1	Mark protocol as complete	27-303		27-303	sp1
Heart-2	Mark protocol as complete	27-302		27-302	sp3
Heart-3	Mark protocol as complete	27-304		27-304	sp1
Heart-4	Mark protocol as complete	27-306		27-306	sp2
Heart-5	Mark protocol as complete	27-307		27-307	sp1
Heart-6	Mark protocol as complete	27-305		27-305	sp2

This example appends the priority of the submitted sample to the name of the derived sample output. The priority is defined by the Priority sample UDF (in Clarity LIMS v4.2 or earlier) or the Priority submitted sample custom field (in Clarity LIMS v5 or later).

Renaming the derived sample consists of the following steps:

- Request the step information (process resource) for the step that generated the derived sample (analyte resource).
- Request the individual analyte resource for the derived sample to be renamed.
- Request the sample resource linked from the analyte resource to get the submitted sample UDF/custom field value to use for the update.
- Update the individual analyte output resource with the new name.

Step 1. Request the Step Information (Process Resource) for the Step that Generated the Derived Sample (Analyte Resource)

When using the REST API, you will often start with the LIMS ID for the step that generated a derived sample. The key API concepts are as follows.

- Information about a step is stored in the process resource.
- In general, automation scripts access information about a step using the processURI, which links to the individual process resource. The input-output-map in the XML returned by the individual process resource gives the script access to the artifacts that were inputs and outputs to the process.
- Information about a derived sample is stored in the analyte resource. This is used as the input and output of a step.

- Analytes are also used to record specific details from lab processing.
- The XML representation for an individual analyte contains a link to the URI of its submitted sample, and to the URI of the process that generated it (parent process).

The following GET method returns the full XML structure for the step:

```
// Retrieve the process processURI =
"http://${hostname}/api/v2/processes/${processLIMSID}" process = GLSRestApiUtils.httpGET
(processURI, username, password)
```

The process variable now holds the complete XML structure returned from the process GET request, as shown in the following example. The URI for each analyte generated is given in the output node in each input-output-map element. For more information on the input-output-map, refer to [View the Inputs and Outputs of a Process/Step](#).

```
<prc:process uri="http://yourIPAddress/api/v2/processes/A13-BMJ-100830-24-1475"
limsid="A13-BMJ-100830-24-1475"> <type>API Cookbook Example 1.3</type> <date-run>2010-08-
30</date-run> <technician uri="http://yourIPAddress/api/v2/researchers/305"> <first-
name>Brandon</first-name> <last-name>Johnson</last-name> </technician> <input-output-map>
<input uri="http://yourIPAddress/api/v2/artifacts/HAM754A3PA1?state=15657" post-process-
uri="http://yourIPAddress/api/v2/artifacts/HAM754A3PA1?state=15678"
limsid="HAM754A3PA1"/> <output
uri="http://yourIPAddress/api/v2/artifacts/HAM754A3AP10?state=15683" output-
type="Analyte" limsid="HAM754A3AP10"/> </input-output-map> <input-output-map> <input
uri="http://yourIPAddress/api/v2/artifacts/HAM754A1PA1?state=15651" post-process-
uri="http://yourIPAddress/api/v2/artifacts/HAM754A1PA1?state=15685"
limsid="HAM754A1PA1"/> <output
uri="http://yourIPAddress/api/v2/artifacts/HAM754A1AP10?state=15680" output-
type="Analyte" limsid="HAM754A1AP10"/> </input-output-map> <input-output-map> <input
uri="http://yourIPAddress/api/v2/artifacts/HAM754A2PA1?state=15656" post-process-
uri="http://yourIPAddress/api/v2/artifacts/HAM754A2PA1?state=15686"
limsid="HAM754A2PA1"/> <output
uri="http://yourIPAddress/api/v2/artifacts/HAM754A2AP10?state=15681" output-
type="Analyte" limsid="HAM754A2AP10"/> </input-output-map> <input-output-map> <input
uri="http://yourIPAddress/api/v2/artifacts/HAM754A6PA1?state=15659" post-process-
uri="http://yourIPAddress/api/v2/artifacts/HAM754A6PA1?state=15679"
limsid="HAM754A6PA1"/> <output
uri="http://yourIPAddress/api/v2/artifacts/HAM754A6AP10?state=15677" output-
type="Analyte" limsid="HAM754A6AP10"/> </input-output-map> <input-output-map> <input
uri="http://yourIPAddress/api/v2/artifacts/HAM754A4PA1?state=15655" post-process-
uri="http://yourIPAddress/api/v2/artifacts/HAM754A4PA1?state=15682"
limsid="HAM754A4PA1"/> <output
uri="http://yourIPAddress/api/v2/artifacts/HAM754A4AP10?state=15687" output-
type="Analyte" limsid="HAM754A4AP10"/> </input-output-map> <input-output-map> <input
uri="http://yourIPAddress/api/v2/artifacts/HAM754A5PA1?state=15652" post-process-
```

```
uri="http://yourIPaddress/api/v2/artifacts/HAM754A5PA1?state=15684"
limsid="HAM754A5PA1"/> <output
uri="http://yourIPaddress/api/v2/artifacts/HAM754A5AP10?state=15688" output-
type="Analyte" limsid="HAM754A5AP10"/> </input-output-map></prc:process>
```

Step 2. Request the Individual Resource for the Derived Sample to be Renamed (Analyte Resource)

Each output node has an output-type attribute that is the user-defined type name of the output. You can iterate through each input-output-map and request the output artifact resource for each output of a particular output-type.

In the following code example, we filter on output-type = Analyte.

```
// For each input-output-map process.'input-output-map'.each { if (it.output.@'output-
type'[0] == "Analyte") { // Retrieve the analyte analyteURI = it.output.@uri[0] analyte =
GLSRestApiUtils.httpGET(analyteURI, username, password) // Retrieve the analyte's sample
and get its Priority UDF's value sampleURI = analyte.sample.@uri[0] sample =
GLSRestApiUtils.httpGET(sampleURI, username, password) samplePriority =
sample.'udf:field'.find { it.@name== 'Priority' }?.text() // Rename the analyte nameNode
= analyte.name[0]
```

The output-type attribute is the user-defined name for each of the output types generated by a process. This is not equivalent to the type element of an artifact whose value is one of several hard-coded artifact types.

If you must filter inputs or outputs from the input-output-map based on the artifact type, you need to GET each artifact in question to discover its type.

 It is important that you remove the state from each of the analyteURIs before you GET them to make sure that you are working with the most recent state. Otherwise, when you PUT the analyteURI back with your UDF changes, you can inadvertently revert information (eg, QC, volume, and concentration) to their previous values.

Step 3. Request the Sample Resource to Get the Field Value to Use for the Update

From the analyte XML, you can use the submitted sample URI to return the sample that maps to that analyte.

[Update Sample Information](#) shows how to set a sample UDF/global field. To get the value of a sample UDF/global field, use the same method to find the field, and then use the `.text()` method to get the field value.

The value of the UDF/custom field is stored in the variable `samplePriority` so that it is then available for the renaming step.

```
// For each input-output-map process.'input-output-map'.each { if (it.output.@'output-
type'[0] == "Analyte") { // Retrieve the analyte analyteURI = it.output.@uri[0] analyte =
GLSRestApiUtils.httpGET(analyteURI, username, password) // Retrieve the analyte's sample
and get its Priority UDF's value sampleURI = analyte.sample.@uri[0] sample =
```

```
GLSRestApiUtils.httpGET(sampleURI, username, password) samplePriority =
sample.'udf:field'.find { it.@name== 'Priority' }?.text() // Rename the analyte nameNode
= analyte.name[0]
```

The variable `analyte` holds the complete XML structure returned from a GET on the URI in the output node. The variable `nameNode` references the XML element in that structure that contains the name of the artifact. The following example shows the XML for the analyte named Heart-1.

```
<art:artifact uri="http://yourIPaddress/api/v2/artifacts/AFF853A43AP2?state=20985"
limsid="AFF853A43AP2"> <name>Heart-1</name> <type>Analyte</type> <output-
type>Analyte</output-type> <parent-process
uri="http://yourIPaddress/api/v2/processes/A13-BMJ-100923-24-2182" limsid="A13-BMJ-
100923-24-2182"/> <qc-flag>UNKNOWN</qc-flag> <location> <container
uri="http://yourIPaddress/api/v2/containers/27-303" limsid="27-303"/> <value>1:1</value>
</location> <working-flag>true</working-flag> <sample
uri="http://yourIPaddress/api/v2/samples/AFF853A43" limsid="AFF853A43"/></art:artifact>
```

Step 4. Update the Analyte Resource with the Name Change

Renaming the derived sample consists of two steps:

1. The name change in the XML.
2. The PUT call to update the analyte resource.

The name change can be performed with the `nameNode` XML element node defined. The following example shows this element defined.

```
newName = nameNode.text() + " " + samplePriority nameNode.setValue(newName) returnNode =
GLSRestApiUtils.httpPUT(analyte, analyte.@uri, username, password) } }
```

The http PUT command updates the artifact resource using the complete XML representation, including the new name.

Expected Output and Results

After a successful PUT, the results can be reviewed in a web browser at <http://yourIPaddress/api/v2/artifacts/TST110A291AP45>.

The following XML resource is returned from the PUT command and is stored in `returnNode`.

```
<art:artifact uri="http://yourIPaddress/api/v2/artifacts/AFF853A43AP2?state=20985"
limsid="AFF853A43AP2"> <name>Heart-1 spl</name> <type>Analyte</type> <output-
type>Analyte</output-type> <parent-process uri="http://yourIPaddress/api/v2/processes/A13-
BMJ-100923-24-2182" limsid="A13-BMJ-100923-24-2182"/> <qc-flag>UNKNOWN</qc-flag> <location>
<container uri="http://yourIPaddress:/api/v2/containers/27-303" limsid="27-303"/>
<value>1:1</value> </location> <working-flag>true</working-flag> <sample
uri="http://yourIPaddress/api/v2/samples/AFF853A43" limsid="AFF853A43"/></art:artifact>
```

In Clarity LIMS, the Assign Next Steps screen shows the new names for the generated derived samples.

JP Cookbook Example
Assign Next Steps (6 samples)

System Administrator

Started
Updated

Apr 24, 2017 16:57:09
Apr 24, 2017 16:58:11

Samples
*required
☒ Mark protocol as complete
Apply

Sample Name ▲	* Next Step	Container Name	Well	LIMS ID (Container)	Priority
▼ Single Well: Tube (6 samples)					
Heart-1 sp1	☒ Mark protocol as complete	27-303		27-303	sp1
Heart-2 sp3	☒ Mark protocol as complete	27-302		27-302	sp3
Heart-3 sp1	☒ Mark protocol as complete	27-304		27-304	sp1
Heart-4 sp2	☒ Mark protocol as complete	27-306		27-306	sp2
Heart-5 sp1	☒ Mark protocol as complete	27-307		27-307	sp1
Heart-6 sp2	☒ Mark protocol as complete	27-305		27-305	sp2

i | This example shows simple renaming of derived samples based on a submitted sample UDF/global field. However, you can use step names, step UDFs (known as master step fields in Clarity LIMS v5 or later), project information, and so on, to rename derived samples and provide critical information to scientists working in the lab.

Attachments

[UpdateAnalyteName.groovy.txt](#)

Find the Container Location of a Derived Sample

As samples are processed in the lab, substances are moved from one container to another. Because container locations are sometimes used to reference the sample in data files, tracking the location of these substances within containers is one of the key values that Clarity LIMS provides to the lab.

Within the REST API (v2 r21 or later), analytes represent the substances on which processes/steps are run. These analytes are the substances that are chemically altered and transferred between containers as samples are processed in the lab.

Each individual sample resource has an analyte artifact that describes its container location and is used to run processes.

In Clarity LIMS, steps are not run on the original submitted samples, but are instead run on (and can also generate) derived samples. In the API, derived samples are known as analytes. Each sample resource, which is the original submitted sample in Clarity LIMS, has a corresponding analyte that is used for running processes/steps and describing placement in a container.

For more information on analyte artifacts and other REST resources, see [Structure of REST Resources](#).

Prerequisites

For all Clarity LIMS users, make sure you have done the following actions:

- Added a sample to Clarity LIMS.
- Run a process/step on the sample, with the same process/step generating a derived sample output.
- Added the generated derived sample to a multi-well container (eg, a 96-well plate).

Code Example

The container location information for an individual derived sample/analyte is located within the XML for the individual artifact resource. Because artifacts are generated by running steps in the LIMS, this is a logical place to keep track of the location.

Within a script, you can use a GET method to request the artifact. The resulting XML structure contains all the information related to the artifact, including its container and well location.

In this example, a derived sample named Brain-600 is placed in well A:1 of a container with LIMS ID 27-1259. This information is found in the location element. The information is highlighted in red.

```
<art:artifact uri="http://yourIPAddress/api/v2/artifacts/HAM751A481PA1?state=9995"
limsid="HAM751A481PA1"> <name> Brain-600 </name> <type>Analyte</type> <output-
type>Analyte</output-type> <volume unit="uL">645.0</volume> <concentration
unit="ug/mL">0.5478</concentration> <qc-flag>UNKNOWN</qc-flag> <location> <container
uri="http://yourIPAddress/api/v2/containers/27-1259" limsid="27-1259"/> <value>A:1</value>
</location> <working-flag>true</working-flag> <sample
uri="http://yourIPAddress/api/v2/samples/HAM751A481" limsid="HAM751A481"/></art:artifact>
```

The location elements have two child data elements:

- One linking to the container URI, which specifies which container the analyte is in.
- One for the well location, which has the name value in the XML structure.

Valid values for a well location can be either numeric or alphabetic, and are determined by the configuration of the container in Clarity LIMS.

Well locations are always represented in the row:column format. For example, a 96-well plate can have locations A:1 and C:12, and a tube can have a single well called 1:1.

Step 1. Retrieve the Artifact

Use the following XML example to retrieve the artifact:

```
// Retrieve the artifact artifactURI =
"http://${hostname}/api/v2/artifacts/${artifactLIMSID}" artifact =
GLSRestApiUtils.httpGET(artifactURI, username, password)
```

Step 2. Access, Store, and Print the Container Location

Because the container position is structured in the row:column format, you can store the row and column in separate variables by splitting the container position on the colon character. You can access the string value of the location value node using the `text()` method, as shown in the following code:

```
// Separate the artifact's position inside of its container containerPosition =  
artifact.location.value.text() positionList = containerPosition.tokenize(':') // Output  
its position row = positionList[0] column = positionList[1] println "This sample is  
located at row: $row, column: $column"
```

Expected Output and Results

Running the script in a console produces the following output:

```
This sample is located at row: A, column: 1
```

Attachments

[GetContainerAnalyteLocation.groovy.txt](#)

Traverse a Pooled and Demultiplexed Sample History/Genealogy

The large capacity of current Next Generation Sequencing (NGS) instruments means that labs are able to perform multiplexed experiments with multiple samples pooled into a single lane or region of the container. Before being pooled, samples are assigned a unique tag or index. After sequencing and initial analysis are complete, the sequencing results must be demultiplexed to separate data and relate the results back to each individual sample.

Clarity LIMS allows you to track a multiplexing workflow by adding reagents and reagent labels to artifacts, and then using the reagent labels to demultiplex the resulting files.

There are several ways to apply reagent labels. However, all methods involve creating placeholders that link the final sequences back to the original submitted samples. Either the lab scientist or an automated process must determine which file actually belongs with which placeholder. For more information on applying reagent labels, refer to [Work with Multiplexing](#).

This example walks through assigning user-defined field (UDF)/custom field values to the demultiplexed output files based on upstream derived sample (analyte) UDF/custom field values. This includes upwards traversal of a sample history / genealogy, based on assigned reagent labels. This differs from upstream traversal based strictly upon process input-output mappings.

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

Prerequisites

If you are using Clarity LIMS v4.2 or earlier, make sure you have completed the following actions:

- Created a project and have added multiple samples to it.
- Run the samples through a sequence of processes that perform the following actions:
 - Reagent addition / reagent label assignment
 - Pooling
 - Demultiplexing (to produce a set of per-reagent-label result file outputs)
- Set a Numeric UDF value on each analyte (derived sample) input to the reagent addition process.

i | A Numeric UDF with no assigned value exists on each of the per-reagent-label result file outputs. The value of this field will be computed from the set of upstream analyte UDF values corresponding to the result file's reagent label.

If you are using Clarity LIMS v5 or later, make sure you have completed the following actions:

- Created a project and have added multiple samples to it.
- Run the samples through a sequence of steps that perform the following:
 - Reagent addition / reagent label assignment
 - Pooling
 - Demultiplexing (to produce a set of per-reagent-label result file outputs)
- Set a Numeric custom field value on each derived sample input to the reagent addition process.

i | A Numeric custom field with no assigned value exists on each of the per-reagent-label result file outputs. The value of this field will be computed from the set of upstream derived sample custom field values corresponding to the reagent label of the result file.

You also must make sure that API v2 r21 or later is installed.

Code Example

Due to the complexity of NGS workflows, beginning at the top level submitted sample resource and working down to the result file is not the most efficient way to traverse the sample history/genealogy. It is easier to start with the result file artifact, and then trace upward to find the process with the UDFs/custom fields that you are looking for.

Starting from the per-reagent-label result file, you can traverse upward in the sample history using the parent process URI in the XML returned for each artifact. At each level of the sample history, the number of artifacts returned may increase due to processes that pooled individual artifacts.

In this example:

- The `upstreamArtifactLUIDs` list represents the current set of relevant artifacts.
- The `foundUpstreamArtifactNodes` list stores the target upstream artifact nodes found.

- The sample history traversal stops at the inputs to the process that performed the reagent addition/reagent label assignment.

```
targetDownstreamArtifactNode = GLSRestApiUtils.httpGET(artifactsListURI + artifactLUID,
username, password) targetReagentLabel = targetDownstreamArtifactNode.'reagent-label'
[0]?.'@name' if (!targetReagentLabel) { println "Specified artifact should contain at least
one reagent-label. Skipping ${artifactLUID}..." continue } // At each upstream level of the
workflow the number of searched artifacts may increase due to Pooling processes
upstreamArtifactLUIDs = [ artifactLUID ] /* * This 'stack' will store all upstream artifacts
that serve as input to an 'Add Multiple Reagents' process * which are subsequently assigned
the 'target' Reagent Label by this process. */ foundUpstreamArtifactNodes = []
```

The traversal is executed using a while loop over the contents of the upstreamArtifactLUIDs list.

The list serves as a stack of artifacts. With each iteration of the loop, an artifact is removed from the end of the list and the relevant input artifacts to its parent process are pushed back onto the end of the list.

```
while (!upstreamArtifactLUIDs.isEmpty()) { currentArtifactLUID = upstreamArtifactLUIDs.pop()
currentArtifactNode = GLSRestApiUtils.httpGET(artifactsListURI + currentArtifactLUID,
username, password) /* * Upstream traversal will stop when either an artifact is found that
does not have reagent label(s) assigned * (i.e. the artifact is the input to a process that
adds reagents and reagent labels), or a root artifact is found. * At this point, the current
artifact is added to the list of 'found' upstream artifact nodes. */ if
(currentArtifactNode.'reagent-label'.isEmpty() || currentArtifactNode.'parent-
process'.isEmpty()) { foundUpstreamArtifactNodes += [currentArtifactNode] } else if
(currentArtifactNode.'reagent-label'.collect { it.'@name' }.contains(targetReagentLabel)) {
/* * If the current artifact contains the 'target' reagent label, continue traversing
upstream. * Get the artifact's parent process */ parentProcessURI =
currentArtifactNode.'parent-process'[0].@uri parentProcessNode = GLSRestApiUtils.httpGET
(parentProcessURI, username, password) // Find all input-output maps for the parent process
parentProcessNode.'input-output-map'.each { ioMapInputLUID = it.'input'[0].@limsid
ioMapOutputLUID = it.'output'[0].@limsid // Push all process input artifacts that have the
current artifact as the mapped process output onto the 'stack' if( ioMapOutputLUID ==
currentArtifactLUID && !upstreamArtifactLUIDs.contains(ioMapInputLUID) ) {
upstreamArtifactLUIDs.push(ioMapInputLUID) } } } }
```

After the loop has executed, the foundUpstreamArtifactNodes list will contain all of the artifacts that are assigned the reagent label of interest upon execution of the next process in the sample history.

The final step in the script assigns a value to a Numeric UDF/custom field on the per-reagent-label output result file (Mean DNA Prep 260:280 Ratio), by computing the mean value of a Numeric UDF/custom field on each of the foundUpstreamArtifactNodes (eg, DNA prep 260:280 ratio).

First, compute the mean using the following example:

```
/* * Compute the 'Mean DNA Prep 260:280 Ratio' for all upstream analyte artifacts that have
the * target Reagent Label applied. The assumption here is that the 'DNA prep 260:280 ratio'
UDF * is set on analytes that serve as input to an 'Add Multiple Reagents' process that
assigns Reagent Labels. */ avgAcrossUpstreamArtifacts = foundUpstreamArtifactNodes.collect
{ foundUdf = it.'udf:field'.find{ it.'@name' == upstreamArtifactUdfToMine } return foundUdf
? foundUdf.value()[0] as double : 0.0 }.sum()/foundUpstreamArtifactNodes.size()
```

Then, set the UDF/custom field on the per-reagent-label output result file using the following example:

```
// Set the computed mean on the 'Mean DNA Prep 260:280 Ratio' UDF on the target downstream
ResultFile targetDownstreamArtifactUDF = targetDownstreamArtifactNode.'udf:field'.find{
it.'@name' == downstreamArtifactUdfToUpdate } if (targetDownstreamArtifactUDF) {
targetDownstreamArtifactUDF.setValue(avgAcrossUpstreamArtifacts) } else {
targetDownstreamArtifactNode.appendNode('udf:field', ['name':downstreamArtifactUdfToUpdate,
'xmlns:udf':'http://genologics.com/ri/userdefined'], avgAcrossUpstreamArtifacts) }
```

Attachments

[TraversingPooledDemuxGenealogy.groovy.txt](#)

View the Inputs and Outputs of a Process/Step

When samples are processed in the lab, they generally produce child samples that are altered in some way. Eventually, the samples are analyzed on an instrument, with the result being a data file. Often these data are analyzed further, which produces additional data files.

The sample processing that occurs in the lab is modeled as steps in the Clarity LIMS web interface. In the REST API (v2 r21 or later), this processing is modeled as processes, and the samples and files that are processed are represented as artifacts. Understanding the representation of inputs and outputs within the XML for an individual process is critical to being able to use the REST API effectively.

Prerequisites

If you are using Clarity LIMS v4.2 or earlier, make sure that you have done the following actions:

- Added samples to Clarity LIMS.
- Defined a process that has an analyte input and an analyte output (with a generated shared result file output) in the Operations Interface.
- Run the process on some samples.

If you are using Clarity LIMS v5 or later, make sure that you have done the following actions:

- Added samples to the LIMS.
- Configured a step that generates derived samples in the Lab Work tab.
- Configured a file placeholder for a sample measurement file to be generated and attached by an automation script at run time. This configuration is done in the Master Step Settings of the step on the Record Details milestone.

- Configured an automation that generates the sample measurement file and have enabled it on the step. This configuration is done in the Automation tab.
- Configured the automation triggers. This configuration is done in the Step Settings screen, under the Record Details milestone.
- Run the step on some samples.

Code Example

In Clarity LIMS v4.2 and earlier, the Input/Output Explorer (in the process summary view of the Operations Interface) lists the inputs and outputs for each process. If you select an input, the corresponding output is highlighted.

Input/Output ExplorerFilesDetailsNotes

Inputs

QC	Name	Container Identifier	Place
	Colon-1	27-2028	
	Colon-2	27-2029	
	Colon-3	27-2030	
	Colon-4	27-2031	
	Colon-5	27-2032	
	Colon-6	27-2033	

Outputs

QC	Name	Container Identifier	Place
	Colon-1	27-2034	
	Colon-2	27-2035	
	Colon-3	27-2036	
	Colon-4	27-2037	
	Colon-5	27-2038	
	Colon-6	27-2039	
	[92-13007] Colo...		

If you select an output, the corresponding inputs are highlighted.

Input/Output Explorer

Files

Details

Notes

Inputs

QC	Name	Container Identifier	Place
	Colon-1	27-2028	
	Colon-2	27-2029	
	Colon-3	27-2030	
	Colon-4	27-2031	
	Colon-5	27-2032	
	Colon-6	27-2033	

Outputs

QC	Name	Container Identifier	Place
	Colon-1	27-2034	
	Colon-2	27-2035	
	Colon-3	27-2036	
	Colon-4	27-2037	
	Colon-5	27-2038	
	Colon-6	27-2039	
	[92-13007] Colo...		

As of Clarity LIMS v5, the Operations Interface Java client has been deprecated. In LIMS v5 and later, there is no equivalent screen to the Input/Output Explorer where you can select step inputs/outputs and generated files and view their corresponding inputs/outputs and files.

However, the following API code example is still relevant and will produce the same results.

Step 1. Request Individual Process Resource

The first step in this example is to request the individual process resource through a GET method. The full XML representation returned includes the input-output-map.

To illustrate the relationships between the inputs and outputs, you can save them using a Groovy Map data structure. This maps the output LIMS IDs to a list of input LIMS IDs associated with each output, as shown in the following example:

```
outputToInputMap = [:] processURI =  
"http://${hostname}/api/v2/processes/${processLIMSID}" process = GLSRestApiUtils.httpGET  
(processURI, username, password)
```

The process variable now holds the complete XML structure returned from the processURI.

In the following example XML snippet, elements of the input-output-map are labeled with <input-output-map>:

```
<prc:process uri="http://IPAddress/api/v2/processes/TES-SA1-130107-24-5259" limsid="TES-  
SA1-130107-24-5259"> <type uri="http://IPAddress/api/v2/processtypes/355">Cookbook  
Example Process</type> <date-run>2013-01-07</date-run> <technician  
uri="http://IPAddress/api/v2/researchers/1"> <first-name>System</first-name> <last-  
name>Administrator</last-name> </technician> <input-output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A3PA1?state=8055"  
uri="http://IPAddress/api/v2/artifacts/ADM224A3PA1?state=8040" limsid="ADM224A3PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/92-13007?state=8065" output-generation-  
type="PerAllInputs" output-type="ResultFile" limsid="92-13007" /> </input-output-map>  
<input-output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A3PA1?state=8055"  
uri="http://IPAddress/api/v2/artifacts/ADM224A3PA1?state=8040" limsid="ADM224A3PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/ADM224A3TE3?state=8054" output-generation-  
type="PerInput" output-type="Analyte" limsid="ADM224A3TE3" /> </input-output-map> <input-  
output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A5PA1?state=8053"  
uri="http://IPAddress/api/v2/artifacts/ADM224A5PA1?state=8047" limsid="ADM224A5PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/92-13007?state=8065" output-generation-  
type="PerAllInputs" output-type="ResultFile" limsid="92-13007" /> </input-output-map>  
<input-output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A5PA1?state=8053"  
uri="http://IPAddress/api/v2/artifacts/ADM224A5PA1?state=8047" limsid="ADM224A5PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/ADM224A5TE3?state=8060" output-generation-  
type="PerInput" output-type="Analyte" limsid="ADM224A5TE3" /> </input-output-map> <input-
```



```
output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A2PA1?state=8056"  
uri="http://IPAddress/api/v2/artifacts/ADM224A2PA1?state=8042" limsid="ADM224A2PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/92-13007?state=8065" output-generation-  
type="PerAllInputs" output-type="ResultFile" limsid="92-13007" /> </input-output-map>  
<input-output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A2PA1?state=8056"  
uri="http://IPAddress/api/v2/artifacts/ADM224A2PA1?state=8042" limsid="ADM224A2PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/ADM224A2TE3?state=8059" output-generation-  
type="PerInput" output-type="Analyte" limsid="ADM224A2TE3" /> </input-output-map> <input-  
output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A4PA1?state=8063"  
uri="http://IPAddress/api/v2/artifacts/ADM224A4PA1?state=8048" limsid="ADM224A4PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/92-13007?state=8065" output-generation-  
type="PerAllInputs" output-type="ResultFile" limsid="92-13007" /> </input-output-map>  
<input-output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A4PA1?state=8063"  
uri="http://IPAddress/api/v2/artifacts/ADM224A4PA1?state=8048" limsid="ADM224A4PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/ADM224A4TE3?state=8057" output-generation-  
type="PerInput" output-type="Analyte" limsid="ADM224A4TE3" /> </input-output-map> <input-  
output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A1PA1?state=8061"  
uri="http://IPAddress/api/v2/artifacts/ADM224A1PA1?state=8046" limsid="ADM224A1PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/92-13007?state=8065" output-generation-  
type="PerAllInputs" output-type="ResultFile" limsid="92-13007" /> </input-output-map>  
<input-output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A1PA1?state=8061"  
uri="http://IPAddress/api/v2/artifacts/ADM224A1PA1?state=8046" limsid="ADM224A1PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/ADM224A1TE3?state=8058" output-generation-  
type="PerInput" output-type="Analyte" limsid="ADM224A1TE3" /> </input-output-map> <input-  
output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A6PA1?state=8064"  
uri="http://IPAddress/api/v2/artifacts/ADM224A6PA1?state=8045" limsid="ADM224A6PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/92-13007?state=8065" output-generation-  
type="PerAllInputs" output-type="ResultFile" limsid="92-13007" /> </input-output-map>  
<input-output-map> <input post-process-  
uri="http://IPAddress/api/v2/artifacts/ADM224A6PA1?state=8064"  
uri="http://IPAddress/api/v2/artifacts/ADM224A6PA1?state=8045" limsid="ADM224A6PA1" />  
<output uri="http://IPAddress/api/v2/artifacts/ADM224A6TE3?state=8062" output-generation-  
type="PerInput" output-type="Analyte" limsid="ADM224A6TE3" /> </input-output-  
map></prc:process>
```

All of the input and output URIs include a `?state= some number`. State allows Clarity LIMS to track historical values for QC, volume, and concentration, so you can compare the state of an analyte before and after a process was run. However, when you make changes to an artifact you should always work with the most current state.

To make sure that you are getting the current state when you do a GET request, simply remove the state from the artifact URI.

Step 2. Map Output LIMS IDs to Input LIMS IDs

You can examine each input-output-map to find details about the relationship represented between inputs and outputs. The following code puts the output and input LIMS IDs into an array named `outputToInputMap`.

As the output type is also important for further processing, `outputToInputMap` is formatted as follows:

```
outputLIMSID -> [output-type, inputLIMSID-1, inputLIMSID-2, inputLIMSID-3, ...|output-
type, inputLIMSID-1, inputLIMSID-2, inputLIMSID-3, ...]
```

If the output is shared for all inputs (eg, the sample measurement file with LIMS ID 92-13007), the inputs to the process are listed. If the output relates to an individual input, only the LIMS ID for that particular input will be listed.

```
// For each io-map in the process, add its information to outputToInputMap
process.'input-output-map'.each { outputType = it.'output'[0].@output-type outputLIMSID
= it.'output'[0].@limsid inputLIMSID = it.'input'[0].@limsid // outputToInputMap stores
all the output type and LIMS IDs of all the inputs to the output if (!outputToInputMap
[outputLIMSID]) { outputToInputMap[outputLIMSID] = [outputType, inputLIMSID] } else { //
If entry already exists, add another input to the list outputToInputMap[outputLIMSID] <<
inputLIMSID }}
```

Shared outputs are represented by multiple input-output-maps that have the same LIMS ID for the output. Inputs with multiple outputs are represented by multiple input-output-maps with the same LIMS ID for the input. Output-per-input relationships are represented by a single input-output-map for each input and output.

Outputs are listed in multiple input-output-map elements when they have multiple input files generating them. The first time any particular output LIMS ID is seen, the output type and input LIMS ID in the input-output-map are added to the list, stored in `outputToInputMap`.

If the output LIMS ID already has a list in `outputToInputMap`, then the code adds input LIMS ID to the list.

Step 3. Print the List

One way to access the information is to print it out. You can run through each key-value pair and print the information it contains, as shown in the following example:

```
// Print the contents of the map, which stores the inputs LIMSIDs under the output's
LIMSID
o+tputToInputMap.each { key, value -> println "$key is a(n) ${value[0]} with
input:" for (int i = 1; i < value.size(); i++) { println '\t' + value[i] } }
```

Expected Output and Results

After running the script on the command line, an output similar to the following will be generated, whereby the inputs used to generate each output are listed.

```
92-13007 is a(n) ResultFile with input: 27-2028 27-2029 27-2030 27-2031 27-2032 27-2033 27-2034 is a(n) Analyte with input: 27-2028 27-2035 is a(n) Analyte with input: 27-2029 27-2036 is a(n) Analyte with input: 27-2030 27-2037 is a(n) Analyte with input: 27-2031 27-2038 is a(n) Analyte with input: 27-2032 27-2039 is a(n) Analyte with input: 27-2033
```

Attachments

[GetProcessInputOutput.groovy.txt](#)

Work with Projects and Accounts

When working with projects and accounts, you can do the following:

- [Remove Information from a Project](#)
- [Add a New Project to the System with UDF/Custom Field Value](#)
- [Get a Project Name](#)
- [Find an Account Registered in the System](#)
- [Update Contact \(User and Client\) Information](#)

Remove Information from a Project

The following example shows you how to remove information from a project using Clarity LIMS and API (compatible with v2 r21 and later).

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

Prerequisites

Before you follow the example, make sure that you have the following items:

- A user-defined field (UDF)/custom field named Objective that is defined for projects.
- A project name that is unique and does not exist in the system.

Code Example

This example does the following actions:

1. POST a new project to Clarity LIMS, with a UDF/custom field value for Objective.
2. Remove a child XML node from the parent XML representing the project resource.
3. Update the project resource.

Step 1. POST a New Project with a UDF Value for Objective

First, set up the information required to perform a successful project POST. The project name must be unique.

```
// Determine project list's URI String projectsListURI =
"http://${hostname}/api/v2/projects" def builder = new StreamingMarkupBuilder()
builder.encoding = "UTF-8" openDate = "2017-08-24" // Build a new project using Markup
Builder def projectDoc = builder.bind { mkp.xmlDeclaration() mkp.declareNamespace(prj:
'http://genologics.com/ri/project') mkp.declareNamespace(udf:
'http://genologics.com/ri/userdefined') 'prj:project'{ 'name' (projectName) 'open-date'
(openDate) 'researcher' (uri:"http://${hostname}/api/v2/researchers/1") 'udf:field'
(name:"Objective","To test httpPOST") } } // Post the new project def projectNode =
GLSRestApiUtils.xmlStringToNode(projectDoc.toString()) projectNode =
GLSRestApiUtils.httpPOST(projectNode, projectsListURI, username, password) println
GLSRestApiUtils.nodeToXmlString(projectNode)
```

The projectNode should contain the response XML from the POST and resemble the following output:

```
<prj:project xmlns:prj="http://genologics.com/ri/project"
uri="http://yourIPAddress/api/v2/projects/WOR512" limsid="WOR512"> <name>httpPOST Project
then remove nodes</name> <open-date>2017-08-24</open-date> <researcher
uri="http://yourIPAddress/api/v2/researchers/1"/> <udf:field
xmlns:udf="http://genologics.com/ri/userdefined" type="String" name="Objective">To test
httpPOST</udf:field> <permissions
uri="http://yourIPAddress/api/v2/permissions/projects/WOR512"/></prj:project>
```

Step 2. Remove Child XML Node from Parent XML Node

The following code removes the child XML node `<udf:field>` from the parent XML node `<prj:project>`:

```
// If projectNode is parentNode, remove the child udf:field projectNode?.children
()?.remove(projectNode.'udf:field'[0]) // Print the node that will be updated println
GLSRestApiUtils.nodeToXmlString(projectNode)
```

If multiple nodes of the same type exist, [0] is the first item in this list of same typed nodes (eg, 0 contains 1st item, 1 contains 2nd item, 2 contains 3rd item, and so on).

To remove the 14th `udf:field`, you would use `projectNode?.children()?.remove`
(projectNode.'udf:field'[13])

Attachments

Add a New Project to the System with UDF/Custom Field Value

Imagine that you use projects in Clarity LIMS to track a collection of sample work that represents a subset of work from a larger translational research study. The translational research study consists of several projects within the LIMS and the information about each of the projects that make up the research study is predefined in another system.

Before the work starts in the lab, you can use the information in the other system to automatically create projects. This reduces errors and means that lab scientists do not have to spend time manually entering data a second time.

This example shows how to automate the creation of a project using a script and the projects resource POST method.

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

Prerequisites

Before you follow the example, make sure you have the following items:

- A user-defined field (UDF)/custom field named Objective that is defined for projects.
- A project name that is unique and does not exist in the system.
- A compatible version of API (v2 r21 or later).

Code Example

Before you can add a project to the system through the API, you must construct the XML representation for the project you want to create. You can then POST the new project resource.

Step 1. Define the Project

You can define the project XML using `StreamingMarkupBuilder`, a built-in Groovy data structure designed to build XML structures.

- Declare the project namespace because you are building a project.
- If you wish to include values for project UDFs/custom fields as part of the project XML you are constructing, then you must also declare the user-defined namespace.

In the following example, the project name, open date, researcher, and a UDF/custom field named Objective are included in the XML constructed for the project:

```
// Determine project list's URI String projectsListURI =
"http://${hostname}/api/v2/projects" def builder = new StreamingMarkupBuilder()
builder.encoding = "UTF-8" openDate = "2017-08-22" // Build a new project using Markup
Builder def projectDoc = builder.bind { mkp.xmlDeclaration() mkp.declareNamespace(prj:
'http://genologics.com/ri/project') mkp.declareNamespace(udf:
'http://genologics.com/ri/userdefined') 'prj:project'{ 'name' (projectName) 'open-date'
(openDate) 'researcher' (uri:"http://${hostname}/api/v2/researchers/1") 'udf:field'
(name:"Objective", "To test httpPOST") } }
```

UDFs / custom fields must be configured in Clarity LIMS before they can be set or updated using the API. You can find a list of the fields defined for a project in your system by using the resource (<http://youripaddress/api/v2/configuration/udfs>) and looking for those with an attach-to-name of project.

For Clarity LIMS v4.2 and earlier, UDTs can be handled in a similar approach. They must be configured in the Operations Interface before they can be set or updated using the API (with the `<udf:type>` tag). Project-related UDTs are only supported in the API and in the Operations Interface.

For Clarity LIMS v5 or later, UDTs are only supported in the API.

Step 2. Post the Project

For the POST to the projects resource to be successful, only project name and researcher URI are required. Adding more details is a good practice for keeping your system organized and understanding what must be accomplished for each project.

The following POST command adds a new project resource using the XML constructed by

StreamingMarkupBuilder:

```
// Turn the markup into a node and post it to the API def projectNode =
GLSRestApiUtils.xmlStringToNode(projectDoc.toString()) projectNode =
GLSRestApiUtils.httpPOST(projectNode, projectsListURI, username, password) println
GLSRestApiUtils.nodeToXmlString(projectNode)
```

Expected Output and Results

The XML returned after a successful POST of the XML built by StreamingMarkupBuilder is the same as the XML representation of the project:

```
<prj:project uri="http://yourIPAddress/api/v2/projects/ADM1101" limsid="ADM1101">
<name>httpPOST Project</name> <open-date>2017-08-22</open-date> <researcher
uri="http://yourIPAddress/api/v2/researchers/1"/> <udf:field
```

```
xmlns:udf="http://genologics.com/ri/userdefined" type="String" name="Objective">To test  
httpPOST</udf:field> <permissions  
uri="http://yourIPaddress/api/v2/permissions/projects/ADM1101"/></prj:project>
```

Attachments

[PostProject.groovy.txt](#)

Get a Project Name

Projects contain a collection of samples submitted to the lab for a specific goal or purpose. Often, a script needs information recorded at the project level to do its task. In this simple example, an HTTP GET against a project is shown to obtain information on the project in XML.

Prerequisites

Before you follow the example, make sure you have the following items:

- A project with named HTTP Get Project Name with GLS Utils.
- The LIMS ID of the project, referred to as `<project limsid>`.
- A compatible version of API (v2 r21 or later).

Code Example

The easiest way to find a project in the system is with its LIMS ID.

- If the project was created in the script (with an HTTP POST), then the LIMS ID is returned as part of the 201 response in the XML.
- If the LIMS ID is not available, but other information uniquely identifies it, you can use the project (list) resource to GET the projects and select the right LIMS ID from the collection.

Working with list resources generally requires the same script logic, so if you need the list of projects to find a specific project, then review the example found in [Find an Account Registered in the System](#). This example demonstrates listing and finding resources for labs, but the same logic applies.

Step 1. Determine the Project URI

The first step is to determine the URI of the project:

```
// Determine the project's URI URI projectURI = new URI  
("http://${hostname}/api/v2/projects/${projectID}")
```

Step 2. Retrieve the Project Resource

Next, use the project LIMS ID to perform an HTTP GET on the resource, and store the response XML in the variable named `projectNode`:

```
// Get a single Project by limsid projectNode = GLSRestApiUtils.httpGET
(projectURI.toString(), username, password) println GLSRestApiUtils.nodeToXmlString
(projectNode)
```

The `projectNode` variable can now be used to access XML elements and/or attributes.

Step 3. Get the Project Name

To get the project name, ask the `projectNode` for the text representation of the name element:

```
// Obtain the name println("Project ID : ${projectNode.name.text()}")
```

Expected Output and Results

The following example shows the expected output after obtaining the project name:

```
<prj:project xmlns:prj="http://genologics.com/ri/project"
uri="http://yourIpAddress/api/v2/projects/LUN3" limsid="LUN3"> <name>HTTP Get Project Name
with GLS Utils</name> <open-date>2017-05-31</open-date> <researcher
uri="http://yourIpAddress/api/v2/researchers/1"/> <permissions
uri="http://yourIpAddress/api/v2/permissions/projects/LUN3"/></prj:project>
```

Attachments

[GetProjectName.groovy.txt](#)

Find an Account Registered in the System

Imagine that each month the new external accounts with which your facility works are contacted with a Welcome package. In this scenario, it would be helpful to obtain a list of accounts that have been modified in the past month.

Accounts are not fully supported in Clarity LIMS v5. Support is planned for a future release. In v5, a workaround is to create an Account from the Projects and Samples screen.

 In Clarity LIMS v2.1, the term Labs was replaced with Accounts. However, the API resource is still called labs.

Prerequisites

Before you follow the example, make sure you have the following items:

- Several accounts that exist in the system.
- At least one account that was modified after a specific date.
- A compatible version of API (v2 r21 or later).

Code Example

In Clarity LIMS v4.2 and earlier, in the Operations Interface, the Accounts view lists the account resources available.

Accounts			
Name	Ship. Institution	Ship. City	Ship. Country
Administrative Lab			
Giant Facility			
University West		Seattle	USA
Universidad Espana			

To obtain a list of all accounts modified after a specific date, you can use a GET request on the accounts list resource and include the `?last-modified` filter.

To specify the last month, a Calendar object is instantiated. This Calendar object is initially set to the date and time of the call, rolled back one month, and then passed as a query parameter to the GET call. The following example returns all labs modified in the last month.

```
// Retrieve and format a date c = Calendar.getInstance() c.add(Calendar.MONTH, -1) df = new
SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ") time = df.format(c.getTime()).replace('+', '%2B')
// Filter labs by last-modified on specified date labsURI =
"http://${hostname}/api/v2/labs?last-modified=" + time labs = GLSRestApiUtils.httpGET
(labsURI, username, password) // For each lab, retrieve it labs.'lab'.each { labResource =
GLSRestApiUtils.httpGET(it.@uri, username, password) println "Welcome to the team,
${labResource.name.text()}" labState = labResource.'shipping-address'.state.text() // If lab
is in Washington, print message if (labState == "WA") { println "Go Huskies!" } }
```

The first GET call returns a list of the first 500 labs that meet the date modified criterion specified. The script iterates through each lab element to look at individual lab details. For each lab, a second GET method populates a lab resource XML node with address information.

The REST list resources are paged. Only the first 500 items are returned when you query for a list of items, (eg, `http://youripaddress/api/v2/artifacts`).

If you cannot filter the list, it is likely that you must iterate through the pages of a list resource to find the items that you are looking for. The URI for the next page of resources is always the last element on the page of a list resource.

Expected Output and Results

In the following example, the XML returned lists three out of the four labs, excluding one due to the date filter:

```
<lab:labs> <lab uri="http://yourIPaddress/api/v2/labs/2"> <name>Administrative Lab</name>
</lab> <lab uri="http://yourIPaddress/api/v2/labs/52"> <name>Giant Facility</name> </lab>
<lab uri="http://yourIPaddress/api/v2/labs/102"> <name>University West</name>
</lab></lab:labs>
```

One of the labs has 'WA' recorded as the state, adding a second printed line to the output:

```
Welcome to the team, Administrative Lab Welcome to the team, Giant Facility Welcome to the
team, University WestGo Huskies!
```

Attachments

[GetLab.groovy.txt](#)

Update Contact (User and Client) Information

The researcher resource holds the personal details for users and clients in Clarity LIMS.

Suppose that you have a separate system that maintains the contact details for your customers and collaborators. You could use this system to synchronize the details for researchers with the details in Clarity LIMS. This example shows how to update the phone number of a researcher using a PUT to the individual researcher resource.

In the Clarity LIMS user interface, the term Labs has been replaced with Accounts. However, the API resource is still called labs and the Collaborations Interface still refers to Labs rather than Accounts. The term Contact has been replaced with Client. The API resource is still called contact.

The LabLink Collaborations Interface is not supported in Clarity LIMS v5 and later. However, because support for this interface is planned for a future release, the Collaborator user role has not been removed.

Prerequisites

Before you follow the example, make sure you have the following items:

- A defined client in Clarity LIMS.
- A compatible version of API (v2 r21 or later).

Code Example

For Clarity LIMS v5 and later, in the web interface, the User and Clients screen lists all users and clients in the system.

The screenshot displays the 'USERS & CLIENTS' section of the Clarity LIMS web interface. At the top, there are tabs for 'LAB WORK', 'CONSUMABLES', 'CUSTOM FIELDS', 'USERS & CLIENTS', and 'AUTOMATION'. Below these, there are sub-tabs for 'Users' and 'Clients'. The 'USERS' tab is active, showing a list of users. The 'USER DETAILS' section on the right shows the details for Sue Erikson.

USERS	USER DETAILS
- Patricia Woo - Paul Mitchum Sue Erikson - System Administrator - Tom Lee	NEW USER + Delete Cancel Save Sue Erikson Title Account EriksonNY Labs > Email serikson@eriksonlabsny.com Username serikson Phone 2122122130 Fax 2122122131

For Clarity LIMS v4.2 and earlier, in the Lab Scientist interface, the User and Clients screen lists all users and clients in the system. Also, in the Operations Interface, the Contacts view lists the users and clients in the system.

Step 1. Retrieve the Researcher Resource

In the API, information for a particular researcher can be retrieved within a script using a GET call:

```
// Retrieve the researcher Node researcher = GLSRestApiUtils.httpGET(researcherURI,
username, password)
```

In this case, the URI represents the individual researcher resource for the researcher named Sue Erikson. The GET returns an XML representation of the researcher, which populates the groovy node researcher.

 The XML representations of individual REST resources are self-contained entities. Always request the complete XML representation before editing any portion of the XML. If you do not use the complete XML when you update the resource, you can inadvertently change data.

The following example shows the XML returned for the Sue Erikson researcher:

```
<res:researcher uri="http://yourIPaddress/api/v2/researchers/103"> <first-
name>Sue</first-name> <last-name>Erikson</last-name> <phone>212-212-2130</phone>
<fax>212-212-2131</fax> <email>serikson@eriksonlabsny.com</email> <lab
uri="http://yourIPaddress/api/v2/labs/102"/> <credentials> <username>serikson</username>
<account-locked>>false</account-locked> <role roleName="webclient"/> <role
roleName="administrator"/> <role roleName="labtech"/> </credentials>
<initials>SEK</initials></res:researcher>
```

Step 2. Update the Researcher Information

Updating the telephone number requires the following steps:

1. Changing the telephone value in the XML.
2. Using a PUT call to update the researcher resource.

The new telephone number for Sue Erikson can be set with the phone value within the Groovy researcher node:

```
newPhoneNumber = '212-212-2133' // Set the new phone number researcher.'phone'
[0].setValue(newPhoneNumber) Node returnNode = GLSRestApiUtils.httpPUT(researcher,
researcher.@uri, username, password) println GLSRestApiUtils.nodeToXmlString(returnNode)
```

The PUT command updates the research resource at the specified URI using the complete XML representation, including the new phone number. A successful PUT returns the new XML in the `returnNode`. An unsuccessful PUT returns the http response code and error message in XML in the `returnNode`.

Expected Output and Results

For a successful update, the resulting XML can also be reviewed in a web browser through the URI at <http://yourIPaddress/api/researchers/103>.

```
<res:researcher uri="http://yourIPaddress/api/v2/researchers/103"> <first-name>Sue</first-name> <last-name>Erikson</last-name> <phone>212-212-2133</phone> <fax>212-212-2131</fax> <email>serikson@eriksonlabsny.com</email> <lab uri="http://yourIPaddress/api/v2/labs/102"/> <credentials> <username>serikson</username> <account-locked>false</account-locked> <role roleName="webclient"/> <role roleName="administrator"/> <role roleName="labtech"/> </credentials> <initials>SEK</initials></res:researcher>
```

In Clarity LIMS, the updated user list shows the new phone number.

Attachments

[UpdateContactInfo.groovy.txt](#)

Work with Multiplexing

When working with multiplexing, you can do the following:

- [Find the Index Sequence for a Reagent Label](#)
- [Demultiplexing](#)
- [Pool Samples with Reagent Labels](#)
- [Apply Reagent Labels with REST](#)
- [Apply Reagent Labels When Samples are Imported](#)
- [Apply Reagent Labels by Adding Reagents to Samples](#)

Find the Index Sequence for a Reagent Label

A common requirement in applications involving indexed sequencing is to determine the sequence corresponding to a reagent label. This example shows how to configure index reagent types, which you can then use to find the sequence for a reagent label. Before you follow the example, make sure that you have a compatible version of API (v2 r14 to v2 r24).

Code Example

Reagents and reagent labels are independent concepts in the API. However, the recommended practice is to name reagent labels after reagent types. This allows you to use the label name to look up the sequence information on the reagent type resource. This practice is consistent with the Operations Interface process wizards. When a reagent is applied to a sample in the user interface, a reagent label with the same name of the reagent type is added to the analyte resource.

The following actions are also recommended:

1. Configure an index reagent type with the correct sequence for each type of index or tag you plan to use.
2. Use the names of the index reagent types as reagent labels.

Following these practices allows you to find the sequence for a reagent label by looking up the sequence in the corresponding reagent type.

Step 1. Configure Index Reagent Types

For each index or tag you plan to use in indexed sequencing, configure a corresponding index reagent type as follows.

1. As administrator, in the Clarity LIMS Operations Interface, select **Configuration > Reagents > Reagent Type**.
2. Add a new reagent type named Index 1.
3. Select **Index** for both the Reagent Category Name and the Special Type.
4. In the Sequence field, enter the index sequence specified by the manufacturer (eg, ATCACG).
5. Repeat these steps for each index you plan to use.

You can now use names such as Index 1 and Index 2 when applying labels to artifacts.

Step 2. Retrieve the Sequence for a Reagent Label with the REST API

After you have configured reagent types for each indexing sequence you intend to use, and have used those reagent type names as reagent label names, you can easily retrieve the corresponding sequence using the REST API.

The following code snippet shows how to retrieve the index sequences (when available):

```
// Determine the URI of the labeled artifact and retrieve it labeledArtifactURI =
"http://${hostname}/api/v2/artifacts/${labeledArtifactLIMSID}" labeledArtifact =
GLSRestApiUtils.httpGET(labeledArtifactURI, username, password) // Gather its reagent
labels reagentLabels = labeledArtifact.'reagent-label'.@name if (!reagentLabels) {
println "No labels found" return } // Build a query URI for possibly multiple reagent
labels and execute it queryParameters = reagentLabels.collect { "name=${it.replace(' ',
'+')}" }.join('and') reagentTypesURI = "http://${hostname}/api/v2/reagenttypes/"
reagentTypeQueryURI = reagentTypesURI + '?' + queryParameters reagentTypeLinks =
GLSRestApiUtils.httpGET(reagentTypeQueryURI, username, password) // For each reagent type
found, retrieve it reagentTypeLinks.'reagent-type'.@uri.each { reagentType =
GLSRestApiUtils.httpGET(it, username, password) reagentTypeName = reagentType.@name
reagentLabels.remove(reagentTypeName) index = reagentType.'special-type'.'attribute'.find
{ it.@name = 'Sequence' }?.@value // Output the result println "Label: $reagentTypeName"
println "Index: $index" } // If there are reagent labels that found no matches if
(reagentLabels) { unmatchedLabels = reagentLabels.join(',') println "No reagent types
found for labels: $unmatchedLabels" }
```

For an artifact labeled with Index 1, this would produce the following information:

```
Label: Index 1 Index: ATCACG
```

Attachments

[RetrievingReagentLabelIndex.groovy.txt](#)

Demultiplexing

Demultiplexing is the last step in an indexed sequencing workflow. While the specifics depend on the sequencing instrument and analysis software used, taking pooled samples through sequencing and analysis produces result files/metrics per lane/identifier tag.

These results will likely be in the form of multiple files that you can import back into Clarity LIMS. To do this, you need to set up a configured process that generates process outputs that apply to inputs per reagent label, usually in the form of `ResultFile` artifacts.

Prerequisites

Before you follow the example, make sure you have the following items:

- Configured reagent types named Index 1 through Index 6 in Clarity LIMS.
- Reagents of type Index 1 through Index 6 in Clarity LIMS.
- A compatible version of API (v2 r14 to v2 r24).

Code Example

Create a Demultiplexing Process in the Clarity LIMS Operations Interface

Configure a process that generates `ResultFile` with process outputs that apply to inputs per reagent label. It is recommended to name your outputs in a way that clearly identifies the samples to which they correspond (eg, `Results for {SubmittedSampleName}-{AppliedReagentLabels}`).

GLS

Modify Process Type

Name: Demultiplex

Abbreviation: DEM

☐ Open the process tab after the process wizard completes

Icon: 

☒ Display in Explorer Tree ☒ Add to Button Bar

Output Details

Output Generation

Attributes

External Programs

Input Types

Output Types

Process Outputs to apply to individual Inputs

	Assign Working Status	Artifact	Display Name	
☐	☐	Analyte	Analyte	
☐		Gel Spot	Gel Spot	
☐		ResultFile	ResultFile	
☐		SearchResultFile	SearchResultFile	

0 rows selected

Shared Outputs that apply to all Inputs

	Assign Working Status	Artifact	Display Name	
☐		Gel 1D	Gel 1D	
☐		Gel 2D	Gel 2D	
☐		ResultFile	ResultFile	
☐		SearchResultFile	SearchResultFile	

0 rows selected

Process Outputs that apply to Inputs per Reagent Label

	Assign Working Status	Artifact	Display Name	
☒		ResultFile	ResultFile	
☐		SearchResultFile	SearchResultFile	

0 rows selected

Help

Save

Cancel

GLS Modify Process Type

Name:

Abbreviation: ☐ Open the process tab after the process wizard completes

Icon:  ☒ Display in Explorer Tree ☒ Add to Button Bar

Input Types		Output Types	
Output Details	Output Generation	Attributes	External Programs

Output Abbreviation:

Generate Outputs of Type: ▼

Number of Outputs to generate per Input per Reagent Label:

☒ Fixed number of Outputs applied to all Inputs (pre-configured)

☐ Variable number of Outputs applied to all Inputs (entered at run-time)

☐ Variable number of Outputs per Input (entered at run-time)

Define the naming pattern for output names (right-click for a list of available variables):

Assuming 3 inputs, the naming pattern will generate the following output names:

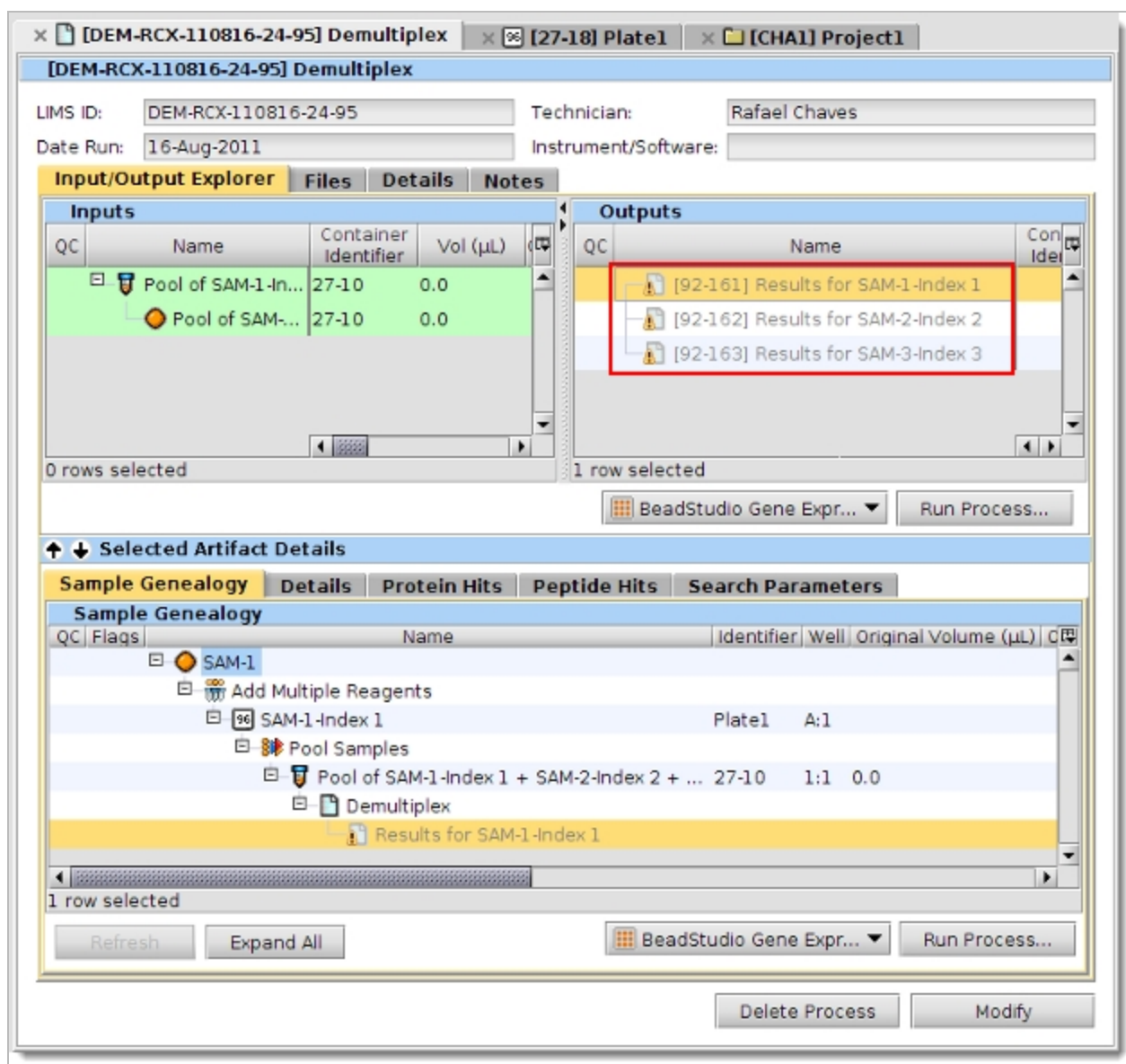
Results for SUBMITTED_SAMPLE_NAME-REAGENT_LABELS

Demultiplexing in the User Interface

Running the demultiplexing process on a labeled pooled input produces a process run in the Operations Interface, similar to the one illustrated below.

Note the following:

- There were three reagent labels in the input analyte (sample) artifact. As a result, three outputs were generated (the process was configured to produce one output result file per label per input).
- The names of the outputs of the demultiplexing process expose the original sample name and label.
- The Operations Interface shows details of the genealogy from the downstream result file all the way back to the original sample.



While reagent labels are not explicitly exposed in the Clarity LIMS client user interface, genealogy views in the Operations Interface are aware of reagent labels and will show the true sample inheritance. As noted above, you can use the `{AppliedReagentLabels}` output naming variable to show the reagent labels applied to each artifact in the user interface.

Demultiplexing in the API with a Process POST

Executing a demultiplexing process by issuing a process POST via the REST API is similar to the typical process execution found in [Run a Process/Step](#).

The key difference is that when executing a demultiplexing process through the REST API, outputs per reagent label are automatically generated from the inputs provided. You do not need to explicitly specify them.

For example, when running the demultiplexing process configured against a single (pooled) sample, you could post a process execution representation like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><prx:process
xmlns:prx="http://genologics.com/ri/processexecution"> <type>Demultiplex</type>
<technician uri="http://yourIPaddress/api/v2/researchers/4"/> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/2-424"/> </input-output-map></prx:process>
```

The input-output-map only refers to inputs, not outputs, because the demultiplexing process is configured to exclusively produce outputs per reagent label.

If your process produces other outputs, such as shared or per-input outputs, you must explicitly specify input-output-maps for them.

Verify with the REST API

Irrespective of whether you use the user interface or the REST API to run the demultiplexing process, the REST API representation for the process looks something like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><prc:process
xmlns:udf="http://genologics.com/ri/userdefined"
xmlns:file="http://genologics.com/ri/file" xmlns:prc="http://genologics.com/ri/process"
uri="http://yourIPaddress/api/v2/processes/DMX-RCX-110816-24-95" limsid="DMX-RCX-110816-
24-95"> <type uri="http://yourIPaddress/api/v2/processtypes/38">Demultiplex</type> <date-
run>2011-08-12</date-run> <technician uri="http://yourIPaddress/api/v2/researchers/4">
<first-name>RC</first-name> <last-name>RC</last-name> </technician> <input-output-map>
<input post-process-uri="http://yourIPaddress/api/v2/artifacts/2-424?state=335"
uri="http://yourIPaddress/api/v2/artifacts/2-424?state=324" limsid="2-424"> <parent-
process uri="http://yourIPaddress/api/v2/processes/PSA-RCX-110812-122-221" limsid="PSA-
RCX-110812-122-221" /> </input> <output uri="http://yourIPaddress/api/v2/artifacts/92-
163?state=339" output-generation-type="PerReagentLabel" output-type="ResultFile"
limsid="92-163" /> </input-output-map> <input-output-map> <input post-process-
uri="http://yourIPaddress/api/v2/artifacts/2-424?state=335"
uri="http://yourIPaddress/api/v2/artifacts/2-424?state=324" limsid="2-424"> <parent-
process uri="http://yourIPaddress/api/v2/processes/PSA-RCX-110812-122-221" limsid="PSA-
RCX-110812-122-221" /> </input> <output uri="http://yourIPaddress/api/v2/artifacts/92-
164?state=336" output-generation-type="PerReagentLabel" output-type="ResultFile"
limsid="92-164" /> </input-output-map> <input-output-map> <input post-process-
uri="http://yourIPaddress/api/v2/artifacts/2-424?state=335"
uri="http://yourIPaddress/api/v2/artifacts/2-424?state=324" limsid="2-424"> <parent-
process uri="http://yourIPaddress/api/v2/processes/PSA-RCX-110812-122-221" limsid="PSA-
RCX-110812-122-221" /> </input> <output uri="http://yourIPaddress/api/v2/artifacts/92-
162?state=338" output-generation-type="PerReagentLabel" output-type="ResultFile"
limsid="92-162" /> </input-output-map></prc:process>
```

For each input with reagent labels, one output was created per reagent label.

In the example, the process ran on one pooled input, and produced three outputs (the pooled input included three reagent labels). The following example shows one of the demultiplexed result file outputs:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><art:artifact
xmlns:udf="http://genologics.com/ri/userdefined"
xmlns:file="http://genologics.com/ri/file" xmlns:art="http://genologics.com/ri/artifact"
uri="http://yourIPaddress/api/v2/artifacts/92-163?state=339" limsid="92-163">
<name>Results for SAM-3 - Index 3</name> <type>ResultFile</type> <output-
type>ResultFile</output-type> <parent-process
uri="http://yourIPaddress/api/v2/processes/DMX-RCX-110816-24-95" limsid="DMX-RCX-110816-
24-95" /> <qc-flag>UNKNOWN</qc-flag> <sample
uri="http://yourIPaddress/api/v2/samples/RCY1A104" limsid="RCY1A104" /> <reagent-label
name="Index 3" /></art:artifact>
```

The output contains only one reagent label, and relates only to the sample that was tagged with the same reagent label. Compare this to [the case of a pooled artifact](#), which has several labels and relates to several samples. This level of traceability (from a demultiplexed output back to its specific original sample) is only possible because the artifacts were labeled before they were pooled.

The artifact name generated by the demultiplexing process output name pattern is ("Results for SAM-3 - Index 3"). You can use the {SubmittedSampleName} naming variable to show true ancestors, and the {AppliedReagentLabels} to show any reagent labels applied to an output.

Pool Samples with Reagent Labels

Before pooling samples in a multiplexed workflow, apply reagent labels using one of the methods described in [Work with Multiplexing](#). After the analyte (derived sample) artifacts are labeled, they can be pooled together without loss of traceability.

Pooling samples is accomplished either by running a pooling step in the user interface, or by using the process resource in the REST API.

For an overview of how REST resources are structured, and to learn how the process resource is used to track workflow in Clarity LIMS, refer to [REST General Concepts](#) and [Structure of REST Resources](#).

Prerequisites

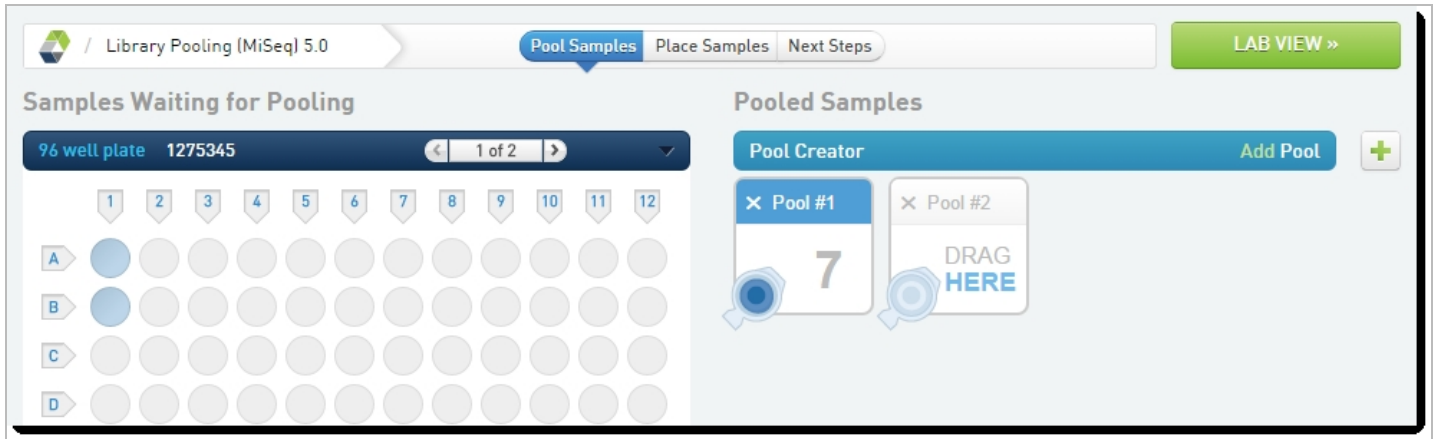
Before you follow the example, make sure that you have the following items:

- Reagent types that are configured in Clarity LIMS and are named index 1 through index 6 .
- Reagents of type index 1 through index 6 that have been added to Clarity LIMS.
- A compatible version of API (v2 r21 or later).

Code Example

Pool in the User Interface

The following screenshot shows a pooling step run from Clarity LIMS.



Pool in the API with a Process POST

Pooling samples in the API is accomplished with a process resource. Information about a step is also stored in the process resource. Such a process has many input samples that map to a shared output sample, such that the shared output is a pool of those inputs. This is achieved with a [process POST](#), where a single input-output-map element in the XML defines the shared output and all its related inputs.

In general, automation scripts access information about a step using the processURI, which links to the individual process resource. The input-output-map in the XML returned by the individual process resource gives the script access to the artifacts that were inputs and outputs to the process.

Information about a derived sample is stored in the analyte resource. This is used as the input and output of a step, and also used to record specific details from lab processing. The XML representation for an individual analyte contains a link to the URI of its submitted sample, and to the URI of the process that generated it (parent process).

The following example pools all samples found in a given container into a tube it creates.

i | No special code is required to handle reagent labels. As processes execute, reagent labels automatically flow from inputs to outputs.

```
// Create the required URIs containerToPoolUri =  
"http://${hostname}/api/v2/containers/${containerLIMSID}" researcherURI =  
"http://${hostname}/api/v2/researchers/1" // Find the artifacts to pool in the given  
container containerToPool = GLSRestApiUtils.httpGET(containerToPoolUri, username,  
password) artifactURIsToPool = containerToPool.'placement'.@uri def builder = new  
StreamingMarkupBuilder() builder.encoding = "UTF-8" // Create a new container using the  
Markup Builder def containerDoc = builder.bind { mkp.xmlDeclaration()  
mkp.declareNamespace(con: 'http://genologics.com/ri/container') mkp.declareNamespace(udf:
```

```
'http://genologics.com/ri/userdefined') 'con:container'{ 'name'("Pooled contents from
${containerToPool.name.text()})" 'type'(uri:"http://${hostname}/api/v2/containerTypes/2",
name:"Tube") } } // Post the new container to the API containerNode =
GLSRestApiUtils.xmlStringToNode(containerDoc.toString()) container =
GLSRestApiUtils.httpPOST(containerNode, "http://${hostname}/api/v2/containers", username,
password) // Create a new Pool Samples process using the StreamingMarkupBuilder
processDoc = new StreamingMarkupBuilder().bind { mkp.declareNamespace(prx:
'http://genologics.com/ri/processesexecution') 'prx:process'{ 'type'('Pool Samples')
'technician'(uri:researcherURI) 'input-output-map'(shared:'true') {
artifactURIsToPool.each { 'input'(uri:it) } 'output'(type:'Analyte') { 'location' {
'container'(uri:container.@uri) 'value'('1:1') } } } } } // Post the Pool Samples process
to the API processNode = GLSRestApiUtils.xmlStringToNode(processDoc.toString())
returnNode = GLSRestApiUtils.httpPOST(processNode, "http://${hostname}/api/v2/processes",
username, password) println GLSRestApiUtils.nodeToXmlString(returnNode)
```

Verify with the REST API

Irrespective of whether you use the user interface or the REST API to pool samples, the pooled sample is available via process GET requests.

The following example shows one pooled output (LIMS ID 2-424) created from three inputs - LIMS IDs RCY1A103PA1, RCY1A104PA1, and RCY1A105PA1:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><prc:process
xmlns:udf="http://genologics.com/ri/userdefined"
xmlns:file="http://genologics.com/ri/file" xmlns:prc="http://genologics.com/ri/process"
uri="http://yourIPAddress/api/v2/processes/PSA-RCX-110812-122-221" limsid="PSA-RCX-
110812-122-221"> <type uri="http://yourIPAddress/api/v2/processtypes/37">Pool
Samples</type> <date-run>2011-08-12</date-run> <technician
uri="http://yourIPAddress/api/v2/researchers/4"> <first-name>RC</first-name> <last-
name>RC</last-name> </technician> <input-output-map> <input post-process-
uri="http://yourIPAddress/api/v2/artifacts/RCY1A104PA1?state=319"
uri="http://yourIPAddress/api/v2/artifacts/RCY1A104PA1?state=315" limsid="RCY1A104PA1" />
<output uri="http://yourIPAddress/api/v2/artifacts/2-424?state=316" output-generation-
type="PerAllInputs" output-type="Sample" limsid="2-424" /> </input-output-map> <input-
output-map> <input post-process-
uri="http://yourIPAddress/api/v2/artifacts/RCY1A105PA1?state=317"
uri="http://yourIPAddress/api/v2/artifacts/RCY1A105PA1?state=314" limsid="RCY1A105PA1" />
<output uri="http://yourIPAddress/api/v2/artifacts/2-424?state=316" output-generation-
type="PerAllInputs" output-type="Sample" limsid="2-424" /> </input-output-map> <input-
output-map> <input post-process-
uri="http://yourIPAddress/api/v2/artifacts/RCY1A103PA1?state=318"
uri="http://yourIPAddress/api/v2/artifacts/RCY1A103PA1?state=310" limsid="RCY1A103PA1" />
```

```
<output uri="http://yourIPaddress/api/v2/artifacts/2-424?state=316" output-generation-
type="PerAllInputs" output-type="Sample" limsid="2-424" /> </input-output-
map></prc:process>
```

Besides deriving from the ancestral sample artifacts, the resulting pooled sample artifact inherits the reagent labels from all inputs. The pooled output produced by the pooling step appears as follows. The pooled artifact shows multiple reagent labels, and multiple ancestor samples.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><art:artifact
xmlns:udf="http://genologics.com/ri/userdefined"
xmlns:file="http://genologics.com/ri/file" xmlns:art="http://genologics.com/ri/artifact"
uri="http://yourIPaddress/api/v2/artifacts/2-424?state=316" limsid="2-424"> <name>Pool of
SAM-1 - Index 1 + SAM-2 - Index 2 + SAM-3 - Index 3</name> <type>Analyte</type> <output-
type>Sample</output-type> <parent-process uri="http://yourIPaddress/api/v2/processes/PSA-
RCX-110812-122-221" limsid="PSA-RCX-110812-122-221" /> <volume unit="uL">0.0</volume>
<qc-flag>UNKNOWN</qc-flag> <location> <container
uri="http://yourIPaddress/api/v2/containers/27-16" limsid="27-16" /> <value>1:1</value>
</location> <working-flag>true</working-flag> <sample
uri="http://yourIPaddress/api/v2/samples/RCY1A104" limsid="RCY1A104" /> <sample
uri="http://yourIPaddress/api/v2/samples/RCY1A105" limsid="RCY1A105" /> <sample
uri="http://yourIPaddress/api/v2/samples/RCY1A103" limsid="RCY1A103" /> <reagent-label
name="Index 1" /> <reagent-label name="Index 2" /> <reagent-label name="Index 3"
/></art:artifact>
```

As processes are executed, reagent labels flow from inputs to outputs.

Attachments

[PoolingSamplesWithReagents.groovy.txt](#)

Apply Reagent Labels with REST

Reagent labels are artifact resource elements and can be applied using a PUT. To apply a reagent label to an artifact using REST, the following steps are required:

1. GET the artifact representation.
2. Insert a reagent-label element with the intended label name.
3. PUT the modified artifact representation back.

You can apply the reagent label to the original analyte (sample) artifact or to a downstream sample or result file.

Prerequisites

Before you follow the example, make sure that you have the following items:

- Reagent types that are configured in Clarity LIMS and are named index 1 through index 6.
- Reagents of type index 1 through index 6 that have been added to Clarity LIMS.

- A compatible version of API (v2 r14 to v2 r24).

Code Example

In this example, you can adjust the following code:

```
def toLabel = GLSRestApiUtils.httpGET(artifactToLabelURI, username, password) println '***
Before labeling' println GLSRestApiUtils.nodeToXmlString(toLabel) println new Node(toLabel,
'reagent-label', [name: 'Index 1']) println '*** After labeling' println
GLSRestApiUtils.nodeToXmlString(toLabel) println GLSRestApiUtils.httpPUT(toLabel,
toLabel.@uri, username, password)
```

By inserting the `reagent-label` element, you end up with the following code. The change is highlighted in red.

```
*** Before labeling <?xml version="1.0" encoding="UTF-8" standalone="yes"?><art:artifact
xmlns:udf="http://genologics.com/ri/userdefined" xmlns:file="http://genologics.com/ri/file"
xmlns:art="http://genologics.com/ri/artifact"
uri="http://yourIPAddress/api/v2/artifacts/RCY1A97PA1?state=301" limsid="RCY1A97PA1">
<name>Sample-1</name> <type>Analyte</type> <output-type>Analyte</output-type> <qc-
flag>UNKNOWN</qc-flag> <location> <container uri="http://yourIPAddress/api/v2/containers/27-
12" limsid="27-12" /> <value>A:1</value> </location> <working-flag>true</working-flag>
<sample uri="http://yourIPAddress/api/v2/samples/RCY1A97" limsid="RCY1A97" /></art:artifact>
*** After labeling <?xml version="1.0" encoding="UTF-8" standalone="yes"?><art:artifact
xmlns:udf="http://genologics.com/ri/userdefined" xmlns:file="http://genologics.com/ri/file"
xmlns:art="http://genologics.com/ri/artifact"
uri="http://yourIPAddress/api/v2/artifacts/RCY1A97PA1?state=301" limsid="RCY1A97PA1">
<name>Sample-1</name> <type>Analyte</type> <output-type>Analyte</output-type> <qc-
flag>UNKNOWN</qc-flag> <location> <container uri="http://yourIPAddress/api/v2/containers/27-
12" limsid="27-12" /> <value>A:1</value> </location> <working-flag>true</working-flag>
<sample uri="http://yourIPAddress/api/v2/samples/RCY1A97" limsid="RCY1A97" /> <reagent-label
name="Index 1" /> </art:artifact>
```

Although it is not mandatory, it is recommended that you name reagent labels after reagent types using the Index special type. This allows you to relate the reagent label back to its sequence.

Apply Reagent Labels When Samples are Imported

When importing sample data into Clarity LIMS using a spreadsheet, you can specify the reagent labels to be applied during the import process. To do this, you must include the reagent label names in the spreadsheet, in a column named Sample/Reagent Label.

Prerequisites

Before you follow the example, make sure you have the following items:

- Reagent types that are configured in Clarity LIMS and are named index 1 through index 6.
- Reagents of type index 1 through index 6 that have been added to Clarity LIMS.
- A compatible version of API (v2 r14 to v2 r24).

Code Example

The following example spreadsheet would import six samples into the system. These samples are Sample-1 through Sample-6 with reagent labels Index 1 through Index 6:

**<TABLE
HEADER>**

Sample/Name	Container/Type	Container/Name	Sample/Well Location	Sample/Reagent Label
-------------	----------------	----------------	-------------------------	-------------------------

**</TABLE
HEADER>**

**<SAMPLE
ENTRIES>**

Sample-1	96 well plate	labeled-samples	A:1	Index 1
Sample-2	96 well plate	labeled-samples	A:2	Index 2
Sample-3	96 well plate	labeled-samples	A:3	Index 3
Sample-4	96 well plate	labeled-samples	A:4	Index 4
Sample-5	96 well plate	labeled-samples	A:5	Index 5
Sample-6	96 well plate	labeled-samples	A:6	Index 6

**</SAMPLE
ENTRIES>**

Although not mandatory, it is recommended that you name reagent labels after reagent types using the Index special type. This allows you to relate the reagent label back to its sequence.

Verify with the REST API

If you examine the REST API representation of the samples imported, you are able to verify the following:

- The sample representation shows no indication that reagent labels were applied.
- The sample artifact (the analyte artifact linked from the sample representation) indicates the label applied via the `<reagent-label>` element.

The following example shows how an imported sample artifact (Sample-1), with reagent label name applied (Index 1), appears when verified via the REST API:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><art:artifact
xmlns:udf="http://genologics.com/ri/userdefined"
xmlns:file="http://genologics.com/ri/file" xmlns:art="http://genologics.com/ri/artifact"
```



```
uri="http://yourIPaddress/api/v2/artifacts/RCY1A97PA1?state=301" limsid="RCY1A97PA1">
<name>Sample-1</name> <type>Analyte</type> <output-type>Analyte</output-type> <qc-
flag>UNKNOWN</qc-flag> <location> <container
uri="http://yourIPaddress/api/v2/containers/27-12" limsid="27-12" /> <value>A:1</value>
</location> <working-flag>true</working-flag> <sample
uri="http://yourIPaddress/api/v2/samples/RCY1A97" limsid="RCY1A97" /> <reagent-label
name="Index 1" /></art:artifact>
```

Apply Reagent Labels by Adding Reagents to Samples

If your samples are already in Clarity LIMS, you can assign reagent labels by running the Add Multiple Reagents process/protocol step from the Clarity LIMS user interface. Adding a reagent implicitly assigns a reagent label to every sample artifact. The reagent label applied is derived from the reagent type used.

Prerequisites

Before you follow the example, make sure that you have the following items:

- Reagent types that are configured in Clarity LIMS and are named index 1 through index 6.
- Reagents of type index 1 through index 6 that have been added to Clarity LIMS.
- A compatible version of API (v2 r14 to v2 r24).

For more information on indexes with reagent labels, refer to [Find the Index Sequence for a Reagent Label](#).

Code Example

The following illustrations show the Add Multiple Reagents process, as run from the Operations Interface.

Run the Add Multiple Reagents Process

In the Add Multiple Reagents wizard panel, reagents (Indexes 1 to 3) are selected and then assigned to the samples (SAM-1 to 3) in the Sample Workspace using a click-and-drag process.

The cells of the Sample Workspace represent the wells of the container used for this process.

Add Multiple Reagents

- On the Samples and Reagents tabs, drag samples and reagents onto wells in the Sample Workspace.
- Adjust the Sample Workspace to reflect the dimensions of the container you are working with.
- For more information, click the Help button.

SamplesReagents

Select Reagent to add

Reagent Type: Index 1, Index 2, Index 3Find

Reagent Source: 27-22 (Index 3)Find

Selected Reagents

Container	Name	Volume (µL)
27-19	Index 1	5.0
27-20	Index 2	5.0
27-22	Index 3	5.0

3 rows selected

Sample Workspace

	1	2	3	4	5	6	7	8
A	SAM-1	SAM-2	SAM-3					
B								
C								
D								
E								

5

 Rows

8

 Columns ☒ Place in rows ☐ Place in columns

Maximum number of reagents per well:

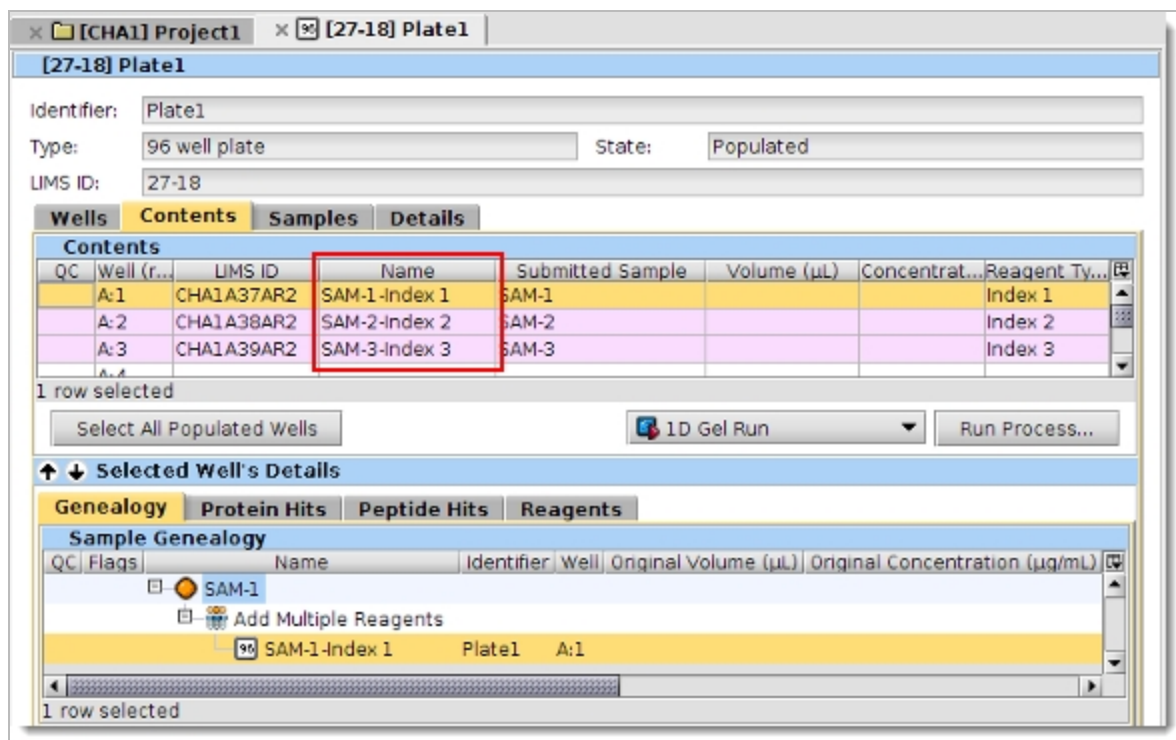
1

Clear Selected Cells

Clear All Cells

When the wizard completes, the Add Multiple Reagents process replaces the input sample artifacts with output analyte artifacts.

In the following illustration, the Name column shows the reagent labels applied to the outputs. These are generated by the default output naming pattern for the Add Multiple Reagents process: {InputItemName}-{AppliedReagentLabels}.



Verify with the REST API

When running the Add Multiple Reagents process, the output analyte artifact names show the reagent label applied, as the output naming pattern in the process configuration uses the `{AppliedReagentLabels}` variable. By examining the REST API representation of the Add Multiple Reagents process, you can verify the following information:

- The output analyte artifacts show a reagent-label element matching the name of the reagent type used.
- The input analyte artifacts are not modified and do not have reagent labels added.
- The input analyte artifacts do not have a location element, as they were displaced by the outputs.
- You can only determine that reagent labels were applied. You cannot determine which reagent was applied.

The following code shows an example of an output from an Add Multiple Reagents process when viewed with the REST API:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><art:artifact
xmlns:udf="http://genologics.com/ri/userdefined"
xmlns:file="http://genologics.com/ri/file" xmlns:art="http://genologics.com/ri/artifact"
uri="http://yourIPaddress/api/v2/artifacts/RCY1A97PA1?state=301" limsid="RCY1A97PA1">
<name>SAM-1-Index 1</name> <type>Analyte</type> <output-type>Analyte</output-type>
<parent-process uri="http://yourIPaddress/api/v2/processes/AMR-RCX-110819-151-25"
limsid="AMR-RCX-110819-151-25"/> <qc-flag>UNKNOWN</qc-flag> <location> <container
```

```
uri="http://yourIPaddress/api/v2/containers/27-12" limsid="27-12" /> <value>A:1</value>
</location> <working-flag>true</working-flag> <sample
uri="http://yourIPaddress/api/v2/samples/RCY1A97" limsid="RCY1A97" /> <reagent-label
name="Index 1" /></art:artifact>
```

Although adding a reagent to a sample automatically assigns a reagent label, reagents and reagent labels are independent concepts in Clarity LIMS. There are ways to add reagent labels that do not involve reagents, and that even when using reagents, it is not possible to accurately determine the reagent used based on the reagent label attached to an artifact.

Working with User Defined Fields/Custom Fields

[About UDFs/Custom Fields and UDTs](#)

[Performing Post-Step Calculations with Custom Fields/UDFs](#)

About UDFs/Custom Fields and UDTs

In the BaseSpace Clarity LIMS web interface, in the Custom Fields configuration screen, administrators can add user-defined information by adding custom fields (global fields or master step fields). At this time, user-defined types (UDTs) are only supported in the API.

Use these custom fields to configure storage locations for data that annotates project, submitted sample, step, derived sample, measurement, and file information recorded in a workflow.

All XML element attributes and values are text. Before using the value in a script, you may want to convert to a strongly-typed variable such as a number or date type.

For details on the formats used in XML, see [Working with User-Defined Fields \(UDF\) and Types \(UDT\)](#).



When updating multiple process outputs or containers, you can increase the script execution speed by using batch operations.

Performing Post-Step Calculations with Custom Fields/UDFs

Compatibility: API version 2 revision 21 and later

Important measurements and values are often calculated from other values. Instead of performing these calculations by hand, and then manually entering them into the LIMS (thereby increasing the probability of error), you can develop scripts to perform these calculations and update the data accordingly.

This example demonstrates the use of scripts and user-defined fields (UDFs) / custom fields for information retrieval and recording of calculation results in the LIMS.

Note:

- Information about a step is stored in the **process** resource in the API.
- Information about a derived sample is stored in the **analyte** resource in the API. This resource is used as the input and output of a step, and also used to record specific details from lab processing.
- As of BaseSpace Clarity LIMS v5.0, the term user-defined field (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called **udf**.

Prerequisites

LIMS v4.2 and earlier:

1. You have defined the following UDFs on **Analyte**:
 - **Concentration**
 - **Size (bp)**
 - **Conc. nM**
2. You have set the three fields configured in step 1 to display in the **Sample** table of the **Record Details** screen.
3. You have configured a **Calc. Prep** process to apply **Concentration, Size (bp)** to generated derived samples.
 - You have run the **Calc. Prep** step and it has generated derived samples.
 - You have input values for the **Calculation** and **Size (bp)** UDFs.
4. You have configured a **Calculation** process to apply **Conc. nM** to generated derived samples (analytes).
 - You have run the **Calculation** step - with the derived samples generated by the **Calc. Prep** step as inputs, and it has generated derived samples.

LIMS v5 and later:

1. You have defined the following custom **global fields** on the **Derived Sample** object:
 - **Concentration**
 - **Size (bp)**
 - **Conc. nM**
2. You have set the three fields configured in step 1 to display in the **Sample** table of the **Record Details** screen.
3. You have configured a **Calc. Prep** step to apply **Concentration, Size (bp)** to generated derived samples.
 - You have run the **Calc. Prep** step and it has generated derived samples.
 - You have input values for the **Calculation** and **Size (bp)** fields.
4. You have configured a **Calculation** step to apply **Conc. nM** to generated derived samples. You have run the **Calculation** step - with the derived samples generated by the **Calc. Prep** step as inputs, and it has generated derived samples.

Code example

First, the values to be used in the calculation - the **Concentration** and **Size (bp)** UDFs / custom fields are applied to the samples by running the **Calc. Prep** preparation step. You can then enter the values for these fields into the LIMS as follows:

LIMS v5 and later:

- In the **Record Details** screen, in the **Sample** table.

LIMS v4.2 and earlier:

- [CAL-SA1-130123-24-4146] Calc. Prep

LIMS ID:

CAL-SA1-130123-24-4146

Technician:

System Administrator

Date Run:

Jan 23, 2017

Instrument/Software:

Input/Output Explorer

Files

Details

Notes

Inputs

QC	Name	Container Identifier	Placement	
	Cookbook Test Sa...	27-1421		

0 rows selected

Outputs

QC	Name	Container Identifier	Placement	
	Cookbook Test Sa...	27-1422		

1 row selected

1D Gel

Run Process...

Selected Artifact Details

Sample Genealogy

Details

Details for

Output per Input: Analyte

☐ Show Non-Detail Fields

Detail Comparison

QC	Name	LIMS ID	Concentration	Size (bp)	
	Cookbook Test Sample	ADM206A1CA2	12.00	15	

Expected output and results

- **(LIMS v5 and later)** In the **Record Details** screen, in the **Sample** table.

- [CAL-SA1-130123-24-4148] Calculation**

LIMS ID: CAL-SA1-130123-24-4148 Technician: System Administrator
 Date Run: Jan 23, 2017 Instrument/Software:

Input/Output Explorer | Files | Details | Notes

Inputs			
QC	Name	Container Identifier	Placement
+ Cookbook Test Sa...		27-1422	

0 rows selected

Outputs			
QC	Name	Container Identifier	Placement
+ Cookbook Test Sa...		27-1423	

1 row selected

1D Gel Run Process...

Selected Artifact Details

Sample Genealogy | Details

Details for Output per Input: Analyte ☐ Show Non-Detail Fields

Detail Comparison			
QC	Name	LIMS ID	Conc. nM
	Cookbook Test Sample	ADM206A1CA3	1218

UsingAnalyteUDFForCalculations.groovy.txt

When working with multiplexing, you can do the following:

- ## Filter Processes by Date and Type

Workflows, chemistry, hardware, and software are continually changing in the lab. As a result, you can determine which samples were processed after a specific change happened.

Using the processes (list) resource you can construct a query that filters the list using both process type and date modified.

Prerequisites

Before you follow the example, make sure you have the following items:

- Samples that have been added to the system.
- Multiple processes of the Cookbook Example type that have been run on different dates.
- A compatible version of API (v2 r21 or later).

Code Example

In Clarity LIMS, when you search for a specific step type, the search results list shows all steps of that type that have been run, along with detailed information about each one. This information includes the protocol that includes the step, the number of samples in the step, the step LIMS ID, and the date the step was run.

The following screenshot shows the search results for the step type Denature and Anneal RNA (TruSight Tumor 170 v1.0).

The list shows the date run for each step, but not the last modified date. This is because a step can be modified after it was run, without changing the date on which it was run.

Your search for "denature and anneal*" returned 3 results.

Displaying 1-3 of 3

CATEGORY RESULTS

Samples

Containers

Projects

Protocol Steps 3

Denature and Anneal RNA (TruSight Tumor 17...

Protocol	RNA QC and cDNA Synth...	Date	Oct 20, 2017	LIMS ID	24-5020
Samples	8	LIMS User	System Administrator	Status	Completed

Denature and Anneal RNA (TruSight Tumor 17...

Protocol	RNA QC and cDNA Synth...	Date	May 10, 2017	LIMS ID	24-938
Samples	4	LIMS User	System Administrator	Status	Completed

Denature and Anneal RNA (TruSight Tumor 17...

Protocol	RNA QC and cDNA Synth...	Date	Apr 11, 2017	LIMS ID	24-835
Samples	3	LIMS User	System Administrator	Status	Completed

Displaying 1-3 of 3

To find the steps that meet the two criteria (step type and date modified), you must to do the following steps:

1. Request a list of all steps (processes), filtered on process type and date modified.
2. After you have the list of processes, you can use a script to print the LIMS ID for each process.

Step 1. List Processes of a Specific Type that were Modified After a Specified Date

To request a list of all processes of a specific type that were modified after a specified date, use the following GET method that uses both the `?type` and `?last-modified` filter on the processes resource:

```
// Retrieve a date and format it to a string c = Calendar.getInstance() c.add
(Calendar.WEEK_OF_YEAR, -1) df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ") time =
df.format(c.getTime()) time = URLEncoder.encode(time, "UTF-8") // Retrieve the processes
that were last-modified in the given date range processesURI =
"http://${hostname}/api/v2/processes?type=Denature%20and%20Anneal%20RNA%20%28TruSight%20
Tumor%20170%20v1.0%29andlast-modified=" + time processes = GLSRestApiUtils.httpGET
(processesURI, username, password)
```

The GET call returns a list of the first 500 processes that match the filter specified. If more than 500 processes match the filter, only the first 500 are available from the first page.

In the XML returned, each process is an element in the list. Each element contains the URI for the individual process resource, which includes the LIMS ID for the process.

The URI for the list of all processes is `http://yourIPaddress/api/processes`. In the example code, the list was filtered by appending the following:

```
?type=Denature%20and%20Anneal%20RNA%20%28TruSight%20Tumor%20170%20v1.0%29andlast-
modified= + time
```

This filters the list to show only processes that are of the Cookbook Example type and were modified after the specified date.

The date must be specified in ISO 8601, including the time. In the example, this is accomplished using an instance of a `Calendar` object and a `SimpleDateFormat` object, and encoding the date using UTF-8. The date specified is one week prior to the time the code is executed.

All of the REST list resources are paged. Only the first 500 items are returned when you query for a list of items, such as `http://youripaddress/api/v2/artifacts`.

If you cannot filter the list, you must iterate through the pages of a list resource to find the items that you are looking for. The URI for the next page of resources is always the last element on the page of a list resource.

After requesting an individual process XML resource, you have access to a large collection of data that lets you modify or view each process. Within the process XML, you can also access the artifacts that were inputs or outputs of the process.

After running the script on the command line, output is generated showing the LIMS ID for each process in the list.

Find Terminal Processes/Steps

Information about a step is stored in the process resource. In general, automation scripts access information about a step using the processURI, which links to the individual process resource. The input-output-map in the XML returned by the individual process resource gives the script access to the artifacts that were inputs and outputs to the process.

Processing a sample in the lab can be complex and is not always linear. This may be because more than one step (referred to as process in the API and in the Operations Interface in Clarity LIMS v4.x and earlier) is run on the same sample, or because a sample has to be modified or restarted because of quality problems.

The following illustration provides a conceptual representation of a Clarity LIMS workflow and its sample/process hierarchy. In this illustration, the terminal processes are circled.

Name	Date Run	LIMS ID
Liver-1		ANN753A1
RNA Extraction	Mar 25, 2010	RNX-MSA-100326-122-408
Liver-1		ANN753A1TP2
UV Quant	Mar 25, 2010	UVQ-MSA-100326-24-854
Spectrophotometer Results glsQCConvertedFile6519.csv		92-910
BioAnalyzer	Mar 25, 2010	BIA-MSA-100326-24-857
BA Combined Results glsQCConvertedFile6521.csv		92-912
RT (Arrays)	Mar 25, 2010	RTX-MSA-100326-122-409
Liver-1		ANN753A1RT3
In vitro transcription (Arrays)	Mar 25, 2010	IVT-MSA-100326-122-410
Liver-1		ANN753A1TP4
Array Hybridization (manual)	Mar 25, 2010	HYB-MSA-100326-133-52
Liver-1		ANN753A1HY5
Array Scan	Mar 25, 2010	ASC-MSA-100326-24-858
Array Scan Results		ANN753A1AS6
BeadStudio Gene Expression Analysis	Mar 25, 2010	BGX-MSA-100326-24-860
BeadStudio Analysis Result Liver stu...		92-921
API Cookbook Example 2.1	Aug 30, 2010	A21-BMJ-100830-24-1474
Liver-1		ANN753A1AP7
API Cookbook Example 1.3	Sep 3, 2010	A13-BMJ-100903-24-1494
Liver-1		ANN753A1AP8
API Cookbook Example 1.4	Sep 3, 2010	A14-BMJ-100903-24-1496
Liver-1		ANN753A1AP12
API Cookbook Example 2.3	Sep 3, 2010	A23-BMJ-100903-24-1495
Liver-1 Individual		ANN753A1AP9
A23-BMJ-100903-24-1495 Shared		ANN753A1AP10
Liver-1 Ex 2.3		ANN753A1AP11

This example finds all terminal artifact (sample)-process pairs. The main steps are as follows.

1. All the processes run on a sample are listed with a process (list) GET method using the ?inputartifactlimsid filter.
2. All the process outputs for an input sample are found with a process (single) GET.
3. Iteration through the input-output maps finds all outputs for the input of interest.

Prerequisites

Before you follow the example, make sure you have the following items:

- A sample to the system.
- Several steps that have been run, with several steps run on a single output at least one time.
- A compatible version of API (v2 r21 or later).

Code Example

To walk down the hierarchy from a particular sample, you must do the following steps:

1. List all the processes that used the sample as an input.
2. For each process on that list, find all the output artifacts that used that particular input. These output artifacts represent the next level down the hierarchy.
3. To find the artifacts for the next level down, repeat steps 1 and 2, starting with each output artifact from the previous round.
4. To find all artifacts in the hierarchy, repeat this process until there are no more output artifacts. The last processes found are the terminal processes.

This example starts from the original submitted sample.

Step 1. Retrieve the Sample Resource and Find its Analyte Artifact (Derived Sample) URI

The first step is to retrieve the sample resource via a GET call and find its analyte artifact (derived sample) URI. The analyte artifact of the sample is the input to the first process in the sample hierarchy.

The following GET method provides the full XML structure for the sample including the analyte artifact URI:

```
// Retrieve the sample sampleURI = "http://${hostname}/api/v2/samples/${sampleLIMSID}"
sample = GLSRestApiUtils.httpGET(sampleURI, username, password) // Initialize storage
variables targetAnalyteLIMSID = sample.artifact.@limsid[0] artifactMap = [:] artifactMap
[targetAnalyteLIMSID] = null lastProcMap = [:] processesURI =
"http://${hostname}/api/v2/processes?inputartifactlimsid="
```

- The `sample.artifact.@limsid` contains the original analyte LIMS ID of the sample. For each level of the hierarchy, the artifacts are stored in a Groovy Map called `artifactMap`. The `artifactMap` uses the process that generated the artifact as the value, and the artifact LIMS ID as the key. At the top sample level, the list is only comprised of the analyte of the original sample. In the map, the process is set to null for this sample analyte.

Step 2. Find All of the Processes Run on the Artifact

To find all the processes run on the artifacts, use a GET method on the process (list) resource with the `?inputartifactlimsid` filter.

In the last line of the example code, the `processURI` string sets up the first part of the URI. The artifact `LIMSID` is added (concatenated) for each GET call in the following while loop:

```
// While there are still output artifacts to process outputArtifactsExist = true while
(outputArtifactsExist) { outputArtifactMap = [:] artifactMap.each { key, value -> //
Retrieve a list of processes that have the given artifact as an input processes =
GLSRestApiUtils.httpGET(processesURI + key, username, password) processes.'process'.each
{ process = GLSRestApiUtils.httpGET(it.@uri, username, password) // For each input-
output-map, add the process limsid to the storage map process.'input-output-map'.each {
iomap -> if(key == iomap.input.@limsid[0]) { outputArtifactMap[iomap.output.@limsid[0]] =
process.@limsid } } } // If there were no processes if(!processes.'process') {
lastProcMap[key] = value } } // If there are no more artifacts to process, set variable
to exit if(outputArtifactMap.isEmpty()) { outputArtifactsExist = false } else {
artifactMap = outputArtifactMap } }
```

The while loop evaluates one level of the hierarchy for every iteration. Each artifact at that level is evaluated. If that artifact was not used as an input to a process, an artifact/process key value pair is stored in the `lastProcMap`. All the Groovy maps in the previous code use this artifact/process pair structure.

The loop continues until there are no artifacts that had outputs generated. For each artifact evaluated, the processes that used the artifact as an input are found and collected in the `processes` variable. Because a process can be run without producing outputs, a GET call is done for each of the processes to determine if the artifact generated any outputs.

Any outputs found will form the next level of the hierarchy. The outputs are temporarily collected in the `outputArtifactMap`. If no processes were found for that artifact, then it is an end leaf node of a hierarchy branch. Those artifact/process pairs are collected in the `lastProcMap`.

You can iterate through each pair of artifact and process LIMS IDs in `outputArtifactMap` and print the results to standard output.

```
// Print the artifact associated with the given process limsid lastProcMap.each { key,
value -> println value + ',' + key }
```

Expected Output and Results

Running the script in a console produces the following output:

```
UVQ-MSA-100326-24-854,92-910 BIA-MSA-100326-24-857,92-912 A23-BMJ-100903-24-
1495,ANN753A1AP11 A23-BMJ-100903-24-1495,ANN753A1AP10 A23-BMJ-100903-24-1495,ANN753A1AP9
A14-BMJ-100903-24-1496,ANN753A1AP12 BGX-MSA-100326-24-860,92-921
```

Run a Process/Step

When samples are processed in the lab, they are sometimes re-arrayed in complex ways that are pre-defined.

You can use the REST API and automation functionality that allows a user to initiate a step that does the following actions:

- Uses a file to define a re-array pattern.

- Executes the step using that re-array pattern. Because the pattern is pre-defined, this decreases the likelihood of an error in recording the re-array.

To accomplish this automation, you must be able to execute a step using the REST API. This example shows a simple step execution that you can apply to any automated step execution needed in your lab.

For a high-level overview of REST resource structure in Clarity LIMS, including how processes are the key to tracking work, refer to [REST General Concepts](#).

Prerequisites

Before you follow the example, make sure that you have the following items:

- Samples that have been added to the system.
- A configured step/process that generates analytes (derived samples) and a shared result file.
- Samples that have been run through the configured process/step.
- A compatible version of API (v2 r21 or later).

 Information about a step is stored in the process resource in the API.

Information about a derived sample is stored in the analyte resource in the API. This resource is used as the input and output of a step, and also used to record specific details from lab processing.

Code Example

To run a step/process on a set of samples, you must first identify the set of samples to be used as inputs. The samples that are inputs to a step/process can often be identified because they are all in the same container, or because they are all outputs of a previous step / process. For more information, refer to [Find the Contents of a Well Location in a Container](#) and [View the Inputs and Outputs of a Process/Step](#).

In this example, you run the step/process on the samples listed in the following table.

Sample Information

Submitted Sample Name	Derived SampleName	Derived Sample LIMS ID	Container LIMS ID	ContainerType	Well
Soleus-1	Soleus-1	AFF853A53AP11	27-4056	96 well plate	A:1
Soleus-2	Soleus-2	AFF853A54AP11	27-4056	96 well plate	A:2
Soleus-3	Soleus-3	AFF853A55AP11	27-4056	96 well plate	A:3

After you have identified the samples, use their LIMS IDs to construct the URIs for the respective analyte (derived sample) artifacts. The artifact URIs are used as the inputs in constructing the XML to POST and execute a process.

You can use `StreamingMarkupBuilder` to construct the XML needed for the POST, as shown in the following example code:

```
// Determine the list URIs and the specified analyte URIs processListURI =
"http://${hostname}/api/v2/processes" researcherURI =
"http://${hostname}/api/v2/researchers/1" analyte1URI =
"http://${hostname}/api/v2/artifacts/${analyteLIMSIDs[0]}" analyte2URI =
"http://${hostname}/api/v2/artifacts/${analyteLIMSIDs[1]}" analyte3URI =
"http://${hostname}/api/v2/artifacts/${analyteLIMSIDs[2]}" // Retrieve the process type
processTypeNode = GLSRestApiUtils.httpGET(processTypeURI, username, password) // Create a
new process using the Markup Builder def processDoc = new StreamingMarkupBuilder().bind {
mkp.xmlDeclaration() mkp.declareNamespace(prx: 'http://genologics.com/ri/processexecution')
'prx:process' { 'type' (processTypeNode.@name) 'technician' (uri:researcherURI) 'input-
output-map' { 'input' (uri:analyte1URI) 'output' (type:'Analyte') { 'location' { 'container'
(uri:container96WellsURI) 'value' ("A:1") } } } 'input-output-map' { 'input' (uri:analyte2URI)
'output' (type:'Analyte') { 'location' { 'container' (uri:container96WellsURI) 'value' ("A:2")
} } } 'input-output-map' { 'input' (uri:analyte3URI) 'output' (type:'Analyte') { 'location'
{ 'container' (uri:container96WellsURI) 'value' ("A:3") } } } 'input-output-map'
(shared:'true') { 'input' (uri:analyte1URI) 'input' (uri:analyte2URI) 'input' (uri:analyte3URI)
'output' (type:'ResultFile') } } } // Post the new process to the API unresolvedProcessNode =
GLSRestApiUtils.xmlStringToNode(processDoc.toString()) returnNode = GLSRestApiUtils.httpPOST
(unresolvedProcessNode, "${processListURI}", username, password)
```

Executing a process uses the `processexecution` (`prx`) namespace. The example is highlighted in red.

The required elements for a successful POST are as follows.

- **type**—The name of the process being run.
- **technician uri**—The URI for the technician that is listed as running the process.
- **input-output-map**—One input output map element for each pair of inputs and outputs.
- **input uri**—The URI for the input artifact.
- **output type**—The type of artifact of the output.

In addition, if the outputs of the process are analytes, then the following are also needed:

- **container uri**—The URI for the container the output is placed in.
- **value**—The well placement for the output.

The process type, technician, input artifact, and container must all exist in the system before the process can be executed. For example, if there is no container with an empty well, you must create a container before running the process.

The XML constructed must match the configuration of the process type. For example, if the process is configured to have both samples and a shared result file as outputs, you must have both of the following items:

- An input-output-map for each pair of sample inputs and outputs.

- An additional input-output-map for the shared result file.

If the POST is successful, the process XML is returned:

```
<prc:process xmlns:prc="http://genologics.com/ri/process"
uri="http://yourIPaddress/api/v2/processes/A22-BMJ-100930-24-2203" limsid="A22-BMJ-100930-
24-2203"> <type>HiSEQ PE</type> <date-run>2016-09-30</date-run> <technician
uri="http://yourIPaddress/api/v2/researchers/305"> <first-name>John-Luck</first-name> <last-
name>Pikkard</last-name> </technician> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/AFF853A55AP11?state=21128" post-process-
uri="http://yourIPaddress/api/v2/artifacts/AFF853A55AP11?state=21135"
limsid="AFF853A55AP11"> <parent-process uri="http://yourIPaddress/api/v2/processes/A33-BMJ-
100930-24-2200" limsid="A33-BMJ-100930-24-2200"/> </input> <output
uri="http://yourIPaddress/api/v2/artifacts/92-2538?state=21134" output-type="ResultFile"
limsid="92-2538"/> </input-output-map> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/AFF853A55AP11?state=21128" post-process-
uri="http://yourIPaddress/api/v2/artifacts/AFF853A55AP11?state=21135"
limsid="AFF853A55AP11"> <parent-process uri="http://yourIPaddress/api/v2/processes/A33-BMJ-
100930-24-2200" limsid="A33-BMJ-100930-24-2200"/> </input> <output
uri="http://yourIPaddress/api/v2/artifacts/AFF853A55AP13?state=21136" output-type="Analyte"
limsid="AFF853A55AP13"/> </input-output-map> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/AFF853A53AP11?state=21124" post-process-
uri="http://yourIPaddress/api/v2/artifacts/AFF853A53AP11?state=21130"
limsid="AFF853A53AP11"> <parent-process uri="http://yourIPaddress/api/v2/processes/A33-BMJ-
100930-24-2200" limsid="A33-BMJ-100930-24-2200"/> </input> <output
uri="http://yourIPaddress/api/v2/artifacts/92-2538?state=21134" output-type="ResultFile"
limsid="92-2538"/> </input-output-map> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/AFF853A53AP11?state=21124" post-process-
uri="http://yourIPaddress/api/v2/artifacts/AFF853A53AP11?state=21130"
limsid="AFF853A53AP11"> <parent-process uri="http://yourIPaddress/api/v2/processes/A33-BMJ-
100930-24-2200" limsid="A33-BMJ-100930-24-2200"/> </input> <output
uri="http://yourIPaddress/api/v2/artifacts/AFF853A53AP13?state=21132" output-type="Analyte"
limsid="AFF853A53AP13"/> </input-output-map> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/AFF853A54AP11?state=21125" post-process-
uri="http://yourIPaddress/api/v2/artifacts/AFF853A54AP11?state=21133"
limsid="AFF853A54AP11"> <parent-process uri="http://yourIPaddress/api/v2/processes/A33-BMJ-
100930-24-2200" limsid="A33-BMJ-100930-24-2200"/> </input> <output
uri="http://yourIPaddress/api/v2/artifacts/92-2538?state=21134" output-type="ResultFile"
limsid="92-2538"/> </input-output-map> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/AFF853A54AP11?state=21125" post-process-
uri="http://yourIPaddress/api/v2/artifacts/AFF853A54AP11?state=21133"
limsid="AFF853A54AP11"> <parent-process uri="http://yourIPaddress/api/v2/processes/A33-BMJ-
```

```
100930-24-2200" limsid="A33-BMJ-100930-24-2200"/> </input> <output
uri="http://yourIPaddress/api/v2/artifacts/AFF853A54AP13?state=21131" output-type="Analyte"
limsid="AFF853A54AP13"/> </input-output-map></prc:process>
```

If the POST is not successful, the XML returned contains the error that occurred when the POST completed:

```
<exc:exception xmlns:exc="http://genologics.com/ri/exception"> <message>The process type
named 'HiSEQ PE' cannot produce the following types of shared outputs:
'ResultFile'.</message></exc:exception>
```

Expected Output and Results

After the step/process has successfully executed, you can open the Record Details screen and see the step outputs. In Clarity LIMS v4.2 and earlier, you can also see these details in the Operations Interface, in the Input/Output Explorer of the process summary view.

[A22-BMJ-100930-24-2203] HiSEQ PE

LIMS ID: A22-BMJ-100930-24-2203

Technician: John-Luck Pikkard

Date Run: Sep 30, 2016

Instrument/Software:

Input/Output Explorer

Files

Details

Notes

Inputs

QC	Name	Placement	Sample LIMS ID
96	27-4056-1		
	Soleus-1	A:1	AFF853A53AP11
	Soleus-2	A:2	AFF853A54AP11
	Soleus-3	A:3	AFF853A55AP11

Outputs

QC	Name	Placement	Sample LIMS ID
96	HiSEQ POST		
	Soleus-1	A:2	AFF853A53AP13
	Soleus-2	A:3	AFF853A54AP13
	Soleus-3	A:1	AFF853A55AP13
	[92-2538] S...		

Attachments

[RunningAProcess.groovy.txt](#)

Update UDF/Custom Field Information for a Process/Step

Steps can have user-defined fields (UDFs)/custom fields that can be used to describe properties of the steps. For example, while a sample UDF/custom field might describe the gender or species of the sample, a process UDF/custom field might describe the room temperature recorded during the step or the reagent lot identifier. Sometimes, some of the information about a step is not be known until it has completed on the instrument, but after it was run in Clarity LIMS. In this example, we will record the Actual Equipment Start Time as a process UDF/custom field after the step has been run in Clarity LIMS. The ISO 8601 convention is used for recording the date and time. For more information, refer to [Filter Processes by Date and Type](#).

i | In the API, information about a step is stored in the process resource.

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

Prerequisites

Before you follow the example, make sure you have the following items:

- For Clarity LIMS v4.2 and earlier:
 - Samples added to the system.
 - A process type that has been defined with a UDF named Actual Equipment Start Time.
 - Samples that have been run through a process of this type.
- For Clarity LIMS v5 and later:
 - Samples added to the system.
 - A custom field named Actual Equipment Start Time that has been configured on a master step (master step field).
 - On the master step, you have configured the field to display on the Record Details milestone, in the Master Step Fields section.
 - You have run samples through a step based on the master step on which the Actual Equipment Start Time field is configured.
- A compatible version of API (v2 r21 or later).

Code Example

Detailed information for each step run in Clarity LIMS, including its name, LIMS ID, and custom fields can be viewed on the Record Details screen.

In the image below, an Actual Equipment Start Time master step field has been configured to display in the Step Details section of the Record Details screen. However, a value for this field has not yet been specified.

Cookbook Example / Cookbook Example

Record Details (6 samples)

System Administrator

Started May 11, 2017 17:42:54
Updated May 11, 2017 17:42:54

Step Details

Actual Equipment Start Time

Sample Details

Container Name

Apply

Sample Name	Container Name	Well	LIMS ID (Container)	Project Name
▶ Single Well: Tube (6 samples)				

In Clarity LIMS v4.2 and earlier, detailed information for each process/step run in Clarity LIMS (including its name, LIMS ID, and UDFs) can be viewed on the Details tab of the Operations Interface.

In the following image, the Actual Equipment Start Time UDF has been defined for the process type. However, a value for this field has not yet been specified.

[A22-BMJ-100927-24-2188] HiSEQ PE

LIMS ID: A22-BMJ-100927-24-2188 Technician: System Administrator

Date Run: May 11, 2017 Instrument/Software:

Input/Output Explorer Files **Details** Notes

Details

Actual Equipment Start Time:

i Detail Fields and Types can be added/removed in the Configuration Dialog.

This example shows how to update the field value to include a date and a time, both of which are parsed in from a log or report file.

Step 1. Request the Process Resource

Before you can change the value of a process UDF/custom field, you must first request the individual process resource via a GET HTTP call. The XML returned from a GET on the individual process resource contains the information about that process. The following GET method provides the complete XML structure for the process:

```
// Determine the specified process URI and retrieve it processURI =
"http://${hostname}/api/v2/processes/${processLIMSID}" processNode =
```

```
GLSRestApiUtils.httpGET(processURI, username, password) // Retrieve the process's 'Actual Equipment Start Time' UDF
startTimeUDF = processNode.'udf:field'.find { it.@name == 'Actual Equipment Start Time' }
newStartTime = parseInstrumentStartTime()
```

The variable `processNode` now holds the complete XML structure retrieved from the resource at `processURI`.

The variable `startTimeUDF` references the XML element node contained in the structure that relates to the Actual Equipment Start Time UDF/custom field (if one exists).

The variable `newStartTime` is a string initialized with a value from the method `parseInstrumentStartTimes`. The details of this method are omitted from this example, but its function is to parse the date and time the instrument started from a log file.

 | The XML representations of individual REST resources are self-contained entities. Always request the complete XML representation before editing any portion of the XML. If you do not use the complete XML when you update the resource, you can inadvertently change data.

The following code shows the XML structure for the process, as stored in the variable `processNode`. There are no child UDF/custom field nodes.

```
<prc:process uri="http://yourIPAddress/api/v2/processes/A22-BMJ-100927-24-2188"
limsid="A22-BMJ-100927-24-2188"> <type>HiSEQ PE</type> <date-run>2017-05-11</date-run>
<technician uri="http://yourIPAddress/api/v2/researchers/305"> <first-name>System</first-
name> <last-name>Administrator</last-name> </technician> <input-output-map> <input
uri="http://yourIPAddress/api/v2/artifacts/AFF853A49PA1?state=21004" post-process-
uri="http://yourIPAddress/api/v2/artifacts/AFF853A49PA1?state=21009"
limsid="AFF853A49PA1"/> <output
uri="http://yourIPAddress/api/v2/artifacts/AFF853A49AP3?state=21010" output-
type="Analyte" limsid="AFF853A49AP3"/> </input-output-map></prc:process>
```

Step 2. Update the Process Resource

There are two possible cases you can encounter when trying to update the UDF/custom field:

- **The field already exists**—In this case, change the value of the UDF/custom field node in the XML.
- **The field does not exist**—In this case, append a new UDF/custom field node to the process resource.

After modifying the process stored in the variable `processNode`, you can use a PUT method to update the process resource.

You can check if the UDF/custom field exists by verifying the value of the `startTimeUDF` variable. If the value is not null, then the field is defined and you can set a new value in the XML. If the field does not exist, you will must append a new node to the process XML resource using the UDF/custom field name and new value.

Before you can append a node to the XML, you must first specify the namespace for the new node. You can use the Groovy built-in `QName` class to do this. A `QName` object defines the qualified name of an XML element and

specifies its namespace. The node you are specifying is a UDF element, so the namespace is `http://genologics.com/ri/userdefined`. The local part is `field` and the prefix is `udf` for the QName, which specifies the element as a UDF/custom field.

To append a new node to the process using the `appendNode` method of the variable `processNode` (which appends a node with the specified QName, attributes, and value). Specify the following attributes for the UDF/custom field element:

- the type
- the name

Both of these elements must match a UDF/custom field that has been specified in the Configuration window for the process type.

```
// If the process's UDF was set, replace it, otherwise create a new node and attach it to
the process if (startTimeUDF) { startTimeUDF.setValue(newStartTime) } else {
startTimeQName = new QName('http://genologics.com/ri/userdefined', 'field', 'udf')
processNode.appendNode(startTimeQName, [type:'Numeric',name:'Actual Equipment Start
Time'], newStartTime) }
```

- The variable `processNode` now holds the complete XML structure for the process with the updated, or added, UDF named Actual Equipment Start Time.

You can save the changes you have made to the process using a PUT method on the process resource:

```
// Update the process in the API returnNode = GLSRestApiUtils.httpPUT(processNode,
processNode.uri, username, password) println GLSRestApiUtils.nodeToXmlString(returnNode)
```

The PUT updates the sample resource at the specified URI using the complete XML representation, including the new value for Actual Instrument Start Time.

If the PUT was successful, it returns the XML resource, as shown below. The updated information is also available at the `http://yourIPaddress/api/v2/processes/A22-BMJ-100927-24-2188` URI.

```
<prc:process uri="http://yourIPaddress/api/v2/processes/A22-BMJ-100927-24-2188"
limsid="A22-BMJ-100927-24-2188"> <type>HiSEQ PE</type> <date-run>2017-05-11</date-run>
<technician uri="http://yourIPaddress/api/v2/researchers/305"> <first-name>System</first-
name> <last-name>Administrator</last-name> </technician> <input-output-map> <input
uri="http://yourIPaddress/api/v2/artifacts/AFF853A49PA1?state=21004" post-process-
uri="http://yourIPaddress/api/v2/artifacts/AFF853A49PA1?state=21009"
limsid="AFF853A49PA1"/> <output
uri="http://yourIPaddress/api/v2/artifacts/AFF853A49AP3?state=21010" output-
type="Analyte" limsid="AFF853A49AP3"/> </input-output-map> <udf:field type="String"
name="Actual Equipment Start Time">2017-05-13T3:14:15-0700</udf:field></prc:process>
```

- If the PUT was unsuccessful, an XML resource is returned with contents that detail why the call was unsuccessful. In the following example error, an incorrect UDF/custom field name was specified. A UDF/custom field named Equipment Start Time was created in the process resource, but no UDF/custom field with that name was configured for the process type/master step.

```
<exc:exception xmlns:exc="http://genologics.com/ri/exception"> <message>Field Equipment  
Start Time does not exist</message></exc:exception>
```

Expected Output and Results

The Step Details section of the updated Record Details screen now shows the Actual Equipment Start Time value.

Cookbook Example / Cookbook Example

Record Details Next Steps

Abort Save NEXT STEPS »

Cookbook Example
Record Details (6 samples)

System Administrator Started May 11, 2017 17:42:54
Updated May 11, 2017 17:42:54

Step Details

Actual Equipment Start Time
2017-05-13T3:14:15-0700

Sample Details

Container Name Apply

Sample Name	Container Name	Well	LIMS ID (Container)	Project Name
▶ Single Well: Tube (6 samples)				

For Clarity LIMS v4.2 and earlier, the updated process view in the Clarity LIMS Operations Interface now shows the Actual Equipment Start Time value.

[A22-BMJ-100927-24-2188] HiSEQ PE
May 11, 2017

LIMS ID: A22-BMJ-100927-24-2188 Technician: System Administrator
Date Run: May 11, 2017 Instrument/Software:

Input/Output Explorer Files **Details** Notes

Details

Actual Equipment Start Time: 2017-05-13T3:14:15-0700

i Detail Fields and Types can be added/removed in the Configuration Dialog.

This example demonstrates a process UDF/master step custom field update through the process resource PUT method.

The ability to modify process properties allows you to update automatically and store lab activity information as it becomes available. Information from equipment log files or other data sources can be collected in this way.

Updating per-run or per-process/step information is powerful because the information can be used to optimize lab work (eg, by tracking trends over time). The data can be compared by instrument, length of run, lab conditions, and even against quality of molecular results.

Attachments

[UpdateProcessUDFInfo.groovy.txt](#)

Work with the Steps Pooling Endpoint

Pooling steps require that each input analyte artifact (derived sample) in the step be inserted into a pool. You can automate this task by using the API steps pooling endpoint. Automation of pooling allows you to reduce error and user interaction with Clarity LIMS.

In this example, a script pools samples based on the value of the pool id user-defined field (UDF)/custom field of the artifact.

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

To keep this example simple, the script does not handle samples with reagent labels.

i | In the API, an artifact is an item generated by an earlier step. There are two types of artifacts: analyte (derived sample) and resultfile (measurement). In the Clarity LIMS web interface, the terms artifact, analyte, and resultfile have been replaced with derived sample or measurement. However, in Clarity LIMS v4.2 and earlier, some areas of the Operations Interface still refer to the former terms.

Prerequisites

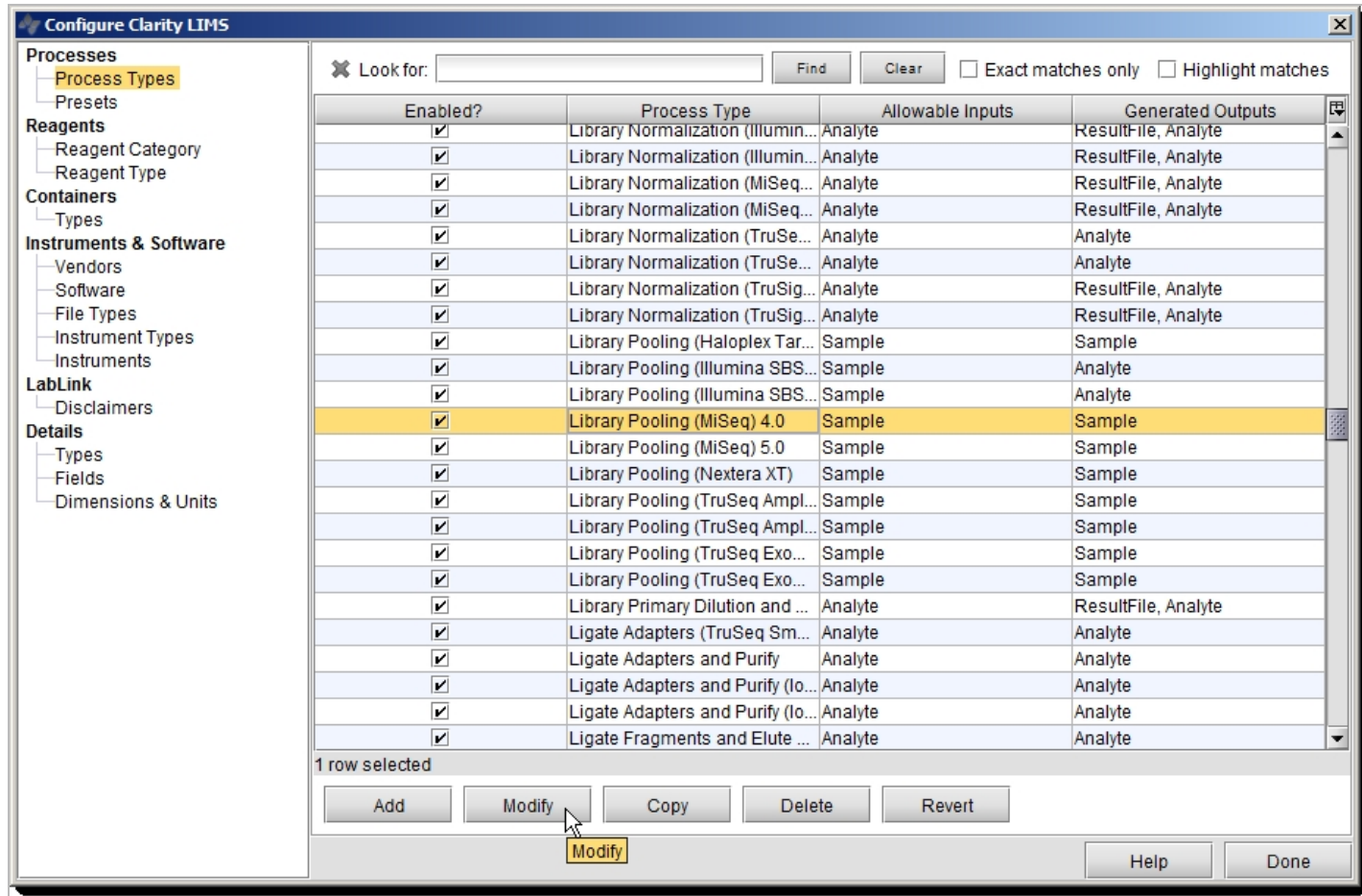
Before you follow the example, make sure that you have the following items:

- A configured analyte UDF/derived-sample custom field named pool id in Clarity LIMS.
- Groovy that is installed on the server and accessible at `/opt/groovy/bin/groovy`.
- The `GLSRestApiUtils.groovy` file that is stored in `/opt/groovy/lib/`.
- The `WorkingWithStepsPoolingEndpoint.groovy` script that is stored in `/opt/gls/clarity/customextensions/`.
- A compatible version of API (v2 r21 or later).

Example (Clarity LIMS v4.2 and Earlier)

Step 1. Modify Pooling Step

1. In the Clarity LIMS Operations Interface, in the Configuration area, select the pooling step to automate. Select **Modify**.



2. On the Attributes tab, select the **Allow pooling of artifacts with the same reagent** checkbox.

This example does not handle input artifacts with reagent labels.

The screenshot shows the 'Modify Process Type' dialog box with the 'Attributes' tab selected. The 'Name' field is 'Library Pooling (MiSeq) 4.0' and the 'Abbreviation' is 'PSL'. There are checkboxes for 'Open the process tab after the process wizard completes', 'Display in Explorer Tree', and 'Add to Button Bar'. The 'Attributes' tab contains several options: 'Allow pooling of artifacts with the same reagent' (checked and highlighted with a red box), 'Do not allow process completion before last wizard panel' (checked), and 'Skip process inputs panel when there are already inputs' (checked). Below these are dropdown menus for 'When the process tab opens:' (set to 'Display and maximize the Input/Output Explorer mini-tabbed page'), 'Output container type:' (set to '96 well plate'), 'Default placement pattern:' (set to 'Place samples vertically'), and 'Group name:' (set to '3. MiSeq').

Modify Process Type

Name: Library Pooling (MiSeq) 4.0

Abbreviation: PSL ☐ Open the process tab after the process wizard completes

Icon: ☐ Display in Explorer Tree ☐ Add to Button Bar

Input Types Output Types Output Details Output Generation **Attributes** External Programs

☒ Allow pooling of artifacts with the same reagent

☒ Do not allow process completion before last wizard panel

☒ Skip process inputs panel when there are already inputs

When the process tab opens: Display and maximize the Input/Output Explorer mini-tabbed page

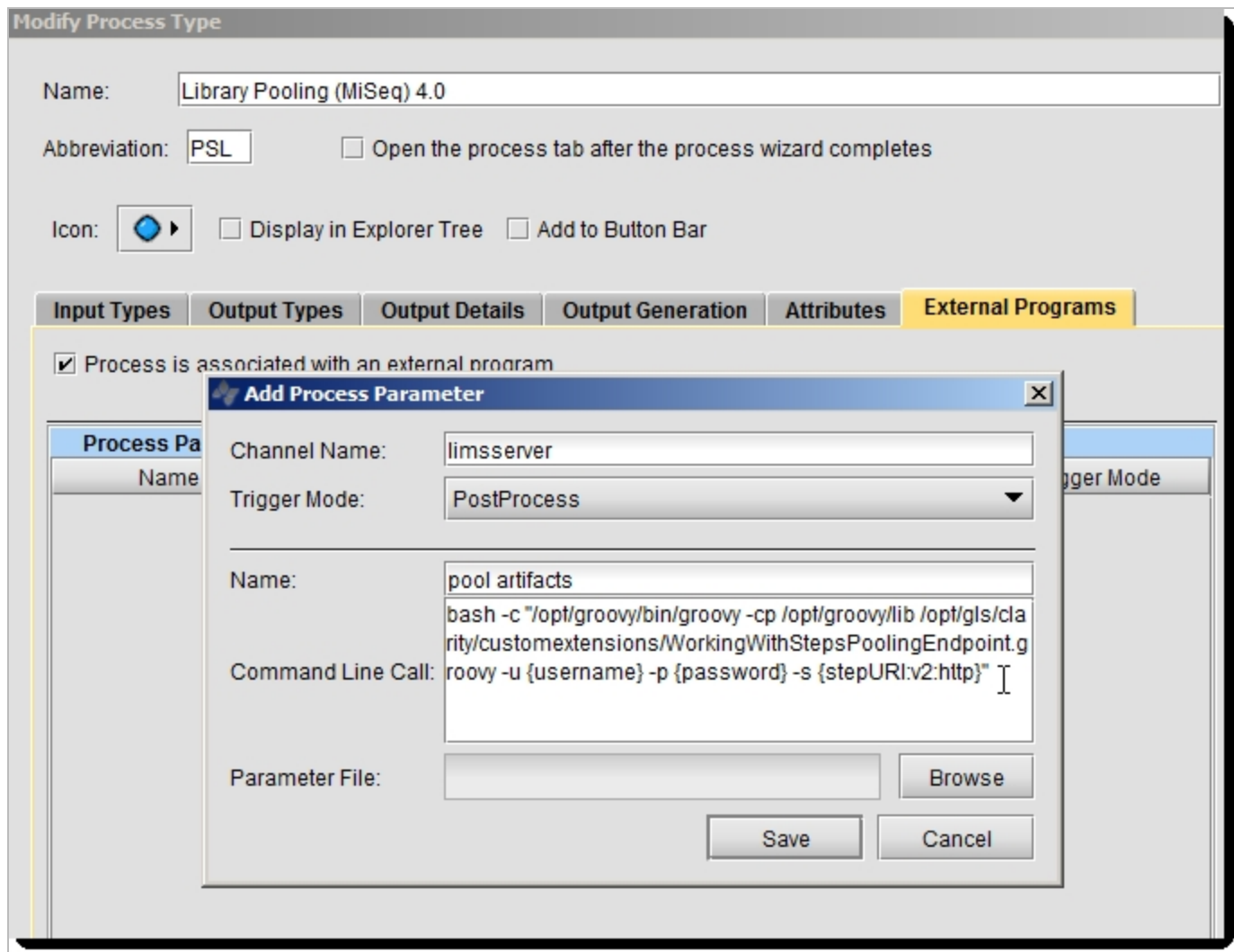
Output container type: 96 well plate

Default placement pattern: Place samples vertically

Group name: 3. MiSeq

3. On the External Programs tab, associate the process with the `WorkingWithStepsPoolingEndpoint.groovy` script. The command line call used in this example is as follows.

```
bash -c "/opt/groovy/bin/groovy -cp /opt/groovy/lib  
/opt/gls/clarity/customextensions/WorkingWithStepsPoolingEndpoint.groovy -u {username} -p  
{password} -s {stepURI:v2:http}"
```

Step 2. Create Protocol, Display pool id UDF, and Configure Automation Trigger

1. In the Clarity LIMS web interface, create a protocol and add the following two steps:
 - The add pool id step that sets the pool id UDF of the artifacts.
 - The pooling step modified in [Step 1. Modify Pooling Step](#).

LIMS Configuration Delete Save Protocol

Protocols
Workflows

Pooling by UDF

Sequencing Capacity ☐ Hidden

Select a Step to Add to your Protocol Add Step +

1 add pool id

2 Library Pooling (MiSeq) 4.0

- Modify the add pool id step and drag the pool id UDF into the Selected Fields area, making it editable in Record Details view.

Select fields to display within Sample Details table

Enter a custom heading for the Sample Details section

☐ Expand container groups

Available Sample Detail Fields

- ☐ testSetPooledContainerUDF_138966185...
- ☐ testSetUDF_1389661858577.udf.container
- ☐ testSetUDFs_1389661855232.udf1
- ☐ testSetUDFs_1389661855232.udf2
- ☒ Derived Sample
 - ☐ % Aligned R1
 - ☐ % Aligned R2
 - ☐ % Bases >=Q30 R1
 - ☐ % Bases >=Q30 R2
 - ☐ % Error Rate R1
 - ☐ % Error Rate R2

Selected Fields

Name	Type	Editable?
Selected Fields		
☐ Sample Name	Derived Sample	No
☐ Container Name	Container	No
☐ Well	Container	No
☐ LIMS ID (Container)	Container	No
☒ pool id	Derived Sample	Yes
☐ Project Name	Project	No

3. Modify the pooling step once more. This time, on the Automation tab, configure the automation so that so that the script is automatically initiated at the beginning of the step.

Protocols

Workflows

Automation

Pooling by UDF » Library Pooling (MiSeq) 4.0

« Back Properties **Automation** Next Steps Queue View Step Setup Record Details Next »

Configure trigger timing of automation actions

Name	Behaviour	Stage of Step	Timing
pool artifacts	Automatically initiated	at beginning of step	

Step 3. Create Workflow and Add Samples

1. Create a workflow containing the protocol created in [Step 2. Create Protocol, Display pool id UDF, and Configure Automation Trigger](#).
2. Create a project and add samples to it. Assign those samples to the workflow.

96 Submitted Samples 🔍 14 Selected Samples Delete

#	Name	Well	Date Sub...	Workflow
96 well plate: Patient Plate - 7623 96 Samples				
1	Patient 739.9 ⓘ	A:1	Jan 17, 2014	✕ Pooling By UDF
2	Patient 403.7 ⓘ	A:2	Jan 17, 2014	✕ Pooling By UDF
3	Patient 900.5 ⓘ	A:3	Jan 17, 2014	✕ Pooling By UDF
4	Patient 245.7 ⓘ	A:4	Jan 17, 2014	✕ Pooling By UDF
5	Patient 326.8 ⓘ	A:5	Jan 17, 2014	✕ Pooling By UDF
6	Patient 534 ⓘ	A:6	Jan 17, 2014	✕ Pooling By UDF
7	Patient 129.4 ⓘ	A:7	Jan 17, 2014	✕ Pooling By UDF
8	Patient 613 ⓘ	A:8	Jan 17, 2014	✕ Pooling By UDF
9	Patient 119.7 ⓘ	A:9	Jan 17, 2014	✕ Pooling By UDF

WORKFLOWS

DRAG SAMPLES HERE

Pooling By UDF

Empty ASSIGN

14 Samples in Pooling By UDF

Step 4. Run Samples Through the Steps

1. Begin working on the samples. In the first step, enter values into the pool id UDF.

 / add pool id

[Record Details](#) [Assign Next Steps](#) [Abort](#) [Save](#) [Assign Next Steps »](#)

Pooling by UDF
Record Details (14 samples)

 **System Administrator** Started
Updated Jan 17, 2014 15:09:39
Jan 17, 2014 15:09:39

Step Details 

No Step Details to Record

Sample Details pool id pool 1 [Apply](#)   

Sample Name	Container Name	Well	LIMS ID (Container)	pool id	Project Name
▼ Single Well: Tube (14 samples)					
Patient 2.6	27-636		27-636	pool 3	Pooling by ...
Patient 109.8	27-626		27-626	pool 3	Pooling by ...
Patient 119.7	27-633		27-633	pool 3	Pooling by ...
Patient 129.4	27-631		27-631	pool 3	Pooling by ...
Patient 245.7	27-628		27-628	pool 3	Pooling by ...
Patient 312.6	27-624		27-624	pool 2	Pooling by ...
Patient 326.8	27-629		27-629	pool 2	Pooling by ...
Patient 403.7	27-625		27-625	pool 2	Pooling by ...
Patient 534	27-630		27-630	pool 2	Pooling by ...
Patient 613	27-632		27-632	pool 2	Pooling by ...
Patient 642.8	27-634		27-634	pool 2	Pooling by ...
Patient 739.9	27-623		27-623	pool 1	Pooling by ...

2. Continue to the pooling step and add samples to the Ice Bucket. Select **Begin Work** to execute the script.

Pooling by UDF / Library Pooling (MiSeq) 4.0 / Ice Bucket Begin Work »

Ice Bucket (14 samples)

#	Sample Name	Well	Container Name	Waiting	
Single Well: Tube 14 Samples					
1	Patient 2.6		27-636	0 minutes	Remove
2	Patient 109.8		27-626	0 minutes	Remove
3	Patient 119.7		27-633	0 minutes	Remove
4	Patient 129.4		27-631	0 minutes	Remove
5	Patient 245.7		27-628	0 minutes	Remove
6	Patient 312.6		27-624	0 minutes	Remove
7	Patient 326.8		27-629	0 minutes	Remove
8	Patient 403.7		27-625	0 minutes	Remove
9	Patient 534		27-630	0 minutes	Remove
10	Patient 613		27-632	0 minutes	Remove
11	Patient 619.8		27-634	0 minutes	Remove

Empty Ice Bucket

CONTAINER OPTIONS

Destination Container
96 well plate

Find Existing Container
Name or LIMSID

Library Pooling (MiSeq) 4.0 Pool Samples Place Samples Record Details Assign Next Steps Abort Save Place Samples »

Samples Waiting for Pooling

Sample List

#	Sample Name	Well	Container Na...
Single Well: Tube			
1	Patient 2.6		
2	Patient 109.8		
3	Patient 119.7		
4	Patient 129.4		
5	Patient 245.7		27-628
6	Patient 312.6		27-624
7	Patient 326.8		27-629
8	Patient 403.7		27-625
9	Patient 534		27-630
10	Patient 613		27-632

Pooled Samples

Pool Creator Add Pool

Pool #1

Executing Custom Program

Please wait while executing 'pool artifacts'...

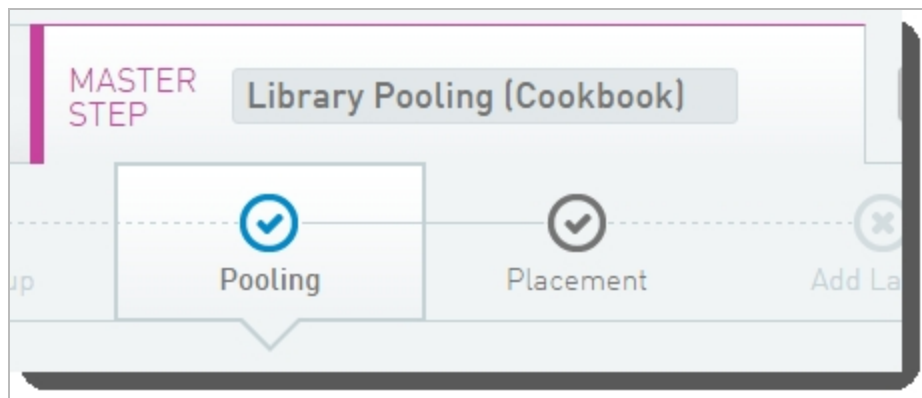
Cancel

Drag your samples from Samples Waiting for Pooling into the first Pool #1. Add new pools by clicking Add Pool button above.

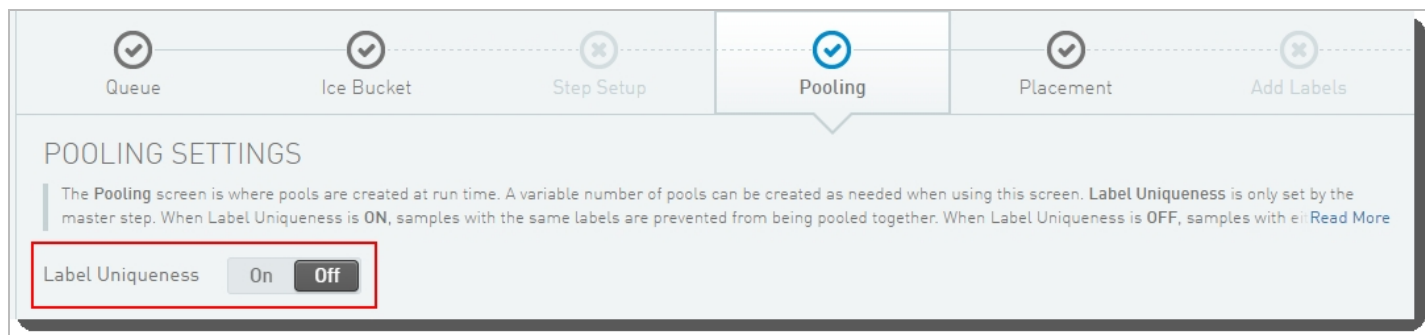
Example (Clarity LIMS v5 and Later)

Step 1. Add Master Step, Protocol, and Library Pooling Step

1. In Clarity LIMS, under Configuration, select the **Lab Work** tab.,
2. Select an existing Pooling master step or add a new one.

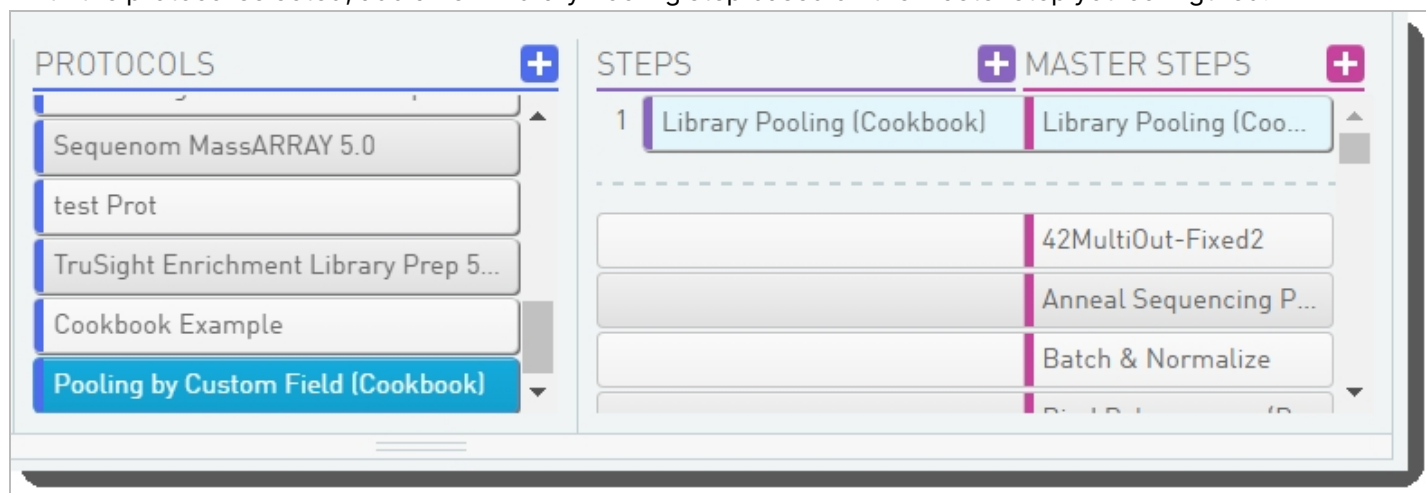


3. On the master step configuration form, select the **Pooling** milestone.
4. On the Pooling Settings form, set the Label Uniqueness toggle switch to **Off**.
5. Select **Save**.



6. Add a new protocol.

7. With the protocol selected, add a new Library Pooling step based on the master step you configured.



Step 2. Configure Step Automation and Trigger

1. In Clarity LIMS, under Configuration, select the **Automation** tab.
2. Add a new step automation. Associate the automation with the `WorkingWithStepsPoolingEndpoint.groovy` script. The command line used in this example is as follows.

```
bash -c "/opt/groovy/bin/groovy -cp /opt/groovy/lib  
/opt/gls/clarity/customextensions/WorkingWithStepsPoolingEndpoint.groovy -u {username} -p  
{password} -s {stepURI:v2:http}"
```

AUTOMATION DETAILS

NEW AUTOMATION +
Delete
Cancel
Save

Pool Samples

Channel Name limsserver

Command Line

```
bash -c "/opt/groovy/bin/groovy -cp /opt/groovy/lib
/opt/gls/clarity/customextensions/WorkingWithStepsPoolingEndpoint.groovy -u
{username} -p {password} -s {stepURL:v2:http}"
```

3. Enable the automation on the configured pooling master step. Select **Save**.
You can now configure the automation trigger on the step or the master step. If you configure the trigger on the master step, the settings will be locked on all steps derived from the master step.

AUTOMATION USE

Enable on these Master Steps

✕ Library Pooling (Cookbook)

4. On the Lab Work tab, select the library pooling step or master step.
5. On the Step Settings or Master Step Settings form, in the Automation section, configure the automation trigger so that the script is automatically initiated at the beginning of the step:
 - **Trigger Location**—Step
 - **Trigger Style**—Automatic upon entry

AUTOMATION

You must configure step automations at the master step level. However, you can configure the automation triggers on either the master step or on the steps derived from that master step. Triggers contain two elements: **Trigger Location** and **Trigger Style**. [Read More](#)

Automation Name	Trigger Location	Trigger Style
Pool Samples	Step >	Automatic upon entry >

Step 3. Configure Protocol and Add Pool ID Step

1. In Clarity LIMS, under Configuration, select the **Lab Work** tab.
2. Select the pooling protocol containing the Library Pooling step.
3. Add the Add Pool ID step that sets the pool id custom field of the samples. Move this step to the top of the Steps list.

PROTOCOLS

- Sequenom MassARRAY 5.0
- test Prot
- TruSight Enrichment Library Pr...
- Cookbook Example
- Pooling by Custom Field (Cookb...**

STEPS

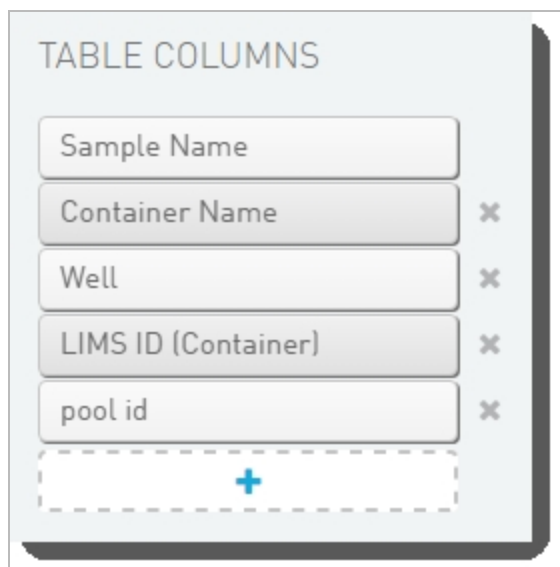
- Add Pool ID**
- Library Pooling (Cookbo...

- 42MultiOut-Fixed2
- Anneal Sequencing...

MASTER STEPS

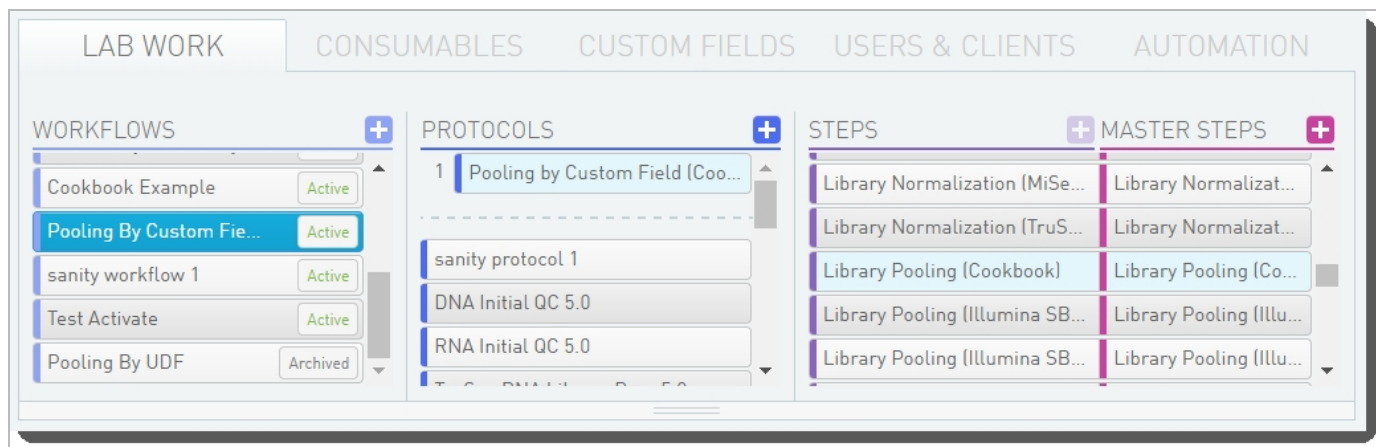
- Add Pool ID
- Library Pooling (C...

4. Select the **Add Pool ID** step.
5. On the Record Details milestone, add the pool id custom field to the Sample Details table.



Step 4. Create Workflow and Add Samples

1. In Clarity LIMS, under Configuration, select the **Lab Work** tab.
2. Create a workflow containing the configured pooling protocol. Activate the workflow.



3. On the Projects and Samples screen, create a project and add samples to it. Assign the samples to your pooling workflow.

SAMPLES & WORKFLOW ASSIGNMENT

SUBMITTED SAMPLES

DERIVED SAMPLES

Select A Filter >

Date Submitted

Delete

MODIFY SAMPLES

#	Name	Container	Well	Date Submitted	Workflow
96 Today					
1	Lung-100TJ	RC3-448	F:11	May 15, 2017	✕ Pooling By Custom Fi...
2	Lung-103WP	RC3-448	D:10	May 15, 2017	✕ Pooling By Custom Fi...
3	Lung-107SK	RC3-448	A:3	May 15, 2017	✕ Pooling By Custom Fi...
4	Lung-122ZM	RC3-448	E:5	May 15, 2017	✕ Pooling By Custom Fi...
5	Lung-150XX	RC3-448	B:3	May 15, 2017	✕ Pooling By Custom Fi...
6	Lung-153CE	RC3-448	C:12	May 15, 2017	✕ Pooling By Custom Fi...
7	Lung-158NV	RC3-448	G:3	May 15, 2017	✕ Pooling By Custom Fi...
8	Lung-164HZ	RC3-448	E:4	May 15, 2017	✕ Pooling By Custom Fi...

Select Group

WORKFLOWS

Assign To Workflow >

0 Samples in Cookbook Example

8 Samples in Pooling By Custom Field

0 Samples in sanity workflow 1

Step 5. Run Samples Through the Steps

1. Begin working on the samples. In the first step, enter values into the pool id custom field.

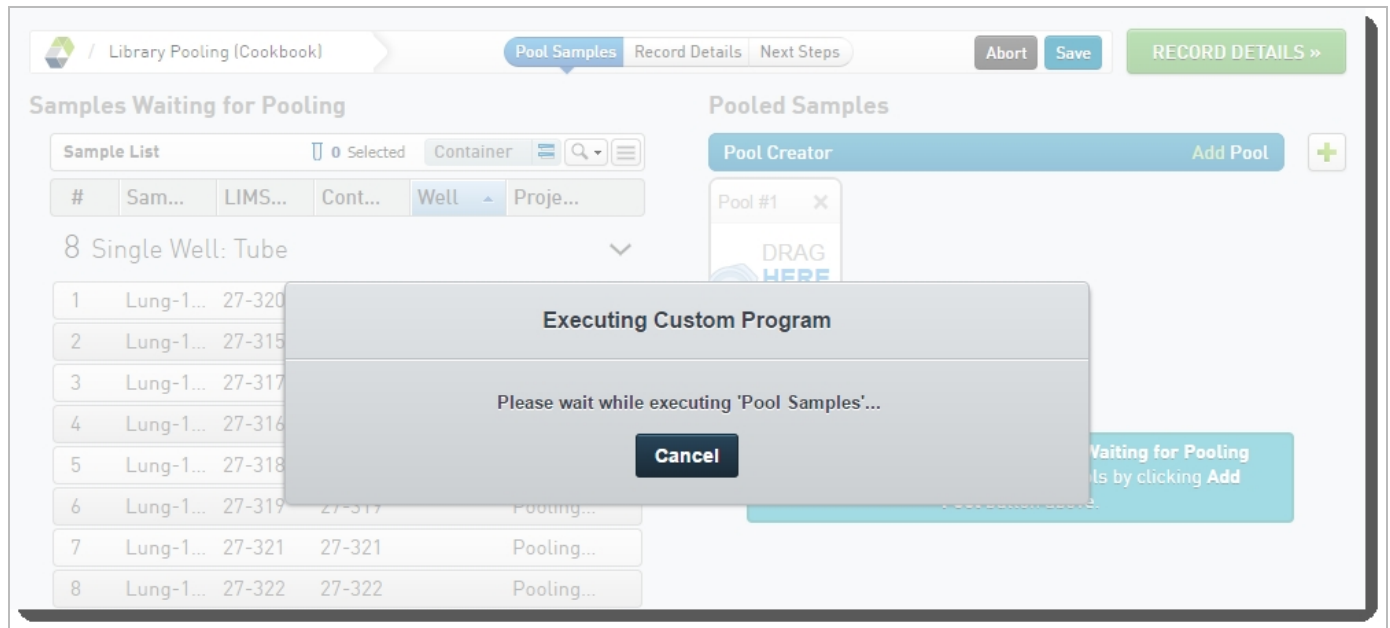
Sample Details

Container Name

Apply

Sample Name	Container Name	Well	LIMS ID (Container)	Project Name	pool id
▼ Single Well: Tube (8 samples)					
Lung-100TJ	27-320		27-320	Pooling Co...	pool 3
Lung-103WP	27-315		27-315	Pooling Co...	pool 3
Lung-107SK	27-317		27-317	Pooling Co...	pool 3
Lung-122ZM	27-316		27-316	Pooling Co...	pool 3
Lung-150XX	27-318		27-318	Pooling Co...	pool 2
Lung-153CE	27-319		27-319	Pooling Co...	pool 2
Lung-158NV	27-321		27-321	Pooling Co...	pool 1
Lung-164HZ	27-322		27-322	Pooling Co...	pool 1

2. Continue to the Library Pooling step and add samples to the Ice Bucket. Select **Begin Work** to execute the script.



How the Script Works

The script is passed the URI of the pooling step. Then, using the URI, the pool node of the step is retrieved. This node contains an available-inputs node that lists the URIs of the available input artifacts.

```
// Get the pool for this step poolNode = GLSRestApiUtils.httpGET(stepURI + '/pools',
username, password) poolNode.'available-inputs'.'input'.each{ artifact_uris.add(it.@uri) }
```

The script retrieves all available input artifacts, and then iterates through the list of retrieved artifacts. For each artifact, the script looks for the pool id custom field. If the field is not found, the script moves on to the next artifact. If the field is found, its value is stored in the `poolID` variable.

When the script encounters a new pool ID, it creates a new pool with a name equal to that ID. Input artifacts are sorted into pools based on the value of its pool id field, and as they are inserted into pools they are removed from the list of available inputs.

```
// Get all input artifacts artifacts = GLSRestApiUtils.batchGET(artifact_uris, username,
password) // Iterate through retrieved artifacts artifacts.each { // If the artifact has a
UDF pool ID, then pool the artifact and remove it from the available-inputs node poolID =
it.find { artifact_field -> artifact_field.@name == POOL_ID_UDF }?.value()[0] if (poolID !=
null) { // If a new pool ID is encountered, then use that new pool ID as the name for a new
pool node if (!poolNode.'pooled-inputs'.'pool'.find { pool -> pool.@name == poolID }) {
newPool = NodeBuilder.newInstance().pool(name:poolID) poolNode.'pooled-inputs'[0].append
(newPool) } inputNode = NodeBuilder.newInstance().input(uri:it.@uri) // Add the input
artifact to the pool with the name equal to 'poolID' poolNode.'pooled-inputs'.'pool'.find {
poolID == it.@name }.append(inputNode) // Remove the input artifact from the 'available-
inputs' node contained in the pool node poolNode.'available-inputs'[0].remove(inputNode) } }
```

After all of the available inputs are iterated through, the updated pool node is sent back to Clarity LIMS:

```
//Update the pool for this step and examine the result data resultNode =  
GLSRestApiUtils.httpPUT(poolNode, poolNode.@uri, username, password)
```

Expected Output and Results

Artifacts with the same Pool ID UDF/custom field will be automatically added to the same pool.

The screenshot displays the Clarity LIMS Pooling interface. On the left, the 'Samples Waiting for Pooling' section features a 'Sample List' table with columns for '#', 'Sample Name', 'Well', and 'Container Na...'. Below the table, it states 'All samples placed'. On the right, the 'Pooled Samples' section includes a 'Pool Creator' header with an 'Add Pool' button and a green plus icon. Below this, there are four pool slots: 'pool 1' with a count of 3, 'pool 2' with a count of 6, 'pool 3' with a count of 5, and 'Pool #4' with a 'DRAG HERE' instruction. Each pool slot has a blue circular icon with a white 'X' and a blue arrow pointing to it.

Attachments

- [WorkingWithStepsPoolingEndpoint.groovy.txt](#)
- [GLSRestApiUtils.groovy.txt](#)

Work with Batch Resources

For a general overview of batch resources, refer to [Introduction to Batch Resources](#).

When working with batch resources, you can do the following:

- [Update UDF/Custom Field Information with Batch Operations](#)
- [Retrieve Multiple Entities with a Single API Interaction](#)
- [Select the Optimal Batch Size](#)

Introduction to Batch Resources

The powerful batch resources included in the Clarity LIMS Rapid Scripting API significantly increase the speed of script execution by allowing batch operations on samples and containers. These resources are useful when working with multiple samples and containers in high throughput labs.

The following simple example uses batch resources to move samples from one workflow queue into another queue. It is useful to review the [Batch Resources Overview](#).

Use a `batch retrieve` request to find all the artifacts in an artifact group, and then use a `batch update` request to move those artifacts into another artifact group.

The following steps are required:

1. Find all the artifacts that are in a particular artifact group.
2. Use the `artifacts.batch.retrieve (list)` resource to retrieve the details for all the artifacts.
3. Use the `artifacts.batch.update (list)` resource to update the artifacts and move them into a different artifact group, posting them back as a batch.

i | The only HTTP method for batch resources is POST.

Prerequisites

Before you follow the steps, make sure that you have the following items:

- Clarity LIMS that contains a collection of samples (artifacts) residing in the same workflow queue (artifact group).
- A second queue that exists into which you can move the collection of samples.
- A compatible version of API (v2 r21 and later).

i | In the REST API, artifacts are grouped with the `artifact group` resource. In Clarity LIMS, an artifact group is displayed as a workflow. Workflows are configured as queues, allowing lab scientists to locate samples to work with on the bench quickly.

Step 1. Find All of the Samples in a Workflow Queue

To find the samples (artifacts) in a workflow queue (artifact group), use the following request, editing the server details and artifact group name to match those in your system:

```
http://your-server-ip/api/v2/artifacts?artifactgroup=my_queue
```

This request returns a list of URI links for all artifacts in the artifact group specified. In our example, the `my_queue` queue contains three artifacts:

```
<art:artifacts xmlns:art="http://genologics.com/ri/artifact"> <artifact limsid="ADM1A1PA1"
uri="http://your-server-ip/api/v2/artifacts/ADM1A1PA1"/> <artifact limsid="ADM1A2PA1"
uri="http://your-server-ip/api/v2/artifacts/ADM1A2PA1"/> <artifact limsid="ADM1A3PA1"
uri="http://your-server-ip/api/v2/artifacts/ADM1A3PA1"/></art:artifacts>
```

Step 2. Retrieve Sample Details

To retrieve the detailed XML for the artifacts, use a `<links>` tag to post the set of URI links to the server using a batch retrieve request:

```
<ri:links xmlns:ri="http://genologics.com/ri"> <link uri="http://your-server-
ip/api/v2/artifacts/ADM1A1PA1" rel="artifacts"/> <link uri="http://your-server-
ip/api/v2/artifacts/ADM1A2PA1" rel="artifacts"/> <link uri="http://your-server-
ip/api/v2/artifacts/ADM1A3PA1" rel="artifacts"/></ri:links>
```

`http://your-server-ip/api/v2/artifacts/batch/retrieve`

This returns the detailed XML for each of the artifacts in the batch:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><art:details
xmlns:udf="http://genologics.com/ri/userdefined" xmlns:file="http://genologics.com/ri/file"
xmlns:art="http://genologics.com/ri/artifact"> <art:artifact uri="http://your-server-
ip/api/v2/artifacts/ADM1A1PA1?state=21" limsid="ADM1A1PA1"> <name>base-1</name>
<type>Analyte</type> <output-type>Analyte</output-type> <qc-flag>UNKNOWN</qc-flag>
<location> <container uri="http://your-server-ip/api/v2/containers/27-10" limsid="27-10"/>
<value>1:1</value> </location> <working-flag>true</working-flag> <sample uri="http://your-
server-ip/api/v2/samples/ADM1A1" limsid="ADM1A1"/> <artifact-group name="my_queue"
uri="http://your-server-ip/api/v2/artifactgroups/1"/> </art:artifact> <art:artifact
uri="http://your-server-ip/api/v2/artifacts/ADM1A2PA1?state=42" limsid="ADM1A2PA1">
<name>base-2</name> <type>Analyte</type> <output-type>Analyte</output-type> <qc-
flag>UNKNOWN</qc-flag> <location> <container uri="http://your-server-
ip/api/v2/containers/27-11" limsid="27-11"/> <value>1:1</value> </location> <working-
flag>true</working-flag> <sample uri="http://your-server-ip/api/v2/samples/ADM1A2"
limsid="ADM1A2"/> <artifact-group name="my_queue" uri="http://your-server-
ip/api/v2/artifactgroups/1"/> </art:artifact> <art:artifact uri="http://your-server-
ip/api/v2/artifacts/ADM1A3PA1?state=12" limsid="ADM1A3PA1"> <name>base-3</name>
<type>Analyte</type> <output-type>Analyte</output-type> <qc-flag>UNKNOWN</qc-flag>
<location> <container uri="http://your-server-ip/api/v2/containers/27-12" limsid="27-12"/>
<value>1:1</value> </location> <working-flag>true</working-flag> <sample uri="http://your-
server-ip/api/v2/samples/ADM1A3" limsid="ADM1A3"/> <artifact-group name="my_queue"
uri="http://your-server-ip/api/v2/artifactgroups/1"/> </art:artifact></art:details>
```

The XML returned includes the artifact group name and URI:

```
<artifact-group name="my_queue" uri="http://your-server-ip/api/v2/artifactgroups/1"/>
```

Step 3. Move All Samples into a Different Queue

To move the artifacts into another queue, simply update the artifact-group name and URI values:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><art:details
xmlns:udf="http://genologics.com/ri/userdefined" xmlns:file="http://genologics.com/ri/file"
xmlns:art="http://genologics.com/ri/artifact"> <art:artifact uri="http://your-server-
ip/api/v2/artifacts/ADM1A1PA1?state=21" limsid="ADM1A1PA1"> <name>base-1</name>
<type>Analyte</type> <output-type>Analyte</output-type> <qc-flag>UNKNOWN</qc-flag>
<location> <container uri="http://your-server-ip/api/v2/containers/27-10" limsid="27-10"/>
<value>1:1</value> </location> <working-flag>true</working-flag> <sample uri="http://your-
server-ip/api/v2/samples/ADM1A1" limsid="ADM1A1"/> <artifact-group name="new_queue"
uri="http://your-server-ip/api/v2/artifactgroups/2"/> </art:artifact> <art:artifact
uri="http://your-server-ip/api/v2/artifacts/ADM1A2PA1?state=42" limsid="ADM1A2PA1">
```

```
<name>base-2</name> <type>Analyte</type> <output-type>Analyte</output-type> <qc-  
flag>UNKNOWN</qc-flag> <location> <container uri="http://your-server-  
ip/api/v2/containers/27-11" limsid="27-11"/> <value>1:1</value> </location> <working-  
flag>true</working-flag> <sample uri="http://your-server-ip/api/v2/samples/ADM1A2"  
limsid="ADM1A2"/> <artifact-group name="new_queue" uri="http://your-server-  
ip/api/v2/artifactgroups/2"/> </art:artifact> <art:artifact uri="http://your-server-  
ip/api/v2/artifacts/ADM1A3PA1?state=12" limsid="ADM1A3PA1"> <name>base-3</name>  
<type>Analyte</type> <output-type>Analyte</output-type> <qc-flag>UNKNOWN</qc-flag>  
<location> <container uri="http://your-server-ip/api/v2/containers/27-12" limsid="27-12"/>  
<value>1:1</value> </location> <working-flag>true</working-flag> <sample uri="http://your-  
server-ip/api/v2/samples/ADM1A3" limsid="ADM1A3"/> <artifact-group name="new_queue"  
uri="http://your-server-ip/api/v2/artifactgroups/2"/> </art:artifact></art:details>
```

Finally, post the XML back to the server using a batch update request:

```
http://your-server-ip/api/v2/artifacts/batch/update
```

Update UDF/Custom Field Information with Batch Operations

As previously shown in [Update UDF/Custom Field Values for a Derived Sample Output](#), you can update the user-defined fields/custom fields of the derived samples (referred to as analytes in the API) generated by a step. This example uses batch operations to improve the performance of that script.

As of Clarity LIMS v5, the term **user-defined field** (UDF) has been replaced with **custom field** in the user interface. However, the API resource is still called UDF.

There are two types of custom fields:

- **Master step fields**—Configured on master steps. Master step fields only apply to the following:
 - The master step on which the fields are configured.
 - The steps derived from those master steps.
- **Global fields**—Configured on entities (eg, submitted sample, derived sample, measurement, etc.). Global fields apply to the entire Clarity LIMS system.

Prerequisites

Before you follow the example, make sure that you have the following items:

- For Clarity LIMS v4.2 and earlier:
 - A UDF named Library Size on Analyte.
 - A configured Library Prep process that applies Library Size to output analytes (derived samples).
 - A Library Prep process that has been run and has generated derived samples.
- For Clarity LIMS v5 and later:
 - A global custom field named Library Size that on the Derived Sample object.
 - A configured Library Prep step that applies Library Size to generated derived samples.

- A Library Prep step that has been run and has generated derived samples.
- A compatible version of API (v2 r21 or later).

Code Example

For Clarity LIMS v4.2 and earlier, in the Operations Interface, the process summary view displays detailed information about a process and its input and output artifacts. You can view the artifact UDFs associated with the process type by selecting outputs and then opening the Details tab below the Input/Output Explorer.

In the process summary view, the Details tab displays the Library Size values for the selected output analytes. The following example uses batch operations to update these values.

[A14-BMJ]-100830-24-1472] API Cookbook Example 1.4

LIMS ID:

Technician:

Date Run:

Instrument/Software:

Input/Output Explorer

Files

Details

Notes

Inputs

QC	Name	Container Identifier	Vo
+	Colon-1	27-1316	0.0
+	Colon-2	27-1317	0.0
+	Colon-3	27-1318	0.0
+	Colon-4	27-1319	0.0
+	Colon-5	27-1320	0.0
+	Colon-6	27-1321	0.0

Outputs

QC	Name	Container Identifier	Vo
+	Colon-1	27-2317	0.0
+	Colon-2	27-2318	0.0
+	Colon-3	27-2319	0.0
+	Colon-4	27-2320	0.0
+	Colon-5	27-2321	0.0
+	Colon-6	27-2322	0.0

↑

↓

Selected Artifact Details

Sample Genealogy

Details

Details for

☐ Show Non-Detail Fields

Detail Comparison

QC	Name	LIMS ID	Library Size
	Colon-1	HAM75 4A1AP8	3.00
	Colon-2	HAM75 4A2AP8	1.00
	Colon-3	HAM75 4A3AP8	4.00
	Colon-4	HAM75 4A4AP8	1.00
	Colon-5	HAM75 4A5AP8	5.00
	Colon-6	HAM75 4A6AP8	9.00

An analyte resource generally contains specific details recorded in lab processes. Those specific details are often stored in analyte UDFs. Analyte UDFs must also be associated with a process. When you update an analyte, only the analyte UDFs that are configured for the process that created the analyte can be updated.

In Clarity LIMS v5 or later, the Record Details screen displays the information about the derived samples generated by a step. You can view the global fields associated with the derived samples in the Sample table.

The following screenshot shows the Library Size values for the derived samples.

JP Cookbook Example
Record Details (8 samples)

System Administrator

Started
Updated
Apr 21, 2017 10:35:21
Apr 21, 2017 10:35:22

Step Data Table

No Step Details to Record

Sample Table

Container Name

Apply

Sample Name

Container Name

Well

LIMS ID (Container)

Project Name

Library Size

Single Well: Tube (8 samples)

Kidney-1060Z5	27-204		27-204	JP Project	3.00
Kidney-1176E2	27-202		27-202	JP Project	1.00
Kidney-1332L4	27-203		27-203	JP Project	4.00
Kidney-1400L5	27-206		27-206	JP Project	1.00
Kidney-1618C6	27-209		27-209	JP Project	5.00
Kidney-1680A4	27-208		27-208	JP Project	9.00

Derived sample information is stored in the API in the analyte resource. Step information is stored in the process resource. Each global field value is stored as an udf.

An analyte resource contains specific derived sample details that are recorded in lab steps. Those details are typically stored in global fields, configured in Clarity LIMS on the Derived Sample object and then associated with the step. When you update the information for a derived sample by updating the analyte API resource, only the global fields that are associated with the step can be updated.

Step 1. Retrieve the Process Information

To retrieve the process information, you can perform a GET on the created process URI as follows.

```
processURI = "http://${hostname}/api/v2/processes/${processLIMSID}" process =  
GLSRestApiUtils.httpGET(processURI, username, password)
```

Step 2. Retrieve URIs of Output Analytes

You can now collect the output analytes and harvest their URIs.

Step 3. Retrieve Analytes

After you have collected the output analyte URIs, you can retrieve the analytes with a `batchGET()` operation. The URIs must be unique for the batch operations to succeed.

```
def outputNodes = GLSRestApiUtils.batchGET(outputAnalyteURIs, username, password)
```

Step 4. Set Analyte Library Size UDF/Custom Field

You can now iterate through our retrieved list of analytes and set each analytes Library Size UDF/custom field to 25.

```
outputNodes.each { GLSRestApiUtils.setUdfValue(it, 'Library Size', '25') }
```

Step 5. Update Analytes

To update the analytes in the system, call `batchPUT()` . It attempts to call a PUT for each node in the list. Each node must be unique.

```
GLSRestApiUtils.batchPUT(outputNodes, username, password)
```

Expected Output and Results

In Clarity LIMS v4.2 and earlier, in the `OperationsInterface`, the process summary view now shows the updated Library Size.

[A14-BMJ]-100830-24-1472] API Cookbook Example 1.4

LIMS ID: A14-BMJ-100830-24-1472

Technician: Brandon Johnson

Date Run: Aug 30, 2010

Instrument/Software:

Input/ Output Explorer

Files

Details

Notes

Inputs

QC	Name	Container Identifier	Vol
+	Colon-1	27-1316	0.0
+	Colon-2	27-1317	0.0
+	Colon-3	27-1318	0.0
+	Colon-4	27-1319	0.0
+	Colon-5	27-1320	0.0
+	Colon-6	27-1321	0.0

Outputs

QC	Name	Container Identifier	Vol
+	Colon-1	27-2317	0.0
+	Colon-2	27-2318	0.0
+	Colon-3	27-2319	0.0
+	Colon-4	27-2320	0.0
+	Colon-5	27-2321	0.0
+	Colon-6	27-2322	0.0

Adapter Ligation for Tra... Run Process...

Selected Artifact Details

Sample Genealogy

Details

Details for

Output per Input: Analyte

Show Non-Detail Fields

Detail Comparison

QC	Name	LIMS ID	Library Size
	Colon-1	HAM754A1AP8	25.00
	Colon-2	HAM754A2AP8	25.00
	Colon-3	HAM754A3AP8	25.00
	Colon-4	HAM754A4AP8	25.00
	Colon-5	HAM754A5AP8	25.00
	Colon-6	HAM754A6AP8	25.00

In Clarity LIMS v5 and later, in the Record Details screen, the Sample table now shows the updated Library Size.

Sample Table					
Sample Name ▲		Container Name	Well	LIMS ID (Container)	Project Name
▼ Single Well: Tube (8 samples)					
Kidney-1060Z5	27-204	27-204	JP Project	25.00	
Kidney-1176E2	27-202	27-202	JP Project	25.00	
Kidney-1332L4	27-203	27-203	JP Project	25.00	
Kidney-1400L5	27-206	27-206	JP Project	25.00	
Kidney-1618C6	27-209	27-209	JP Project	25.00	
Kidney-1680A4	27-208	27-208	JP Project	25.00	

Attachments

[UsingBatchPut.groovy.txt](#)

Retrieve Multiple Entities with a Single API Interaction

In Clarity LIMS, you want to process multiple entities. To accomplish this quickly and effectively, you can use batch operations, which allows you to retrieve multiple entities using a single interaction with the API, instead of iterating over a list and retrieving each entity individually.

Batch operations greatly improve the performance of the script. These methods are available for containers and artifacts. In this example, both entities are retrieved using the `batchGet()` operation. If you would like to update a batch of output analytes (derived samples), you can increase the script execution speed by using batch operations. For more information, refer to [Batch Resources Overview](#) and [Introduction to Batch Resources](#).

Prerequisites

Before you follow the example, make sure that you have the following items:

- Several samples that have been added to Clarity LIMS.
- A process/step that generates derived samples in containers has been run on the samples.
- A compatible version of API (v2 r21 or later).

Code Example

When derived samples ('analyte artifacts' in the API) are run through a process / step, their information can be accessed by examining that process / step. In this example, you retrieve the input artifacts and their respective containers.

To do this effectively using batch operations, we must collect the URIs of the entities. These URIs must be unique, otherwise the batch operation fails. Then, the entities can be retrieved in one action. It is important to note that only one type of entity can be retrieved in a call.

Groovy Example

Step 1. Retrieve the Process Information

To retrieve the process step information, use the GET method with the process LIMS ID:

```
processURI = "http://${hostname}/api/v2/processes/${processLIMSID}" processNode =  
GLSRestApiUtils.httpGET(processURI, username, password)
```

Step 2. Retrieve the Artifact URIs

To retrieve the artifact URIs, collect the inputs of the input-output-map of the process. A condition of the batchGET operation is that every entity to get must be unique. Therefore, you must call unique on your list.

```
def inputUris = processNode.'input-output-map'.collect { it.'input'[0].@uri }.unique()
```

Step 3. Retrieve Unique Input Analytes and Their Containers

You can now use batchGET to retrieve the unique input analytes:

```
def inputNodes = GLSRestApiUtils.batchGET(inputUris, username, password)
```

The same can be done to gather the containers of the analytes:

```
def containerUris = inputNodes.collect { it.'location'.'container'[0].@uri }.unique() def  
containerNodes = GLSRestApiUtils.batchGET(containerUris, username, password)
```

Expected Output and Results

You have collected the unique containers in which the artifacts are located. By printing the name and URI of each container, an output similar to the following is obtained.

```
AutomatedTestContainer-549374968 http://yourIpAddress/api/v2/containers/27-1478
```

Python Example

Step 1. Retrieve the Step Information

To retrieve the step information, use the GET method with the step LIMS ID:

```
stepXML = api.GET( stepURI + "/details" )
```

Step 2. Retrieve the Artifact URIs

To retrieve the artifact IDs, collect the inputs of the **input-output-map** of the step. A condition of the batch retrieve operation is that every entity to get must be unique. To do this, you add the LUIDs to a set().

```
LUIDs = set() for inputArtifact in parseString( stepXML ).getElementsByTagName( "input" ) :  
artifactLUID = inputArtifact.getAttribute( "limsid" ) LUIDs.add( artifactLUID )
```

Step 3. Retrieve Unique Input Analytes

You can now use the function getArtifacts(), which is included in the glsapiutils.py to retrieve the unique input analytes:

```
batchXML = api.getArtifacts( LUIDs )
```

Attachments

- [UsingBatchGet.groovy.txt](#)
- [Batchexample.py.txt](#)

Select the Optimal Batch Size

The Clarity LIMS API has batch retrieve endpoints for samples, artifacts, containers, and files. This article talks generically about links for any of those four entities.

When using the batch endpoints, you want to process upwards of hundreds of links. Intuitively, you may think that a single API call with all the links would be the fastest way to retrieve the data. However, analysis of the API performance shows that as the number of links increases beyond a threshold, the time per object increases.

To retrieve the data in the most efficient way, it is best to do multiple POSTs containing the optimal sized batch. A batch call takes longer than a GET to the endpoint of the sample to retrieve the data for a single sample (or other entity). However, after more than one or two samples are needed, the batch endpoint is more efficient.

Prerequisites

Before you follow the example, make sure that you are aware of what the optimal batch size is based on the following information:

- The optimal size depends on your specific server and the number of UDFs/custom fields or other data attached to the object being retrieved.
- The optimal batch size can be different for artifacts, samples, files, and containers. For example, if the optimal size for samples is 500, 10 batches of 500 samples retrieve the data faster than one batch of 5000.

You must also have a compatible version of API (v2 r21 or later).

Determine Optimal Batch Size

Attached is a simple python script that will time how long batch retrieve takes for an array of batch sizes. The efficiency is measured by the duration of the call divided by the number of links posted.

Hard-Coded Parameters

The attached script has hard coded parameters to define the range and increments of batch sizes to test. Also, the number of replications for each size is adjustable. These parameters are found on line 110, and may not require any modification because they are already set to the following by default:

```
replications = 3 # how many times each batch will be measured
repetitions = 1 # how many measurements will be taken for each batch size
R = range( 100, 300 ) # range of the batch sizes to be measured (where min >= 1)
q = 25 # batch size incremental increase
```

For example, the previous parameters test the following sizes: 100, 125, 150, 175, 200, 225, 250, 275.

Command Line Parameters

The parameters that will must specific to the server are entered at the command line.

- **-u**—Username
- **-p**—Password
- **-s**—Hostname, including /api/v2
- **-t**—Entity (eg, artifact, sample, file, or container)

An example of the full syntax to invoke the script is as follows.

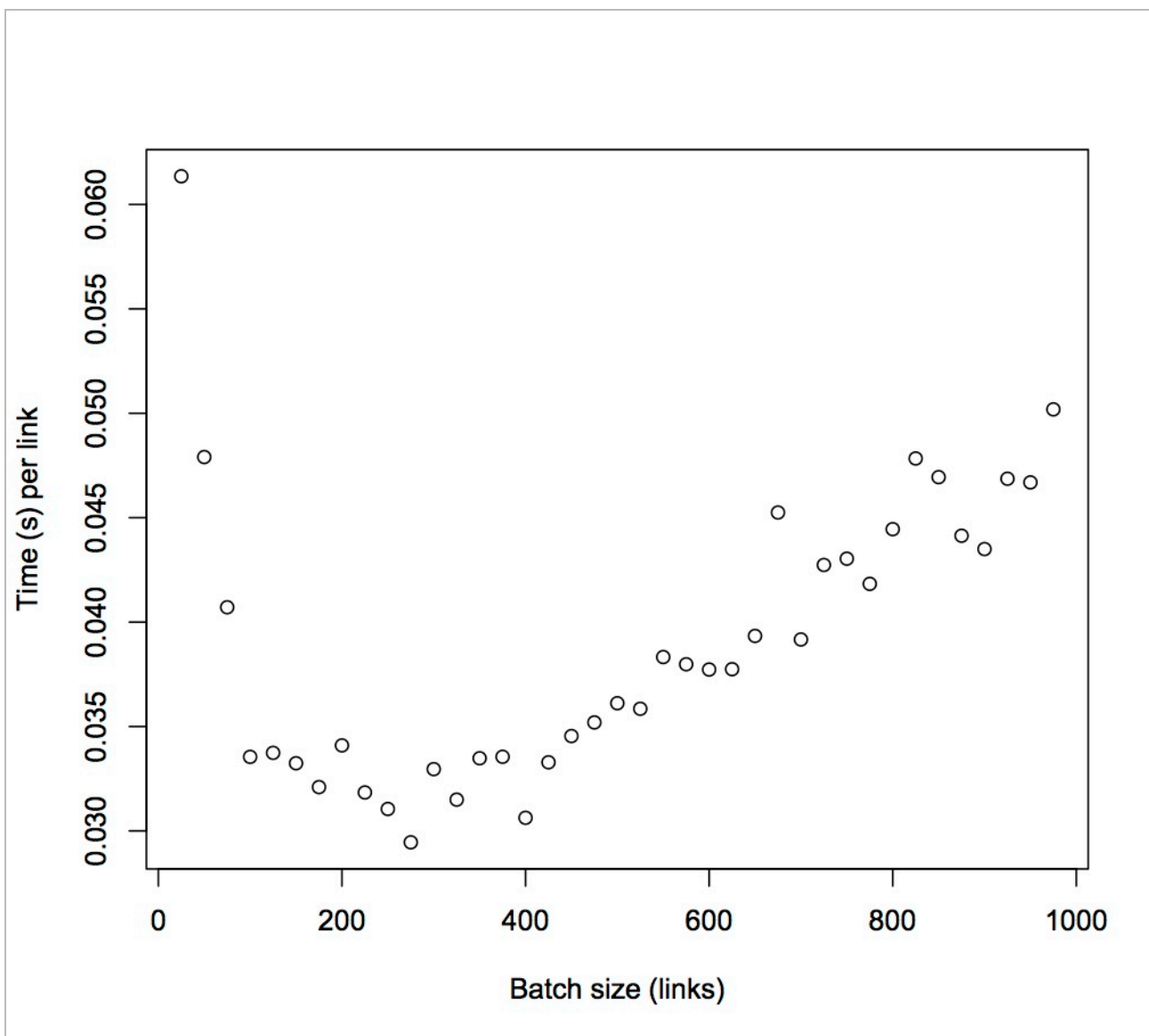
```
python BatchOptimalSizeTest.py -p apipassword -u apiuser -s
https://demo.basespacelims.com/api/v2 -t artifact
```

Expected Output and Results

The script tracks how long each batch call takes to complete. The script outputs a .txt file with the raw numeric data and the batch size that returns the minimum value, and is the most efficient.

```
Analyzing results for: artifactBatch sizes:[25, 50, 75, 100, 125, 150, 175, 200, 225, 250,
275, 300, 325, 350, 375, 400, 425, 450, 475, 500, 525, 550, 575, 600, 625, 650, 675, 700,
725, 750, 775, 800, 825, 850, 875, 900, 925, 950, 975]Time (s) per entity:
[0.061350816726684576, 0.04790449237823486, 0.040710381189982096, 0.03354618215560913,
0.033738230133056636, 0.03324082946777344, 0.03209760447910854, 0.03409448790550232,
0.03184072346157498, 0.031050360870361327, 0.029453758586536753, 0.03295832395553589,
0.03149744004469652, 0.03347888347080776, 0.033550281016031906, 0.030628018498420718,
0.03328620989182416, 0.03454347112443712, 0.035195479945132606, 0.0361147011756897,
0.03584921982174828, 0.0383262753053145, 0.037979933946029, 0.03772751696904501,
0.03774445213317871, 0.03933756652245155, 0.04524845660174335, 0.03916741977419172,
0.04273618560001768, 0.043037356503804525, 0.04183078679730815, 0.044450711250305176,
0.0478362009453051, 0.04694189671909108, 0.044135747201102124, 0.04349724955028958,
0.04686621408204775, 0.046690188458091336, 0.05018808247492863] Duration (s) of batch call:
[1.5337704181671143, 2.395224618911743, 3.053278589248657, 3.354618215560913,
4.21727876663208, 4.986124420166016, 5.617080783843994, 6.8188975811004635,
7.1641627788543705, 7.762590217590332, 8.099783611297607, 9.887497186660767,
10.236668014526368, 11.717609214782716, 12.581355381011964, 12.251207399368287,
14.146639204025268, 15.544562005996704, 16.717852973937987, 18.057350587844848,
18.820840406417847, 21.079451417922975, 21.838462018966673, 22.636510181427003,
23.590282583236693, 25.569418239593507, 30.54270820617676, 27.417193841934203,
30.983734560012817, 32.278017377853395, 32.418859767913816, 35.56056900024414,
39.46486577987671, 39.90061221122742, 38.61877880096436, 39.14752459526062,
43.351248025894165, 44.35567903518677, 48.93338041305542]275 artifacts was the most
efficient batch size
```

Viewing this data in a scatterplot format, you can see the range of optimal batch sizes for the artifacts/batch/retrieve endpoint is about 200 to 300 artifacts. This would be valid for artifacts only and each entity (eg, sample, file, or container) should be evaluated separately.



The shortest time per artifact is the most efficient batch size, as shown in the following example:

```
275 artifacts was the most efficient batch size
```

Proxy Timeout

By default, the Clarity LIMS configuration of send and receive timeout is 60 seconds. Large batch calls will not complete if their duration is greater than the timeout configuration. This configuration is at

/etc/httpd/conf/httpd.conf.

Attachments

[BatchOptimalSizeTest.py.txt](#)

Work with Files

When working with files, you can do the following:

- [Attach a File with REST and Python](#)
- [Attach Files Located Outside the Default File Storage Repository](#)
- [Attach a File to a File Placeholder with REST](#)

Attach a File with REST and Python

This example attaches a file to the Clarity LIMS server file storage repository instead of linking to an existing file on the network. For more information on linking to an existing file, refer to is covered in [Attach Files Located Outside the Default File Storage Repository](#).

Both examples are useful in practice, depending on your network, storage architecture, and the size of the file. The file attachment method used in this example is equivalent to a lab scientist manually importing a file and attaching it to a file placeholder in the Clarity LIMS user interface.

Prerequisites

Before you follow the example, make sure you have the following items:

- Files that can be attached to projects, steps, or result file artifacts.
- The script that uses the Python package Requests and relies upon `glsapiutil.py`.
- A compatible version of API (v2).

Use EPP/Automation to Attach Files

You can also use EPP/automation to handle attaching files. Though this method is not as flexible, it does attach a file automatically. With this method, the attached files are copied to the Clarity LIMS server and attached based on a LIMS ID in the created file name. Automation scripts are powerful, but they are only called when the process/step is created, whereas the code in this example can run at any time. For more information, refer to [Attach a File to a File Placeholder with REST](#).

Code Example

Attaching a file requires the use of `glsstorage` and `files` resources.

The `glsstorage` resource assigns a file location, much like a file placeholder in the user interface. The `files` resource is used to associate the physical disk location to the `ResultFile` artifact. In combination, these resources allow flexible management of files, and the integration of external file manipulation and transfer tools.

Attaching a file is done in three main steps:

1. Using a POST, create a storage location for the file to `glsstorage`.

This returns the XML that POSTs to the files resource. The XML includes a content-location.

2. Link the file to the placeholder (creating a unique file LIMS ID) by POSTing the glsstorage response to the `/api/v2/files` endpoint.
3. Using the `/api/v2/files/{limsid}/upload` endpoint, upload the file.

Attachments

- [replace_file_APPLICATION_EXAMPLE.py.txt](#)
- [rest.version.files](#)
- [rest.version.glsstorage](#)
- [rest.version.files.limsid.upload](#)

Attach Files Located Outside the Default File Storage Repository

In Next Generation Sequencing (NGS), a large amount of data is generated in a single run. Because the volume of data is so large, it makes sense to link to data files as they exist on the network, rather than copying files to the Clarity LIMS file server.

This example shows how you can use the files resource to associate a process output result file with a file on the network using the HTTP protocol.

Prerequisites

Before you follow the example, make sure you have the following items:

- Samples that have been added to the system.
- A NGS process that takes analyte (derived sample) inputs and creates a single result file output for each input.
- A flow cell container with 8 rows and 1 column. Both the rows and columns are numbers and start at 1.
- Samples that have been added to a flow cell and have run the flow cell through your NGS process.
- An HTTP file store that has been set up in the Clarity LIMS configuration file.
- The appropriate directory structure and files to be linked to your HTTP file store. For this example, you need a directory with the same name as the flow cell on which you run your process.
 - You require a subdirectory for each lane named by lane number (eg, 1 containing the file you want to link).
 - The name of the file must be easily programmatically determined (eg, `s_1_export.txt` for lane number 1).
- A compatible version of API (v2 r14 to v2 r24).

Setup

This example assumes you are using the standard non-production scripting sandbox server, which uses Apache to serve files with HTTP. For more information on this server, refer to [Useful Tools](#).

If you plan to use an alternative file storage configuration, contact the IT administrator of the Clarity LIMS server.

The administrator uses the `omxprops-ConfigTool.jar` configuration tool. For more information on this configuration tool, refer to *Non-Production Scripting Sandbox Server - IT Admin Guide*.

The `omxprops-ConfigTool.jar` tool is at `/opt/gls/clarity/tools/propertytool/`. For more information, refer to [Remote HTTP Filestore Setup](#).

Code Example

After running a NGS process on a full flow cell, in the Operations Interface, the Input/Output Explorer for the process run shows the relationship between the inputs in the flow cell and the result file output placeholders.

The screenshot shows the 'Input/Output Explorer' window for the process 'NEX-SA1-101126-24-1'. The window is divided into two main sections: 'Inputs' and 'Outputs'.

Inputs Table:

QC	Name	Container Identifier	Placement	Vol (µL)	Conc (µg/...
	27-18-1	27-18-1			
	diseased-liver-1	27-18-1	1:1	0.0	100
	diseased-liver-2	27-18-1	2:1	0.0	100
	diseased-liver-3	27-18-1	3:1	0.0	100
	diseased-liver-4	27-18-1	4:1	0.0	100
	diseased-liver-5	27-18-1	5:1	0.0	100
	diseased-liver-6	27-18-1	6:1	0.0	100
	diseased-liver-7	27-18-1	7:1	0.0	100
	diseased-liver-8	27-18-1	8:1	0.0	100

Outputs Table:

QC	Name	Container Identifier	Original Filename	Project	Submitted Sample	QC Pass	QC Fail	Sample LIMS ID
	[ADM1A1NE3] results.txt			Test Pr...	diseas...			ADM1A1N...
	[ADM1A2NE3] results.txt			Test Pr...	diseas...			ADM1A2N...
	[ADM1A3NE3] results.txt			Test Pr...	diseas...			ADM1A3N...
	[ADM1A4NE3] results.txt			Test Pr...	diseas...			ADM1A4N...
	[ADM1A5NE3] results.txt			Test Pr...	diseas...			ADM1A5N...
	[ADM1A6NE3] results.txt			Test Pr...	diseas...			ADM1A6N...
	[ADM1A7NE3] results.txt			Test Pr...	diseas...			ADM1A7N...
	[ADM1A8NE3] results.txt			Test Pr...	diseas...			ADM1A8N...

Associating a file that resides on a server that is accessible via the HTTP protocol requires the following steps:

1. Matching up the file with the correct file placeholder.
2. Constructing the XML to POST to the files resource.
3. A POST to the files resource, which associates the file and creates the link between the artifact and the file.

Step 1. Match the File with the Correct File Placeholder

The following code snippet shows how, by starting with only the name of the flow cell, we can obtain the API resource for the flow cell container and the NGS process run.

Because the location of all the files we want to link is known, the combination of the flow cell and NGS process contains all the additional information required to link each file to the correct file placeholder, so that the results are associated with the correct sample.

1. Get a containers list filtering by container name, where the name is equal to the name of the directory.

Your container name must be unique in the system as only one result is expected by this script.

```
def containers = GLSRestApiUtils.httpGET("${baseURI}containers?name=${containerName}",  
username, password) println(containers)
```

2. Get the full XML for the container.

The container list only contains the URI and LIMS ID of the container and we require all the placements as well.

```
def container = GLSRestApiUtils.httpGET(containers.'container'[0].@uri, username, password)
```

- Using the LIMS ID of one of the placements in the container, use the API to find the processes that have used it as an input.

For this example, the expectation is that only one process has been run on the flow cell. If that is not the case, the first one returned is used.

```
def processes = GLSRestApiUtils.httpGET
("${baseURI}processes?inputartifactlimsid=${container.'placement'[0].@limsid}", username, password)
```

- Complete the XML with the input-output-map and make another call to the API to retrieve the complete process.

As for the container case, the list resource just gives us the URI and LIMS ID of the process.

```
def process = GLSRestApiUtils.httpGET(processes.'process'[0].@uri, username, password)
```

The code in this example includes some helper closures and a persistent single HTTP connection. The persistent connection helps improve the performance of the script.

The location of the file to be linked is constructed for each POST, based on the directory structure. For each input-output-map element in the process, it is stored in the FileURI variable. The value provided here is used in both the <content-location> and <original-location> elements in the file node that is posted.

```
String placement = container.placement.find{ it['@limsid'] == inputlimsid }.value[0].text
() String laneNum = placement.split(':')[0] String fileURI =
"${illuminaRunDirectoryURI}/${laneNum}/s_${laneNum}_export.txt"
```

The location of the result file artifact is stored in the variable outputURI. It is obtained from the <output> element in the input-output-map as follows.

```
String outputURI = it.output[0].@uri.split("\\?")[0]
```

Step 2. Construct the XML to POST to the Files Resource

The first requirement is to create an XML resource for the file you want to submit to the system.

As in the example from [Attach a File to a File Placeholder with REST](#), you create the resource using StreamingMarkupBuilder, which you use in a POST to the files resource.

When you construct the individual file XML, you must specify the content-location, attach-to, and original-location child nodes.

In this example, you are posting a file link to a file in an HTTP file storage location, so the content-location and original-location are the same.

```
def fileBuilder = new StreamingMarkupBuilder() fileBuilder.encoding = "UTF-8" def fileXML
= fileBuilder.bind { mkp.xmlDeclaration() mkp.declareNamespace(file:
'http://genomics.com/ri/file') 'file:file'{ 'content-location'(fileURI) 'attached-to'
(outputURI) 'original-location'(fileURI) } }
```

Step 3. POST to the Files Resource

The POST requires an XML node as input. Therefore, you must convert fileDoc from a writable closure to an XML node using `GLSGeneusRestApiUtils.xmlStringToNode`.

You can then submit the new file resource to the API via a POST as follows.

```
// Convert the fileXML to a node and link the file by posting the node to the files
resource in the API Node fileNode = new XmlParser().parseText(fileXML.toString())
GLSRestApiUtils.httpPOST(fileNode, "${baseURI}files", username, password)
```

The following example shows the XML for FileNode, which is used in the POST:

```
<?xml version='1.0' encoding='UTF-8'?><file:file
xmlns:file='http://genologics.com/ri/file'> <content-location>http://Root of your http
file store/illumina/27-18-1/1/s_1_export.txt</content-location> <attached-
to>http://yourIPAddress/api/v2/artifacts/ADM1A1NE3</attached-to> <original-
location>http://Root of your http file store/illumina/27-18-1/1/s_1_export.txt</original-
location></file:file>
```

The POST command does the following:

- POSTs the XML constructed by StreamingMarkupBuilder to the files resource.
- Adds a link from the files list resource to the new file.
- After the POST executes, a link to the new file resource is added to the ResultFile artifact resource because it was specified in the attached-to field of the new file resource.

The following code is the XML for the ResultFile artifact after the POST, which contains a link to the new file resource on the last line:

```
<art:artifact uri="http://yourIPAddress/api/v2/artifacts/ADM1A1NE3?state=48"
limsid="ADM1A1NE3"> <name>results.txt</name> <type>ResultFile</type> <output-
type>ResultFile</output-type> <parent-process
uri="http://yourIPAddress/api/v2/processes/NEX-SA1-101126-24-1" limsid="NEX-SA1-101126-
24-1"/> <qc-flag>UNKNOWN</qc-flag> <sample
uri="http://yourIPAddress/api/v2/samples/ADM1A1" limsid="ADM1A1"/> <file:file
limsid="ADM1A1NE3-40-112" uri="http://yourIPAddress/api/v2/files/ADM1A1NE3-40-
112"/></art:artifact>
```

The file resource available from the API is slightly altered from the XML submitted to the POST. The file is assigned a LIMS ID and a URI.

The following code is the XML resource for the file that is available from the API:

```
<file:file uri="http://yourIPAddress/api/v2/files/ADM1A1NE3-40-112" limsid="ADM1A1NE3-40-
112"> <attached-to>http://yourIPAddress/api/v2/artifacts/ADM1A1NE3</attached-to>
<content-location>http://Root of your http file store/illumina/27-18-1/1/s_1_
export.txt</content-location> <original-location>http://Root of your http file
store/illumina/27-18-1/1/s_1_export.txt</original-location> <is-published>false</is-
published></file:file>
```

The POST to the files resource associates a physical file to the ResultFile artifact. In the user interface, this POST changes the file icon.

⚠ Before POSTing to the files resource, you must make sure that the file exists in the location referenced by the `content-location` element. If the file does not exist in this location, the POST fails.

Expected Output and Results

The value returned from the POST is stored in the `returnNode` variable.

If the command was successful, the above XML is returned. If the command was unsuccessful, an XML document explaining the error is returned. For example, if a file was already attached to the result file at `artifactURI`, the following document is stored in `returnNode`, as returned from the POST. In this case, the file that is already attached to the artifact must first be removed via the file resource DELETE method.

```
<exc:exception xmlns:exc="http://genologics.com/ri/exception"> <message>Removing file in this manner not permitted</message></exc:exception>
```

After a successful POST HTTP command, in the Operations Interface, the Input/Output Explorer of the process summary view shows all the attached files.

[NEX-SA1-101126-24-1] NextGen Sequencing

LIMS ID: NEX-SA1-101126-24-1

Technician: System Administrator

Date Run: Nov 26, 2010

Instrument/Software:

Input/Output Explorer

Files

Details

Notes

Inputs

QC	Name	Container Identifier	Placement	Vol (µL)	Conc (µg/...
	27-18-1	27-18-1			
	diseased-liver-1	27-18-1	1:1	0.0	100
	diseased-liver-2	27-18-1	2:1	0.0	100
	diseased-liver-3	27-18-1	3:1	0.0	100
	diseased-liver-4	27-18-1	4:1	0.0	100
	diseased-liver-5	27-18-1	5:1	0.0	100
	diseased-liver-6	27-18-1	6:1	0.0	100
	diseased-liver-7	27-18-1	7:1	0.0	100
	diseased-liver-8	27-18-1	8:1	0.0	100

Outputs

QC	Name	Container Identifier	Original Filename	Project	Submitted Sample	QC Pass	QC Fail	Sample LIMS ID	Cont LIMS ID
	[ADM1A1NE3] re...		s_1_export.txt	Test Pr...	diseas...			ADM1A1N...	
	[ADM1A2NE3] re...		s_2_export.txt	Test Pr...	diseas...			ADM1A2N...	
	[ADM1A3NE3] re...		s_3_export.txt	Test Pr...	diseas...			ADM1A3N...	
	[ADM1A4NE3] re...		s_4_export.txt	Test Pr...	diseas...			ADM1A4N...	
	[ADM1A5NE3] re...		s_5_export.txt	Test Pr...	diseas...			ADM1A5N...	
	[ADM1A6NE3] re...		s_6_export.txt	Test Pr...	diseas...			ADM1A6N...	
	[ADM1A7NE3] re...		s_7_export.txt	Test Pr...	diseas...			ADM1A7N...	
	[ADM1A8NE3] re...		s_8_export.txt	Test Pr...	diseas...			ADM1A8N...	

Attachments

[AttachingFileNotOnClarity.groovy.txt](#)

Attach a File to a File Placeholder with REST

Many lab processes create small files to summarize results. This example attaches a file to the Clarity LIMS server file storage repository, rather than linking to an existing file on the network. Linking to an existing file is covered in [Attach Files Located Outside the Default File Storage Repository](#). Both examples are useful in practice, depending on your network, storage architecture, and the size of the file.

The file attachment method used in this example is equivalent to a lab scientist manually importing a file and attaching it to a file placeholder in the Clarity LIMS user interface.

! | Before POSTing to the files resource, you must make sure that the file exists in the location referenced by the `content-location` element. If the file does not exist in this location, the POST fails.

Use EPP/Automation to Attach Files

You can also use EPP/automation to handle attaching files. Though this method is not as flexible, it does attach a file automatically. With this method, the attached files are copied to the Clarity LIMS server and attached based on a LIMS ID in the created file name. Automation scripts are powerful, but they are only called when the process/step is created, whereas the code in this example can run at any time. For more information, refer to [Attach a File to a File Placeholder with REST](#).

Prerequisites

Before you follow the example, make sure that you have the following items:

- Samples that have been added to the system.
- A process with analytes (derived samples) as inputs and result files as outputs that has been run on some samples.
- The file to be attached (example uses results.csv) that exists in the same directory from which the script is executed.
- The JSch library that has been imported onto the classpath. JSch is used by some Groovy closures (refer to fileSFTPHelper) in the attached script for SFTP logic. For more information, refer to <http://www.jcraft.com/jsch/>.
- A compatible version of API (v2 r14).

Code Example

After you run a process with a result file output, a file placeholder icon displays next to the result file. The following image shows this in the Clarity LIMS Operations Interface.

[AFF853A40] Calcaneus-1

LIMS ID:

Project:

Sample Name:

Total Volume (µL):

Reagent Types:

Date Received:

Date Completed:

Genealogy

Files

Details

Notes

Biosource

Experimental Conditions

Sample Genealogy

Name	Date Run	LIMS ID
Calcaneus-1		AFF853A40
API Cookbook Example 5.4	Sep 21, 2010	A54-BMJ-100921-24-2178
Calcaneus-1		AFF853A40AP2

Attaching a file to a process output requires the use of `glsstorage` and `files` resources.

`glsstorage` assigns a file location, much like a file placeholder in the user interface. The `glsstorage` POST is a request to create a unique file name (name and directory) for a future disk file. The `files` resource is used to associate the physical disk location to the `ResultFile` artifact. In combination, these resources allow flexible management of files, and the integration of external file manipulation and transfer tools.

You can attach a file as follows.

1. Create a storage location for the file, using a POST to `glsstorage`. This returns the XML that POSTs to the `files` resource. The XML includes a `content-location`.
2. Copy the file to the new storage location.
3. POST the file XML to attach the file to the result file placeholder.

Step 1. Create a Storage Location for the File

The first step in attaching a file to the placeholder is to create a location for the file on the LIMS server, using a POST to the `glsstorage` resource. A POST to `glsstorage` requires the `attached-to` and `original-location` child elements to be defined within the file XML content. (These are the only two elements required to post to `glsstorage`.)

The file XML content is created using `StreamingMarkupBuilder` in Groovy. Our example code begins by defining the current location of the file in the variable `fileOriginalLocation`. In addition, the `artifactURI` variable is defined as the URI with which to associate the file. In this case, the resource is a `ResultFile` artifact.

```
// Create a new file Node using StreamingMarkupBuilder def fileDoc = new
StreamingMarkupBuilder().bind { mkp.xmlDeclaration() mkp.declareNamespace
(file:'http://genomics.com/ri/file') 'file:file'{ 'attached-to'("${artifactURI}")
'original-location'("${fileOriginalLocation}") } }
```

Before the POST to `glsstorage`, the previously created XML appears as follows:

```
<file:file xmlns:file="http://genologics.com/ri/file"> <attached-to>http://yourIPAddress/api/v2/artifacts/AFF853A40AP2</attached-to> <original-location>/home/glsftp/Testing/results.csv</original-location></file:file>
```

The following code shows the POST to glsstorage. The XML returned by the POST to glsstorage is stored in the variable `resolvedFileNode`.

```
// Post the file Node to the API unresolvedFileNode = GLSRestApiUtils.xmlStringToNode
(fileDoc.toString()) resolvedFileNode = GLSRestApiUtils.httpPOST(unresolvedFileNode,
"${apiIndexURI}/glsstorage", username, password) println GLSRestApiUtils.nodeToXmlString
(resolvedFileNode)
```

The XML returned by the POST method includes a new `content-location` child element. The `content-location` is the new directory and file name on the Clarity LIMS server to which the file should be copied.

```
<file:file xmlns:file="http://genologics.com/ri/file"> <content-
location>sftp://yourIPAddress/home/glsftp/Process/2010/9/A54-BMJ-100921-24-
2178/AFF853A40AP2-40-2911.csv</content-location> <attached-
to>http://yourIPAddress/api/v2/artifacts/AFF853A40AP2</attached-to> <original-
location>/home/glsftp/Testing/results.csv</original-location></file:file>
```

If the POST to glsstorage was unsuccessful, an XML document explaining the error is returned. For example, if the artifact specified by `artifactURI` does not exist, then the POST fails, and `resolvedFileNode` holds the content as follows.

```
<exc:exception xmlns:exc="http://genologics.com/ri/exception"> <message>Requested
artifact not found</message></exc:exception>
```

Step 2. Copy the File to the New Storage Location

The next step is to copy the file from current to new location. SFTP is used through the `fileSFTPHelper` closure.

```
// Copy the file from its current location def fileFinalLocation = new URI
(resolvedFileNode.'content-location'[0].text()) File sourceFileObj = new File
(fileOriginalLocation) def sftpSuccess = fileSFTPHelper(fileFinalLocation, sourceFileObj,
glsftpUsername, glsftpPassword) println("SFTP success : $sftpSuccess")
```

If there has never been a file attached to the `ResultFile` placeholder, the directory specified by the `content-location` element will not exist.

The `fileSFTPHelper` Groovy closure takes care of creating the directory required, using the `destRemoteFileURI` parameter. It then copies the file to the new file location using SFTP.

To make sure that the copy was successful, you can make sure that the `$sftpSuccess` variable was true.

i | The `glsstorage` methods do not create, move, or delete disk files. Use file operations that fit within your scripting or system automation practices.

Step 3. Attach File to the Result File Placeholder

After you SFTP the file to the `content-location`, you can POST `resolvedFileNode` to the files (list) REST resource.

POSTs to this resource require file XML content with the `attached-to`, `original-location`, and `content-location` child elements defined.

After the POST to the files resource, the file information is attached to the resource specified in the `attached-to` child element:

```
// Post the file node to the API returnNode = GLSRestApiUtils.httpPOST(resolvedFileNode,
"${apiIndexURI}/files", username, password) println GLSRestApiUtils.nodeToXmlString
(returnNode)
```

i | The POST to the files (list) REST resource associates a physical file with the `ResultFile` artifact. In the user interface, this POST changes the icon from a placeholder to an attached file.

If the POST was successful, the updated XML is in the `returnNode` variable, and contains a LIMS ID and a URI, as shown in the following example:

```
<file:file uri="http://yourIPaddress/api/v2/files/AFF853A40AP2-40-2908"
limsid="AFF853A40AP2-40-2908"> <content-
location>sftp://yourIPaddress/home/glsftp/Process/2010/9/A54-BMJ-100921-24-
2178/AFF853A40AP2-40-2907.csv</content-location> <attached-
to>http://yourIPaddress/api/v2/artifacts/AFF853A40AP2</attached-to> <original-
location>/home/glsftp/Testing/results.csv</original-location> <is-published>>false</is-
published></file:file>
```

The artifact resource for the result file now contains a file element with the file LIMS ID and URI:

```
<art:artifact uri="http://yourIPaddress/api/v2/artifacts/AFF853A40AP2?state=20968"
limsid="AFF853A40AP2"> <name>Calcaneus-1</name> <type>ResultFile</type> <output-
type>ResultFile</output-type> <parent-process
uri="http://yourIPaddress/api/v2/processes/A54-BMJ-100921-24-2178" limsid="A54-BMJ-
100921-24-2178"/> <qc-flag>UNKNOWN</qc-flag> <sample
uri="http://yourIPaddress/api/v2/samples/AFF853A40" limsid="AFF853A40"/> <file:file
limsid="AFF853A40AP2-40-2908" uri="http://yourIPaddress/api/v2/files/AFF853A40AP2-40-
2908"/></art:artifact>
```

Expected Output and Results

When the script completes, an attached file icon displays in the Clarity LIMS Operations Interface. The lab scientist can download the file from this view.

[AFF853A40] Calcaneus-1

LIMS ID:

Project:

Sample Name:

Total Volume (µL): Reagent Types:

Date Received: Date Completed:

Genealogy | **Files** | **Details** | **Notes** | **Biosource** | **Experimental Conditions**

Sample Genealogy

Name	Date Run	LIMS ID
Calcaneus-1		AFF853A40
API Cookbook Example 5.4	Sep 21, 2010	A54-BMJ-100921-24-2178
Calcaneus-1 results.csv		AFF853A40AP2

Attachments

- [cookbookExamples.properties.txt](#)
- [PostFileToProcess.groovy.txt](#)

Work with Controls

When working with controls, you can automate their removal from a workflow. For more information, refer to [Automated Removal of Controls from a Workflow](#).

Automated Removal of Controls from a Workflow

At times, control samples in the lab need only be used in a portion of a workflow. For example, E. coli genomic DNA is often prepared alongside samples in Library Preparation protocols and then validated during a Library Validation QC protocol to confirm that nothing went wrong during Library Preparation.

Because the utility of such a control sample is short-lived, there is no need to spend sequencing effort on it. In this scenario, it is advantageous to prevent control samples from advancing in the workflow automatically.

This can be accomplished through the API by implementing an EPP/automation script that removes the control at the end of a step. This example shows how to automate the removal of control samples at the end of a step and remove them from workflows. You can use this method on any step configured to generate individual derived sample (analyte) or `ResultFile` (measurement) outputs. The step can have any number of per-all-input result file (shared file) outputs, as they will be ignored by the script.

i | In the API, an artifact is an item generated by an earlier step. There are two types of artifacts: analyte (derived sample) and `resultfile` (measurement). In the Clarity LIMS web interface, the terms artifact, analyte, and `resultfile` have been replaced with derived sample or measurement. However, some areas of the Operations Interface (in Clarity LIMS v4.2 and earlier) still refer to the former terms.

Prerequisites

Before you follow the example, make sure that you have done the following actions:

- You have `GLSRestApiUtils.groovy` in your Groovy lib folder.
- You have downloaded the `ControlSampleRemoval.groovy` file and have placed it in `/opt/gls/clarity/customextensions/`.
- You have completed the required configuration steps.

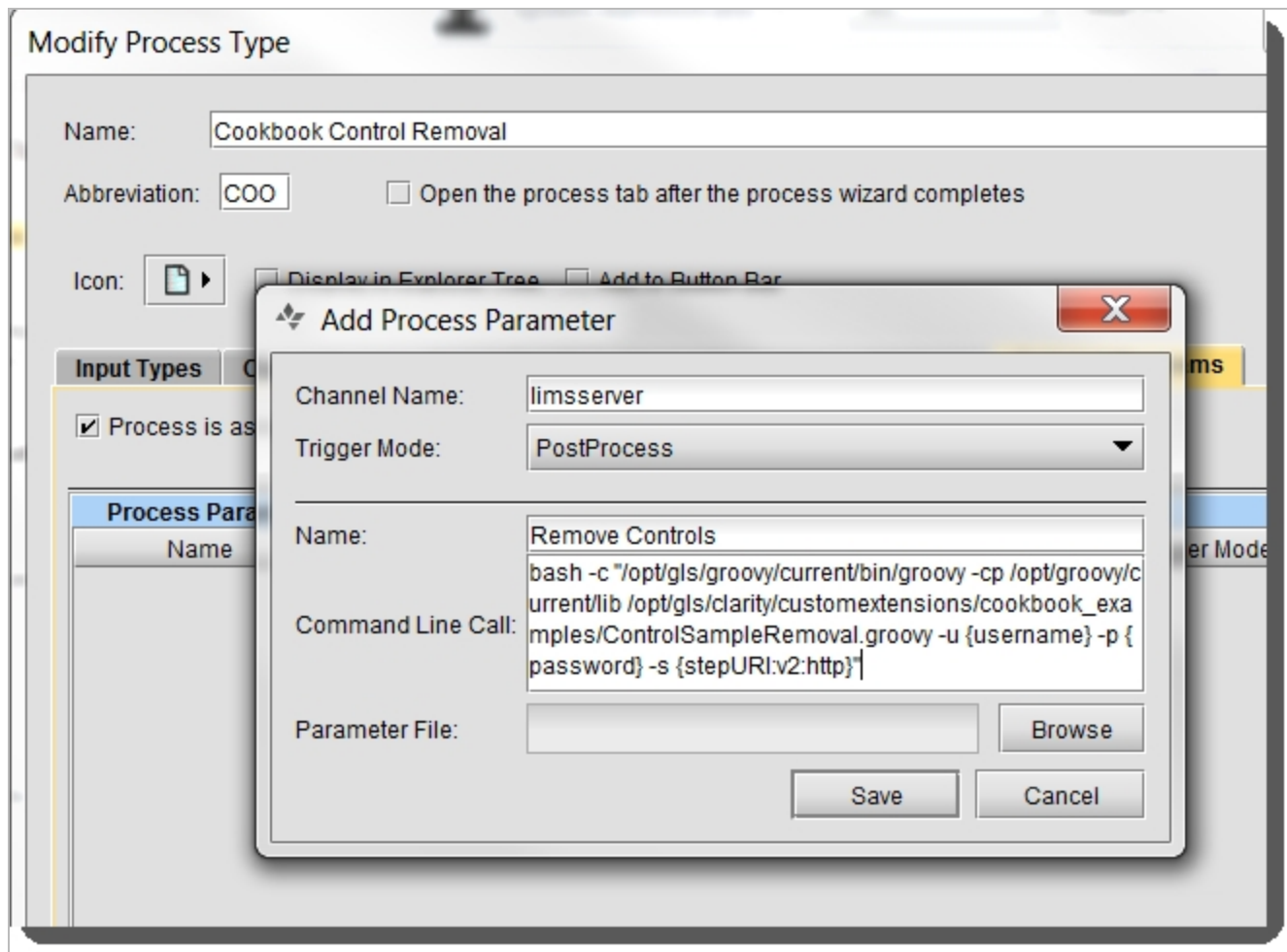
Configuration

Before you can use this example script, you will must complete the following configuration steps.

Configure (Clarity LIMS v4.2 and Earlier)

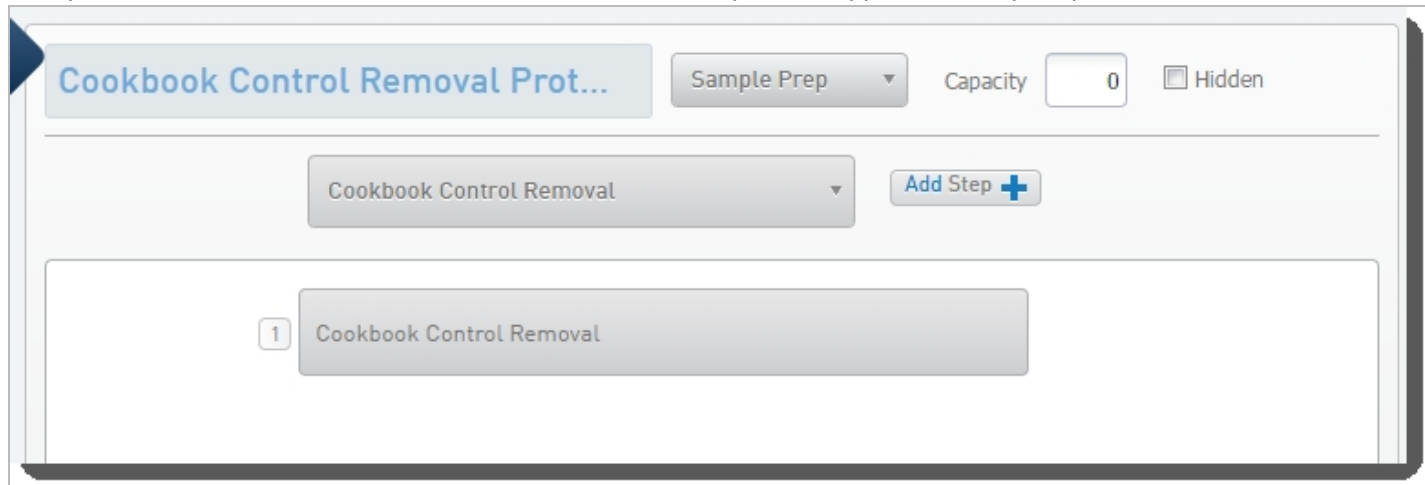
1. In the Clarity LIMS Operations Interface, create a process type named Cookbook Control Removal. The process can have any combination of inputs, and any number of outputs.
2. On the new process type, on the External Programs tab, configure a process parameter named Remove Controls.
3. In the Command Line Call field, enter the following example code.
Modifying the file paths for the Groovy installation on your server.

```
bash -c "/opt/gls/groovy/current/bin/groovy -cp /opt/groovy/current/lib  
/opt/gls/clarity/customextensions/cookbook_examples/ControlSampleRemoval.groovy -u  
{username} -p {password} -s {stepURI:v2:http}"
```



4. In the Clarity LIMS Web Interface, on the Configuration screen, create a protocol named Cookbook Control Removal Protocol.

This protocol should have the Cookbook Control Removal process type as its only step.



- On the Reagents and Controls screen, enable one or more control samples on the Cookbook Control Removal step.
- Configure the Cookbook Control Removal step. On the Automation tab, configure the trigger timing such that the script is initiated automatically at the end of the step.

Name	Behaviour	Stage of Step	Timing
Remove Controls	Automatically initi...	at end of step	

- From Configuration, create a workflow named Cookbook Control Removal Workflow. This workflow should contain the Cookbook Control Removal Protocol.

- i** | You can add the script to any step in any workflow, and you do not need to create a separate step to run it. For this example, it can also be beneficial to add a second step (any type) after this removal step to make sure that the controls were removed.

Configure (Clarity LIMS v5 and Later)

Complete the following steps in the Clarity LIMS Configuration area.

- On the Lab Work tab, create a master step named Cookbook Control Removal.
The master step can be of any type that generates derived samples or measurements. It can generate any number of derived sample or measurement outputs.
- On the Lab Work tab, create a protocol named Cookbook Control Removal Protocol.

3. Add a new Cookbook Control Removal step to the protocol based on the Cookbook Control Removal master step.

The screenshot displays the 'LAB WORK' tab in a software interface. It features three main panels: 'WORKFLOWS', 'PROTOCOLS', and 'STEPS'. The 'WORKFLOWS' panel lists several workflows, each with a 'Pending' status. The 'PROTOCOLS' panel shows a list of protocols, with 'Cookbook Control Removal Protocol' highlighted in blue. The 'STEPS' panel shows a list of steps, with 'Cookbook Control Removal' selected. Below these panels, a 'STEP' configuration area shows a workflow diagram with steps: 'Queue', 'Ice Bucket', 'Step Setup', and 'Auto-Placement'. The 'STEP SETTINGS' section below the diagram contains a dropdown menu for 'cookbook' with options: 'Cookbook Control Removal', 'Library Pooling (Cookbook)', and 'Library Pooling (MiSeq) 5.0 (Cookbook Example)'. A hand cursor is pointing at the 'Cookbook Control Removal' option. At the bottom, a note states: 'Each step is generated from a master step, and is unique to the protocol in which it is contained. At run time, samples in a step are generated from the master step, including: the Queue, the Ice Bucket, the Step Setup, and the Auto-Placement.'

4. On the Automation tab, configure a step automation and name the automation Remove Controls.
5. In the Command Line field, enter the following code example.

Modifying the file paths for the Groovy installation on your server.

```
bash -c "/opt/gls/groovy/current/bin/groovy -cp /opt/groovy/current/lib  
/opt/gls/clarity/customextensions/cookbook_examples/ControlSampleRemoval.groovy -u  
{username} -p {password} -s {stepURI:v2:http}"
```


AUTOMATION DETAILS

NEW AUTOMATION +
Delete
Cancel
Save

Remove Controls

Channel Name limsserver

Command Line

```
bash -c "/opt/gls/groovy/current/bin/groovy -cp /opt/groovy/current/lib
/opt/gls/clarify/customextensions/cookbook_examples/ControlSampleRemoval.groovy -u {username} -p
{password} -s {stepURI:v2:http}"
```

6. Enable the automation on the Cookbook Control Removal master step.
You can now configure the automation trigger on the step or the master step. If you configure the trigger on the master step, the settings are locked on all steps derived from the master step.

AUTOMATION USE

Enable on these Master Steps

× Cookbook Control Removal

7. On the Lab Work tab, select the master step or step.
8. On the Master Step Settings or Step Settings form, in the Automation section, configure the automation trigger as follows so that the script is initiated automatically at the end of the step.
 - **Trigger Location**—Step
 - **Trigger Style**—Automatic upon exit

AUTOMATION

You must configure step automations at the master step level. However, you can configure the automation triggers on either the master step or on the steps derived from that master step. Triggers contain two elements: **Trigger Location** and **Trigger Style**. [Read More](#)

Automation Name	Trigger Location	Trigger Style
Remove Controls	Step	Automatic upon exit

- From Consumables, select the **Control Samples** tab.
- In the Control Samples tab, enable one or more control samples on the Cookbook Control Removal step.

CONTROL USE

Status of Control Active Archived

Single step only Yes No

Enable on these steps

Cookbook Control Rem... >> Cookbook Control Rem...

Select Protocol... >> Select Step...

- On the Lab Work tab, create a workflow named Cookbook Control Removal Workflow. This workflow should contain the Cookbook Control Removal Protocol. You can add the script to any step in any workflow, and you do not need to create a separate step to run it.

For this example, it can also be beneficial to add a second step (any type) after this removal step to make sure that the controls were removed.

Code Example

The following table defines the three parameters/tokens used by the script. As of Clarity LIMS v5.0, the term command-line parameter has been replaced with token.

Parameter/Token	Definition
-s {stepURI}	This is the protocol step URI in the http://<YourIP>/api/v2/steps/<ProtocolStepLimsid> format.
-u {username}	This is the Clarity LIMS username. It is required.
-p {password}	This is the Clarity LIMS password. It is required.

After the script has processed the parameters / tokens and ensured that all the required information is available, it can begin to process the samples to determine if they should be removed.

Step 1. Retrieve Next Steps and Protocol Information

To begin, retrieve the list of permitted actions for the step from the API. This contains a list of the URIs for the input analytes and their next steps.

You can use this information to get the URI of the current step, which allows you to obtain the information for the step itself.

```
// Retrieve the next steps and protocol step information
def actionsList = GLSRestApiUtils.httpGET(stepURI + '/actions', username, password)
def currentProtocolStepURI = actionsList.'configuration'[0].@uri
def currentProtocolStep = GLSRestApiUtils.httpGET(currentProtocolStepURI, username, password)
```

Step 2. Determine the URIs of Desired Next Steps

Next, look at the possible next steps that can be used in this step. In doing so, you are able to collect the next-step-uri values that are associated with the next step names within the NEXT_STEPS map.

In this case, you are looking for the URI associated with the Remove from workflow option.

```
// Determine the uris of the desired next
steps
currentProtocolStep.'transitions'.each { if(NEXT_STEPS.containsKey(it.@name)) { NEXT_STEPS[it.@name] = it.'next-step-uri' }}
```

Step 3. Find Controls and Change Their Next Steps to Remove

After you have retrieved the URIs for the desired next steps, you can iterate through the actions list checking artifacts to see if they are controls. If the artifact associated with any next action is a control, change the next action of the artifact to be the removal next step URI you retrieved previously.

```
// Feedback variable
int controlSamples = 0
// Collect and batchGet to retrieve a list of artifacts that are at the end of step
def artifactList = GLSRestApiUtils.batchGET(actionsList.'next-actions'.each { it.'next-action' }, username, password)
// Single out the artifacts that belong to controls
def controlList = artifactList.findAll{it.'control-type'=='uri'}
// Convert the artifact list of removals to just the stripped down URIs
def removalList = controlList.collect { it.'uri'.split(/\?/) }
// Go through the original action list (what we need to put) and modify next steps
```

```
if their artifact URI is in the removal listactionsList.'next-actions'.'next-action'.each
{ if(removalList.contains(it.@'artifact-uri')){ controlSamples++ it.@'action' = 'remove'
it.@'step-uri' = NEXT_STEPS[REMOVAL_STEP] }}
```

Step 4. Update the Next Steps and Define Success Message

Finally, send the change for the next step information to the desired endpoint and then define our success message to the user. The message allows you to inform the user of the results of the script.

```
// Update the next steps in the APIGLSRestApiUtils.httpPUT(actionsList, stepURI +
'/actions', username, password)// Define the success message to the useroutputMessage =
"Script has completed successfully.${LINE_TERMINATOR}" + "${controlSamples} controls were
removed and ${samples} were not removed from the workflow(s)."
```

Expected Outcome and Results

Assuming samples and a control sample have been placed in the Cookbook Control Removal Workflow Ice Bucket, you can proceed as normal through the step.

- On the Assign Next Steps screen, provide various Next Step values, if desired.
- Proceed with the completion of the step. A message box displays, alerting you to the execution of a custom script.
- When the script completes, a success message displays and the controls have been removed from the workflow.

Attachments

[ControlSampleRemoval.groovy.txt](#)

About the Application Examples

The BaseSpace Clarity LIMS Application Examples use example scripts to help you learn how to work with REST and EPP (automation) scripts. Application examples are larger scripts that teach you how to do something useful. The purpose of these examples is to get you scripting quickly.

Be aware that the application examples differ from purchased Clarity LIMS integrations:

- Purchased LIMS integrations cannot be modified. Because Illumina guarantees specific functionality when using integrations, any code or script modifications must only be performed by the Illumina Integrations or Support teams.
- Application examples are learning tools that are intended to be modified. The application examples follow best practices in terms of over-all script construction; however, their exact functions are less critical than the integrations.



The API documentation includes the terms External Program Integration Plug-in (EPP) and EPP node. As of BaseSpace Clarity LIMS v5.0, these terms are deprecated. The term EPP has been replaced with automation. EPP node is referred to as the Automation Worker or Automation Worker node. These components are used to trigger and run scripts, typically after lab activities are recorded in the LIMS.

The best way to get started is to download the example script and try it out. Once you have seen how the script works, you can dissect it and use the pieces to create your own script.

Python API Library (glsapiutil.py) Location

The latest glsapiutil and glsapiutil3 Python libraries can be found on the [GitHub](#) page. Download links to these files will be removed from example documentation over time.

Scripts That Help Automate Steps

[Route Artifacts Based Off a Template File](#)

[Invoking bcl2fastq from BCL Conversion and Demultiplexing Step](#)

[Email Notifications](#)

[Finishing the Current Step and Starting the Next](#)

[Adding Downstream Samples to Additional Workflows](#)

[Advancing/Completing a Protocol Step via the API](#)

[Setting a Default Next Action](#)

[Automatic Placement of Samples Based on Input Plate Map \(Multiple Plates\)](#)

[Automatic Placement of Samples Based on Input Plate Map](#)

[Publishing Files to LabLink](#)

[Automatic Pooling Based on a Sample UDF/Custom Field](#)

[Completing a Step Programmatically](#)

[Automatic Sample Placement into Existing Containers](#)

[Creating Multiple Containers / Types for Placement](#)

[Routing Output Artifacts to Specific Workflows/Stages](#)

[Starting a Protocol Step via the API](#)

[Setting Quality Control Flags](#)

[Applying Indexing Patterns to Containers Automatically](#)

[Assignment of Sample Next Steps Based On a UDF](#)

[Parsing Metadata into UDFs \(BCL Conversion and Demultiplexing\)](#)

Route Artifacts Based Off a Template File

Workflows do not always have a linear configuration. There are situations when samples progressing through a workflow must branch off and be directed to different stages, or even different workflows. You can add samples to a queue using the `/api/{version}/route/artifacts` endpoint.

In the lab, decisions are made dynamically. At the initiation of this example workflow, it is not known whether the sample is destined to be sequenced on a HiSeq or MiSeq instrument. As a result, the derived samples must be routed to a different workflow stage midworkflow.

This programmatic approach to queuing samples can be used many times throughout a workflow. This approach eliminates the need to write multiple complex scripts - each of which must be maintained over time.

This example describes a Python script that takes instruction from a .csv template file. Removing any hard-coded reference to specific UDFs / custom fields or workflow stages from the script allows for easy configuration and support. All business logic can be implemented solely through generation of a template and EPP / automation command.

The template contains UDF / custom field names and values. If samples in the active step have fields with matching values, they are queued for the specified workflow stage.

Step Configuration

The step is configured to display the two checkbox analyte UDFs / derived sample custom fields. These fields are used to select the destination workflow stages for each derived sample / analyte. You can queue the sample for HiSeq, MiSeq, or both.

Sample Details			
Sample Name	* QC	Go to HiSeq	Go to MiSeq
96 well plate: Purified plate (8 samples)			
RNA Sample 289		☒	☐
RNA Sample 290		☒	☐
RNA Sample 291		☐	☒
RNA Sample 292		☐	☒
RNA Sample 293		☐	☐
RNA Sample 294		☐	☐
RNA Sample 295		☒	☒
RNA Sample 296		☒	☒

In the preceding example of the Sample Details screen, the user selected:

- Two samples to be queued for HiSeq
- Two samples for MiSeq
- Two samples where routing is not selected
- Two samples for both HiSeq and MiSeq

Step Details

Route by template successful! June 7, 2017 - 1:30 pm
22 samples were added to the http://localhost:9080/api/v2/configuration/workflows/1252/stages/2507 step.

Route all samples to:

Step A
Step A
Step B

Route by template Run

Files

Routing Log
92-76023

Sample Details
Route A
Apply

Sample Name	Container Name	Route A	Go to MiSeq	Go to HiSeq	Well	Comment	LIMS ID (Container)
96 well plate: 27-12201 (22 samples)							
3RNA	27-12201				A:1		27-12201
4RNA	27-12201				A:2		27-12201

In the preceding example of the Step Details screen, the user selected:

- A step level UDF / custom field to assign all samples to a given step. All output samples are routed to the associated step. Samples can be duplicated in the first step of the following protocol if:
- Samples are routed as the last step of a protocol and;
- The action of the next step is **Mark Protocol as Complete**.

The duplication is due to the Next Steps action queuing the artifact for its default destination in addition to the script routing the same artifact.

The solution here is to set the default **Next Steps** action to **Remove from Workflow** instead. This action can be automated using the Lab Logic Toolkit or the API.

LIMS v4.x

Make sure that the **Next Steps are set by a script** option is set to **Yes**.

Protocols

Workflows

Automation

SamplesOut » Sandbox Sample Output

« Back Properties Automation **Next Steps**

Next Step Options

Next Steps are set by a script **Yes** No

Automatically start Next Step Yes **No**

LIMS v5.x

On the protocol configuration screen, make sure that Start Next Step is set to **Automatic**.

Start Next Step	MANUAL	AUTOMATIC	MANUAL
-----------------	--------	-----------	--------

Template Configuration

Each UDF name and value pair correspond to a workflow and stage combination as the destination to route the artifact.

- The UDF name in the template could be configured as a step or analyte level UDF. If a step level UDF has a value that is specified in the template, all analytes in the step are routed.
- The UDFs can be of any type (Numeric, Text, Checkbox). If a checkbox UDF is used, the available values are true or false.
- UDF values in the template are not case-sensitive.

The template requires four columns:

- UDF_NAME
- UDF_VALUE
- WORKFLOW_NAME

- STAGE_NAME

For this example, the template values would be:

- UDF_NAME, UDF_VALUE, WF_NAME, STAGE_NAME
- Route all samples to: Step A, Workflow A, Stage A
- Route all samples to: Step B, Workflow B, Stage B
- Route A, True, Workflow A, Stage A
- Go to HiSeq, True, TruSeq Nano DNA for HiSeq 5.0, Library Normalization (Illumina SBS) 5.0
- Go to MiSeq, True, TruSeq DNA PCR-Free for MiSeq 5.0, Sort MiSeq Samples (MiSeq) 5.0

This script might be used numerous times in different EPP / automation scripts, with each referencing a different template.

Using a Template String Instead of a Template File

Due to restrictions on file server access, this script accepts routing template instructions using a string EPP parameter with lines separated by a newline character ('\n'). The following example shows how this parameter string would be added to represent the previous template:

```
-r "Route all samples to:, Step A, Workflow A, Stage A\nRoute all samples to:, Step B, Workflow B, Stage B\nRoute A, True, Workflow A, Stage A\nGo to HiSeq, True, TruSeq Nano DNA for HiSeq 5.0, Library Normalization (Illumina SBS) 5.0\nGo to MiSeq, True, TruSeq DNA PCR-Free for MiSeq 5.0, Sort MiSeq Samples (MiSeq) 5.0"
```

Required Parameters

The EPP/automation that calls the script must contain the following parameters:

-p	--password	The password of the current user (Required)
-s	--stepURI	The URI of the step that launches the script, the {stepURI:v2} token (Required)
-l	--log	The path, or limsid of the log file, the {compoundOutputFileLuidN} token (Required)
-t	--template	The path to the template file (Required)*
-r	--template_string	A string containing the template information (Required)*

Either **--template or **--template_string** is required. If both are provided, **--template_string** is used.*

Optional Parameter

--input Uses the input artifacts instead of output artifacts. Default = False

When the `--input` parameter is used, the script routes input artifacts instead of the default output artifacts. UDF values of the input artifacts (instead of outputs) are against the template file.

An example of the full syntax to invoke the script is as follows:

```
bash -c "/usr/bin/python /opt/gls/clarity/customextensions/Route_by_template.py -u {username} -p {password} -s {stepURI:v2} -t /opt/gls/clarity/customextensions/routing_template.csv -l {compoundOutputFileLuid0}"
```

Or, if you wish to route the inputs instead of outputs:

```
bash -c "/usr/bin/python /opt/gls/clarity/customextensions/Route_by_template.py --input -u {username} -p {password} -s {stepURI:v2} -t /opt/gls/clarity/customextensions/routing_template.csv -l {compoundOutputFileLuid0}"
```

User Interaction

When the **Record Details** screen is entered, the UDF / custom field checkboxes or drop-down options specify to which workflow/stage combination each derived sample is sent.

About the Code

The first important piece of information required is the URI of the destination stage. There is a unique stage URI assigned to each workflow and stage combination.

A stage URI can change across LIMS instances (such as switching from Dev to Prod). Therefore, the script gathers the stage URI from the **workflow and stage name**. This process occurs even when the workflows are identical.

The main method in the script is **routeAnalytes()** and it carries out several operations:

1. Gathers the information for the process that triggered the script, including output (or input) analytes.
2. For each analyte, evaluates which UDFs have been set, and adds the analyte to a list of analytes to route.
3. Creates the XML message for each stage.
4. Does a POST to the REST API to add the analytes to the queue in Clarity LIMS.

This example, while in itself useful, also serves to demonstrate a more general concept. Routing artifacts is valuable in any situation where a sample must be queued for a stage outside of the usual order of a workflow. This routing is applicable even when routing newly submitted samples to the first stage in a workflow.

For more information, see the artifact/route REST API documentation.

Assumptions and Notes

- You are running a version of Python supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The attached files are placed on the LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The Python API Library (glsapiutil.py) is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from the [GitHub](#) page.

- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[routing_template.csv](#)

[route_by_template.py.txt](#)

Invoking bcl2fastq from BCL Conversion and Demultiplexing Step

Illumina sequencing protocols include a **BCL Conversion and Demultiplexing** step. This stage allows you to select the command options for running bcl2fastq2. The bcl2fastq must be initiated through a command-line call on the BCL server. This example allows you to initiate the bcl2fastq2 conversion software by the selection of a button in BaseSpace Clarity LIMS.

Step Configuration

The "out of the box" step is configured to include the following UDFs / custom fields. You can select these options on the **Record Details** screen. You can also configure additional custom options.

About the Code

The main method in the script is convertData(). This method performs several operations:

1. The script determines the run folder. The link to the run folder is attached as a result file to the sequencing step.
 - The script searches for the appropriate sequencing step and downloads the result file containing the link.
 - The script changes directories into the run folder.
2. The script gathers all the step level UDFs / custom fields from the **BCL Conversion and Demultiplexing** step.
3. Using the information gathered, the script builds the command that is executed on the BCL server. The command consists of two parts: cd (changing directory) into the run folder.
 - Executing the bcl2fastq command with the selected options.

Script Configuration

This script must be copied to the BCL server because the script is executed on the BCL server remote Automated Informatics (AI) / Automation Worker (AW) node.

By default, the remote AI / AW node does not come with a custom extensions folder. Therefore, if this script is the first script on the server you can create a **customextensions** folder in **/opt/gls/**.

 It is not recommended to have the **customextensions** folder in the **remoteai** folder as the **remoteai** folder can get overwritten.

When uploading the script, ensure the following:

- The path to the bcl2fastq application is correct (line 17)

- The sequencing process type matches exactly the name of the process type / master step the artifact went through (the **-d** parameter)
- The **customextensions** folder contains both **glsapiutil.py** and **glsfileutil.py** modules. See [Assumptions and Notes](#).

Parameters

The script accepts the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script - the {stepURI:v2} token (Required)
-d	The display name of the sequencing step (Required)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/customextensions/ kickoff_bcl2fastq2.py -u {username} -p {password}
-s {stepURI:v2} -d 'Illumina Sequencing (Illumina SBS) 5.0'
```

Assumptions and Notes

- You are running a version of Python supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The attached files are placed on the BCL server, in the **/opt/gls/clarity/customextensions** folder.
- The Python API Library (glsapiutil.py) is placed on the BCL server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from the [GitHub](#) page.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[kickoff_bcl2fastq2.py.txt](#)

[glsfileutil.py.txt](#)

Email Notifications

Stakeholders are interested in the progress of samples as they move through a workflow. Email alerts of events can provide them with real-time notifications.

Some possible uses of notifications include the following:

- Completion of a workflow for the billing department
- Manager review requests
- Notice of new files added via the LabLink interface
- Updates on samples that are not following a standard path through a workflow

BaseSpace Clarity LIMS provides a simple way of accomplishing this using a combination of the Clarity LIMS API, EPP / automation triggers, and Simple Mail Transfer Protocol (SMTP).

Solution

The **send_email()** method uses the Python **smtplib** module to create a Simple Mail Transfer Protocol (SMTP) object to build and send the email. The attached script does the following:

- 1. Gathers relevant data from the Clarity LIMS API endpoints.
- 2. Generates an email body according to a template.
- 3. Calls the send_email() function.

Mail Server Configuration

Connect to Clarity LIMS SMTP with:

```
host='localhost', port=25
```

Because of server restrictions, the script can send emails from only:

```
noreply.clarity@illumina.com
```

Automation Parameters

The automation / EPP command is configured to pass the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script (Required)

Example command line:

```
python /opt/gls/clarity/customextensions/emails_from_Clarity.py -u {username} -p {password} -s {stepURI}
```

User Interaction

- The script can be executed using a Clarity LIMS automation / EPP command, and triggered by one of the following methods:
 - Manually, via a button on the **Record Details** screen.
 - Automatically, at a step milestone (on entry to or exit from a screen).
- The script can also be triggered outside of a Clarity LIMS workflow, using a time-based job scheduler such as cron.

Assumptions and Notes

- You are running a version of Python supported by Clarity LIMS, as documented in the [Technical Requirements](#).

- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[emails_from_clarity.py.txt](#)

Finishing the Current Step and Starting the Next

Background

Many facilities have linear, high-throughput workflows. Finishing the current step and finding the samples that have been queued for the next step. Then putting the samples into the virtual ice bucket. Then actually starting the next step. All of these steps can be seen as unnecessary overhead.

This solution provides a methodology that allows a script to finish the current step, and start the next.

We have already illustrated how a step can be run in its entirety via the API. Finishing one step and starting the next would seem a straightforward exercise, but, as most groups have discovered, there is a catch.

Consider the step to be autocompleted as Step A, with Step B being the next step to be autostarted.

- The script to complete Step A and start Step B is triggered from Step A. In the solution defined below, the script is triggered manually by clicking a button on the **Record Details** screen.
- As Step A invokes a script:

```
The step itself cannot be completed - because the script has not completed successfully.  
and
```

```
The script cannot successfully complete unless the step has been completed.
```

To break this circle, the script initiated from Step A must invoke a **second script**. The first script completes successfully, and the second script is responsible for finishing Step A and starting Step B.

Solution

Invoking Script: *runLater.py*

The **runLater.py** script launches the subsequent script (**finishStep.py**) by handing it off to the Linux 'at' command (refer to [Notes](#)). This process effectively 'disconnects' the execution of the **finishStep.py** script from **runLater.py**, allowing **runLater.py** to complete successfully, and then Step A to be completed.

Parameters

The script accepts the following parameter:

```
-c The command to be passed to the at command (Required)
```

Although this script accepts just a single parameter, the value of this parameter is complex. This script is the full command line and the parameters that must be passed to the **finishStep.py** script.

Following is an example of a complete EPP / automation command-line string that invokes **runLater.py**, and in turn **finishStep.py**:

```
/usr/bin/python /opt/gls/clarity/customextensions/runLater.py -c "/usr/bin/python  
/opt/gls/clarity/customextensions/finishStep.py -u {username} -p {password} -s  
{stepURI:v2:http} -a START"
```

Invoking Script: *finishStep.py*

The **finishStep.py** script completes and starts Steps A and B respectively.

Parameters

The script accepts the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script (Required)
-a	The action, or mode that the script runs under (Required). Accepted values are START (which completes the current step and starts the next) or COMPLETE (completes the current step)

About the Code

The first thing that this script does is go to sleep for five seconds before the main method is called. This delay allows the application to detect that the invoking script has completed successfully, and allow this script to work. The duration of this delay can be adjusted to allow for your server load, etc.

The central method (**finishStep()**) assumes that the current Step (Step A) is on the **Record Details screen**. Next, the method calls **advanceStep()** to move the step onto the screen to allow the next step to be selected. By default, the **routeAnalytes()** method selects the first available next step in the protocol / workflow for each analyte. Then the **advanceStep()** method is called, which completes Step A.

If the script has been passed a value of 'START' for the -a parameter, the next step (Step B) will now be started. The **startNextStep()** method handles this process, carrying out the following high-level actions:

- It determines the default output container type for Step B.
- It invokes an instance of Step B on the analytes routed to it via the **routeAnalytes()** method.
- If the Step B instance was invoked successfully, it gathers the LUID of the step.

At this point, it is possible that the **apiuser** user has started the step. This situation is not ideal. We would like to update it so that it shows up in the **Work in Progress section** of the GUI for the user that initiated Step A.

This script makes a note of the current user and then updates the new instance of Step B with the current user. This update can fail if Step B has mandatory fields that must be populated on the **Record Details** screen (refer to [Notes](#)). It can also fail if the step has mandatory reagent kits that must be populated on the **Record Details** screen.

If the update does fail, the step has been started, but the **apiuser** user still owns it. It can be continued as normal.

Assumptions

- The attached files are placed on the BaseSpace Clarity LIMS server, in the following location:
opt/gls/clarity/customextensions
- The attached files are readable by the **glsai** user.
- Updating the HOSTNAME global variable such that it points to your Clarity LIMS server is required.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.
- The scripts use rudimentary logging. After the scripts are installed and validated, these logs are of limited value, and their creation can likely be removed from the scripts.

Notes

- The Linux 'at' command might not be installed on your Missing variable reference server. For installation instructions, refer to the Linux documentation.
- For a solution to this issue, see [Validating Process / Step level UDFs](#).

Attachments

[runLater.py.txt](#)

[clarityHelpers.py.txt](#)

[finishStep.py.txt](#)

Adding Downstream Samples to Additional Workflows

When processing samples, there can be circumstances in which you must add downstream samples to additional workflows. This sample addition is not easy to achieve using the Server Name interfaces, but is easy to do via the API.

Solution

This example provides a Python script that can be used to add samples to an additional workflow step. The example also includes information on the key API interactions involved.

- It is the outputs, not the inputs, of the process that are added to the workflow step. (If you would like to add the inputs, changing this step is simple.)
- This example is an **add** function, not a **move**. If you would like to remove the samples from the current workflow, you can do arrange to do so by building an **<unassign>** element.
- The process is configured to produce analyte outputs, and not result files.

Parameters

The script accepts the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-l	The limsid of the process invoking the script (Required)
-s	The URI of the step that launches the script (Required)
-w	The name of the destination workflow (Required)
-g	The name of the desired stage within the workflow (Required)

About the Code

The step URI (-s) parameter is used to report a meaningful message and status back to the user. These reports depend upon the outcome of the script.

Initially, the parameters are gathered and the helper object defined in the **glsapiutil.py** has been instantiated and initialized. Its methods can be called to take care of the RESTful GET/PUT/POST functionality. The script calls the following functions:

- **getStageURI()**
- **routeAnalytes()**.

getStageURI() function

The **getStageURI()** function converts the **workflowname** and **stagename** parameters into a URI that is used as the assign element, for example:

```
<assign stage-uri="http://localhost:8090/api/v2/configuration/workflows/7/stages/4">
```

routeAnalytes() function

The **routeAnalytes()** function gathers the outputs of the process, and harvests their URIs to populate in the **<artifact>** elements. This function also uses the reporting mechanism of the step URI parameter.

API resource

The crucial resource in this example is the **route/artifacts** resource. This API endpoint can only be POSTed to, and accepts XML of the following form:

```
<rt:routing xmlns:rt="http://genologics.com/ri/routing"> <assign stage-  
uri="http://localhost:8090/api/v2/configuration/workflows/7/stages/4"> <artifact  
uri="http://localhost:8090/api/v2/artifacts/5"/> <artifact  
uri="http://localhost:8090/api/v2/artifacts/6"/> </assign> <assign workflow-  
uri="http://localhost:8090/api/v2/configuration/workflows/7"> <artifact  
uri="http://localhost:8090/api/v2/artifacts/8"/> <artifact  
uri="http://localhost:8090/api/v2/artifacts/9"/> </assign> <unassign workflow-  
uri="http://localhost:8090/api/v2/configuration/workflows/10"> <artifact  
uri="http://localhost:8090/api/v2/artifacts/11"/> <artifact  
uri="http://localhost:8090/api/v2/artifacts/12"/> </unassign></rt:routing>
```

For more information, refer to the [route/artifacts REST API documentation](#). Also useful are the [configuration/workflow resources](#), both single and list.

Assumptions and Notes

- The attached file is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The Python API Library (glsapiutil.py) is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from our [GitHub](#) page.
- The HOSTNAME global variable must be updated such that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[addToStep.py.txt](#)

Advancing/Completing a Protocol Step via the API

This topic forms a natural partner to the [Starting a Protocol Step via the API](#) application example. When protocol steps are being initiated programmatically, we must know how to advance the step through the various states to completion.

Solution

Advancing a step is quite a simple task. It requires the use of the **steps/advance** API endpoint - in fact, little else is needed.

Let us consider a partially completed step with ID 24-1234. To advance the step to the next state, the following is required:

1. Perform a GET to the resource **.../api/v2/steps/24-1234** , saving the XML response.
2. POST the XML from step 1 to **.../api/v2/steps/24-1234/advance**, and monitor the returned XML for success.
3. If successful, the protocol step advances to its next state, as if the lab scientist had advanced it via the Clarity LIMS interface.
4. Advancing a protocol step that is in its **final state** completes the step.

The Python **advanceStep (STEP_URI)** method shown in this document advances a step through its various states. To achieve this process, the URI of the step is passed to the method to be advanced/completed.

```
def advanceStep( STEP_URI ): response = False stXML = API.getResourceByURI( STEP_URI ) rXML = API.createObject( stXML, STEP_URI + "/advance" ) rDOM = parseString( rXML ) nodes = rDOM.getElementsByTagName( "configuration" ) if len( nodes ) > 0: response = True return response
```

Assumptions and Notes

- The **glsapiutil.py** file is placed on the BaseSpace Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. The latest glsapiutil (and glsapiutil3) Python libraries are on the [GitHub](#) page.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Setting a Default Next Action

Often, a protocol step will have just a single 'next action' that is required to continue the workflow. In such cases, it can be desirable to automatically set the default action of the next step.

This example script shows how this task can be achieved programmatically.

Solution

Step Configuration

The step is configured to invoke a script that sets the default next action when you exit the **Record Details** screen of the step.

Parameters

The automation command is configured to pass the following parameters to the script:

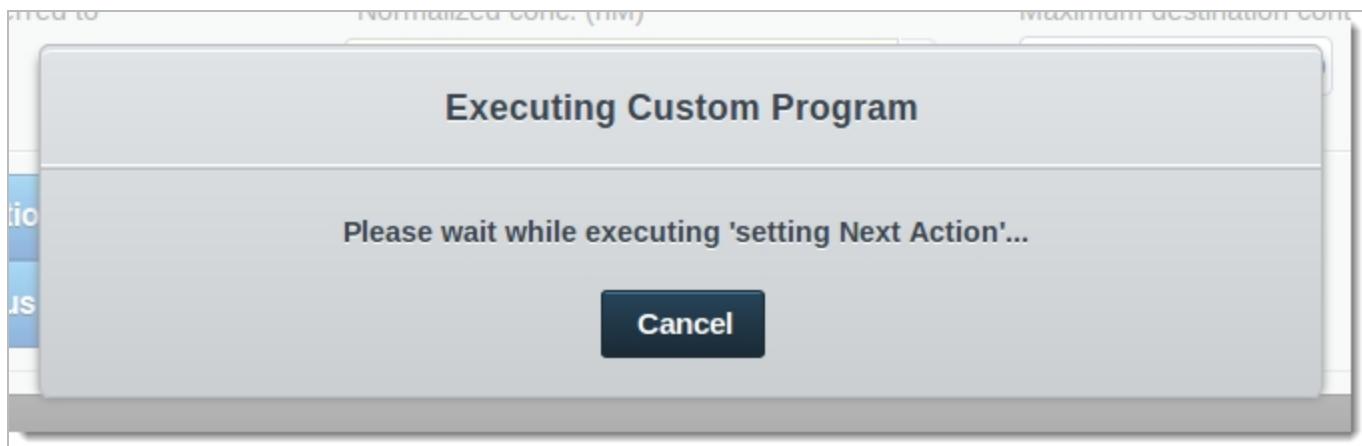
-u	The username of the current user (Required)	
-p	The password of the current user (Required)	
-s	The URI of the step that launches the script (Required)	The {stepURI} token
-a	Sets the next action to a fixed value. (Optional)	This parameter is for advanced use, when you want to set the next action to a fixed value. Possible steps can be 'repeat step', 'remove from workflow', and so on.

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/setDefaultNextAction.py -u admin -p securepassword -s https://demo-5-1.claritylims.com/api/v2/steps/122-7953
```

User Interaction

When the lab scientist exits the **Record Details** screen, the script is invoked and the following message displays:



If the script completes successfully, the LIMS displays the default next action. If the current step is the **final step** of the protocol, it is instead marked as complete:

Samples		*required	Select this sample's next step	Apply		
Sample Name	*Next Step	Normalized conc. (nM)	Container Name	Well	LIMS ID (Container)	
▼ 96 well plate: 27-5152 (2 samples)						
DNA Extraction Control-1	✔ Mark protocol as complete		27-5152	A:1	27-5152	
DNA Extraction Control-2	✔ Mark protocol as complete		27-5152	A:2	27-5152	

Manual Intervention

At this point, the lab scientist is able to intervene manually and reroute failed samples accordingly.

i | It is not possible to prevent the user from selecting a particular next-step action. However, it is possible to add a validation script that checks the next actions selected by the user against a list of valid choices.

If the selected next step is not a valid choice, you can configure the script such that it takes one of the following actions:

- Replaces the next steps with steps that suit your business rules.
- Issues a warning, and prevents the step from completing until the user has changed the next step according to your business rules.

About the Code

The main method in the script is **routeAnalytes()**. The method in turn carries out several operations:

1. The **actions** resource of the current protocol step is investigated.
2. The possible next steps are identified.

3. If there is no next step, the current step is set to 'Mark protocol as complete.'
4. If there are multiple next steps, the first is used to set the next action.

Assumptions and Notes

- The attached file is placed on the LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The Python API Library (glsapiutil.py) is placed on the BaseSpace Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from the [Illumina GitHub resources](#).
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[setDefaultNextAction.py.txt](#)

Automatic Placement of Samples Based on Input Plate Map (Multiple Plates)

The BaseSpace Clarity LIMS interface offers tremendous flexibility when placing the outputs of a step into new containers. Consider if your protocol step **always** involves the placement of samples using the plate map of the input plate. If yes, it makes sense to automate sample placement.

This example provides a script that allows sample 'autoplacement' to occur, and describes how the script can be triggered.

This script is similar in concept to the [Automatic Placement of Samples Based on Input Plate Map](#) example. The difference being that this updated example is able to work with multiple input plates, which can be of different types.

Solution

In this example, samples are placed according to the following logic:

- The process type / master step is configured to produce just one output analyte (derived sample) for every input analyte.
- The output analytes are placed on the same type of container as the corresponding source plate.
- Each occupied well on the source plate populates the corresponding well on the destination plate.
- The destination plate is named such that it has the text ' *DEST-* ' prepended to the name of its corresponding source plate.

Step Configuration

In this example, the step is configured to invoke the script on entry to the Sample Placement screen.

Parameters

The EPP / automation command is configured to pass the following parameters:

-l	The limsid of the process invoking the script (Required)
-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the protocol step that launches the script- the {stepURI:v2:http} token (Required)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/autoplaceSamplesDefaultMulti.py -l 122-7953 -u admin -p securepassword -s http://192.168.9.123:8080/api/v2/steps/122-5601
```

About the Code

The main method in the script is **autoPlace()**. This method executes several operations:

1. Calls to **createContainer()** prompt the creation of the destination plates.
2. It harvests enough information so that the objects required by the subsequent code can retrieve the required objects using the batch API operations. Doing so involves using some additional code to build and manage the cache of objects retrieved in the batch operations:
 - `cacheArtifact()`
 - `prepareCache()`
 - `getArtifact()`
3. The cached analytes are then accessed. After the source well to which the analyte maps has been determined, the output placement can be set. This information is presented in XML that can be POSTed back to the server in the format required for the placements resource.
4. After all the analytes have been processed, the placements XML is further supplemented with required information, and POSTed to the `../steps/<stepID>/placements` API resource.
5. Finally, a meaningful message is reported back to the user via the `../steps/<stepID>/programstatus` API resource.

Assumptions and Notes

- The attached file is placed on the Clarity LIMS server, in the `/opt/gls/clarity/customextensions` folder.
- The Python API Library (`glsapiutil.py`) is placed on the Clarity LIMS server, in the `/opt/gls/clarity/customextensions` folder. You can download the latest `glsapiutil` library from our [GitHub page](#).
- The `HOSTNAME` global variable must be updated so that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[autoplaceSamplesDefaultMulti.py.txt](#)

Automatic Placement of Samples Based on Input Plate Map

The BaseSpace Clarity LIMS interface offers tremendous flexibility when placing the outputs of a protocol step into new containers. Consider if your protocol step **always** involves the placement of samples using the plate map of the input plate. If yes, it makes sense to automate sample placement.

This example provides a script that allows sample 'autoplacement' to occur. It also describes how the script can be triggered, and what the lab scientist sees when the script is running.

i | For an example script that automates sample placement using multiple plates, see the [Automatic Placement of Samples Based on Input Plate Map \(Multiple Plates\)](#) example.

Solution

In this example, samples are placed according to the following logic:

- The step produces one output sample for every input sample.
- The output samples are placed on a 96 well plate.
- Each occupied well on the source 96 well place populates the corresponding well on the destination 96 well plate.

Step Configuration

In this example, the step is configured to invoke the script on entry to the Sample Placement screen.

Parameters

The EPP / automation command is configured to pass the following parameters:

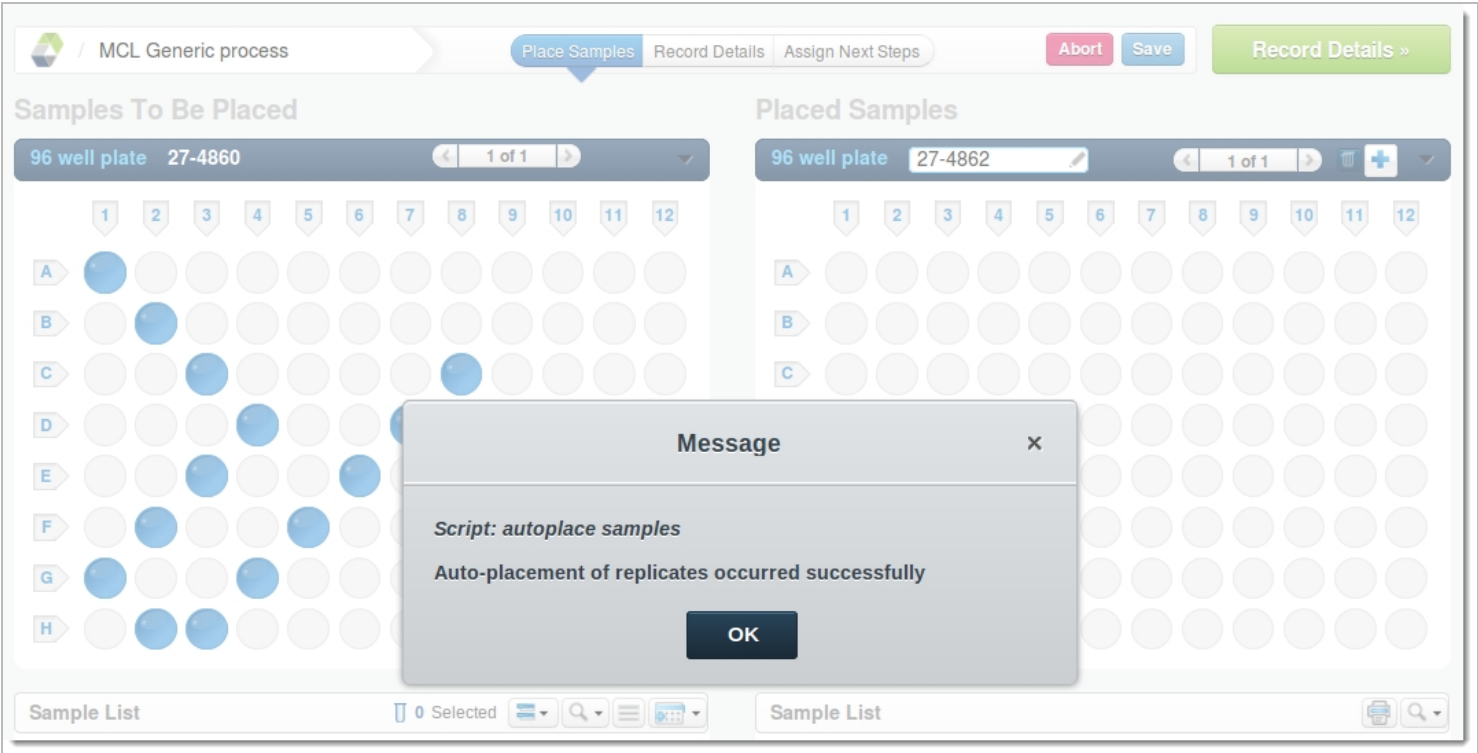
-l	The luid of the process invoking the script (Required)
-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script - the {stepURI:v2:http} token (Required)

An example of the full syntax to invoke the script is as follows:

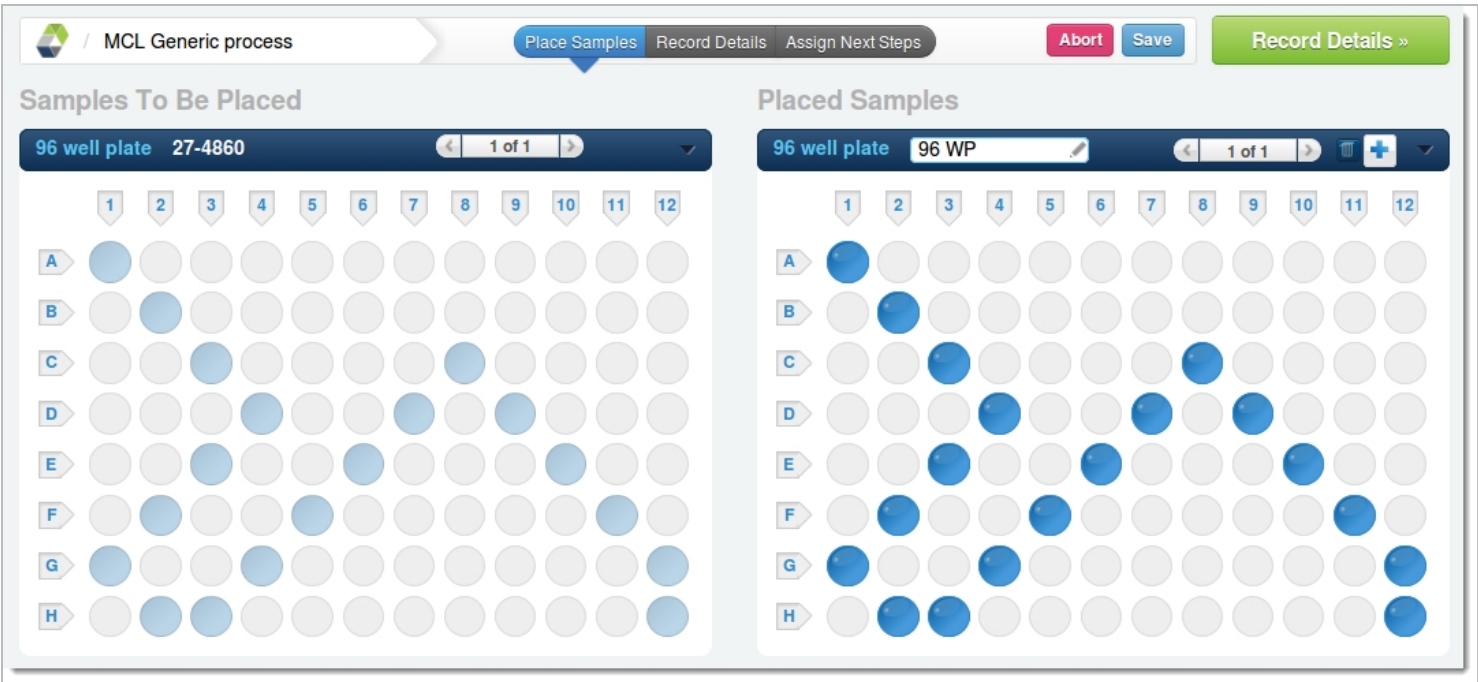
```
/usr/bin/python /opt/gls/clarity/customextensions/autoplaceSamplesDefault.py -l 122-7953 -u admin -p securepassword -s http://192.168.9.123:8080/api/v2/steps/122-5601
```

User Interaction

When the lab scientist enters the Sample Placement screen, a message similar to the following appears:



When the script has completed, the rightmost **Placed Samples** area of the Placement screen displays the container of autoplace samples:



At this point, the lab scientist can review the constituents of the container, and complete the step as normal.

About the Code

The main method in the script is **autoPlace()**. This method in turn carries out several operations:

1. A call to **createContainer()** prompts the creation of the destination 96 well plate.
2. The method harvests enough information so that the objects required by the subsequent code can retrieve the required objects using the 'batch' API operations. You add additional code to build and manage the cache of objects retrieved in the batch operations:
 - `cacheArtifact()`
 - `prepareCache()`
 - `getArtifact()`
3. The cached analytes are accessed and the source well to which each analyte maps is determined.
4. Output placement can then be set. This information is presented in XML that can be POSTed back to the server in the correct format required for the **placements** resource.
5. After the analytes have been processed, the placements XML is further supplemented with required information, and POSTed to the `../steps/<stepID>/placements` API resource.
6. Finally, a meaningful message is reported back to the user via the `../steps/<stepID>/programstatus` API resource.

Assumptions and Notes

- The attached file is placed on the Clarity LIMS server, in the `/opt/gls/clarity/customextensions` folder.
- The Python API Library (`glsapiutil.py`) is placed on the Clarity LIMS server, in the `/opt/gls/clarity/customextensions` folder. You can download the latest `glsapiutil` library from our [GitHub page](#).
- The `HOSTNAME` global variable must be updated so that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[autoplaceSamplesDefault.py.txt](#)

Publishing Files to LabLink

Some steps produce data that you would like your collaborators to have access to. One way to achieve this access is to mark the files as 'Published' in the BaseSpace Clarity LIMS **Operations Interface**. However, if the number of files to be published is large, this method can be laborious.

This example provides an alternative method and uses a script to publish the files programmatically via the API.

Solution

In this example, we have a protocol step based on a Sanger/capillary sequencing workflow. This step produces up to two files per sample (a **.seq** and a **.ab1** file).

Our example script runs at the end of the protocol step. The script publishes the output files so that they are available to collaborators in the LabLink **Collaborations Interface**.

Parameters

The EPP / automation command is configured to pass the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the protocol step that launches the script - the {stepURI:v2:http} token (Required)

An example of the full syntax used to invoke the script is as follows:

```
bash -l -c "/usr/bin/python /opt/gls/clarity/customextensions/publishFilesToLabLink.py -u
admin -p securepassword -s https://demo-5-1.claritylims.com/api/v2/steps/24-7953"
```

User Interaction

After the script has completed its execution, collaborators are able to view and download the files from the LabLink **Collaborations Interface**.

Project name: Sanger sequencing

Project LIMS ID: HOG55

Status: Open

Open date: 2012-10-31 17:00:00.0

Close date:

Invoice date:

Samples Files Notes

Filter by: All

File extension:

Imported after:

Imported before:

Apply

Name	Date Imported	Project	Sample	Process
Sample1.seq	Nov 1, 2012	Sanger sequencing	Sanger Sample 1	Capillary Electrophoresis (Sanger)
Sample1.ab1	Nov 1, 2012	Sanger sequencing	Sanger Sample 1	Capillary Electrophoresis (Sanger)
Sample2.seq	Nov 1, 2012	Sanger sequencing	Sanger Sample 2	Capillary Electrophoresis (Sanger)
Sample2.ab1	Nov 1, 2012	Sanger sequencing	Sanger Sample 2	Capillary Electrophoresis (Sanger)
Sample3.ab1	Nov 1, 2012	Sanger sequencing	Sanger Sample 3	Capillary Electrophoresis (Sanger)
Sample3.seq	Nov 1, 2012	Sanger sequencing	Sanger Sample 3	Capillary Electrophoresis (Sanger)
Sample4.seq	Nov 1, 2012	Sanger sequencing	Sanger Sample 4	Capillary Electrophoresis (Sanger)
Sample4.ab1	Nov 1, 2012	Sanger sequencing	Sanger Sample 4	Capillary Electrophoresis (Sanger)
Sample5.ab1	Nov 1, 2012	Sanger sequencing	Sanger Sample 5	Capillary Electrophoresis (Sanger)
Sample5.seq	Nov 1, 2012	Sanger sequencing	Sanger Sample 5	Capillary Electrophoresis (Sanger)

About the Code

The main method used in the script is **publishFiles()**. The method in turn carries out several operations:

1. In a single transaction using the 'batch' method, the limsids of the steps' artifacts are gathered and the artifacts are retrieved.
2. Each artifact is investigated. If there is an associated file resource, its limsid is stored.
3. The files resources are retrieved in a single transaction using the 'batch' method.
4. For each file resource, the value of the <is-published> node is set to 'true'.
5. The files resources are saved back to Clarity LIMS in a single transaction using the 'batch' method.

Assumptions and Cotes

- The attached file is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.

- The Python API Library (glsapiutil.py) is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from our [GitHub page](#).
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[publishFilesToLabLink.py.txt](#)

Automatic Pooling Based on a Sample UDF/Custom Field

In some facilities, when samples are initially submitted into BaseSpace Clarity LIMS, it has already been determined which samples are combined to give pooled libraries. In such cases, it is desirable to automate the pooling of samples within the protocol step. Doing so means that the lab scientist does not have to manually pool the samples in Clarity LIMS interface. This automatic pooling saves time and effort and reduces errors.

Solution

This example provides a script that allows 'autopooling' to occur. It also describes how the script can be triggered, and what the lab scientist sees when the script is running.

The attached script relies upon a user-defined field (UDF) / custom field, named **Pooling Group**, at the analyte (sample) level.

This UDF is used to determine the constitution of each pool. This determination makes sure that samples combined to create a pool having the name of the **Pooling Group** value all have a common **Pooling Group** value.

For example, consider the **Operations Interface** (LIMS v4.x and earlier) Samples list shown in the accompanying image. The highlighted samples have a common **Pooling Group** value. Therefore, they combine to create a pool named **210131122-pg1**.

Samples							
✖ Look for: SG-8745		Find	Clear	☐ Exact matches only		☒ Highlight matches	
QC	Sample Name	Workflows	Container	Well (row:column)	Sample LIMS ID ▲	Pooling Group	T
	SG-8745-1	MCL MinSeq	ML0802	A:1	ADM1251A1	210131122-pg1	
	SG-8745-2	MCL MinSeq	ML0802	A:2	ADM1251A2	210131122-pg1	
	SG-8745-3	MCL MinSeq	ML0802	A:3	ADM1251A3	210131122-pg1	
	SG-8745-4	MCL MinSeq	ML0802	A:4	ADM1251A4	210131122-pg1	
	SG-8745-5	MCL MinSeq	ML0802	A:5	ADM1251A5	210131122-pg1	
	SG-8745-6	MCL MinSeq	ML0802	A:6	ADM1251A6	210131122-pg1	
	SG-8745-7	MCL MinSeq	ML0802	A:7	ADM1251A7	210131122-pg1	
	SG-8745-8	MCL MinSeq	ML0802	A:8	ADM1251A8	210131122-pg1	
	SG-8745-9	MCL MinSeq	ML0802	A:9	ADM1251A9	210131122-pg1	
	SG-8745-10	MCL MinSeq	ML0802	A:10	ADM1251A10	210131122-pg1	
	SG-8745-11	MCL MinSeq	ML0802	A:11	ADM1251A11	210131122-pg1	
	SG-8745-12	MCL MinSeq	ML0802	A:12	ADM1251A12	210131122-pg1	
	SG-8745-13	MCL MinSeq	ML0802	B:1	ADM1251A13	210131122-pg1	
	SG-8745-14	MCL MinSeq	ML0802	B:2	ADM1251A14	210131122-pg1	
	SG-8745-15	MCL MinSeq	ML0802	B:3	ADM1251A15	210131122-pg1	
	SG-8745-16	MCL MinSeq	ML0802	B:4	ADM1251A16	210131122-pg1	
	SG-8745-17	MCL MinSeq	ML0802	B:5	ADM1251A17	210131122-pg2	
	SG-8745-18	MCL MinSeq	ML0802	B:6	ADM1251A18	210131122-pg2	
	SG-8745-19	MCL MinSeq	ML0802	B:7	ADM1251A19	210131122-pg2	
	SG-8745-20	MCL MinSeq	ML0802	B:8	ADM1251A20	210131122-pg2	

Step Configuration

In this example, the **Pooling** protocol step is configured to invoke the script when you enter the Pooling screen of the step.

Parameters

The EPP/automation command is configured to pass the following parameters:

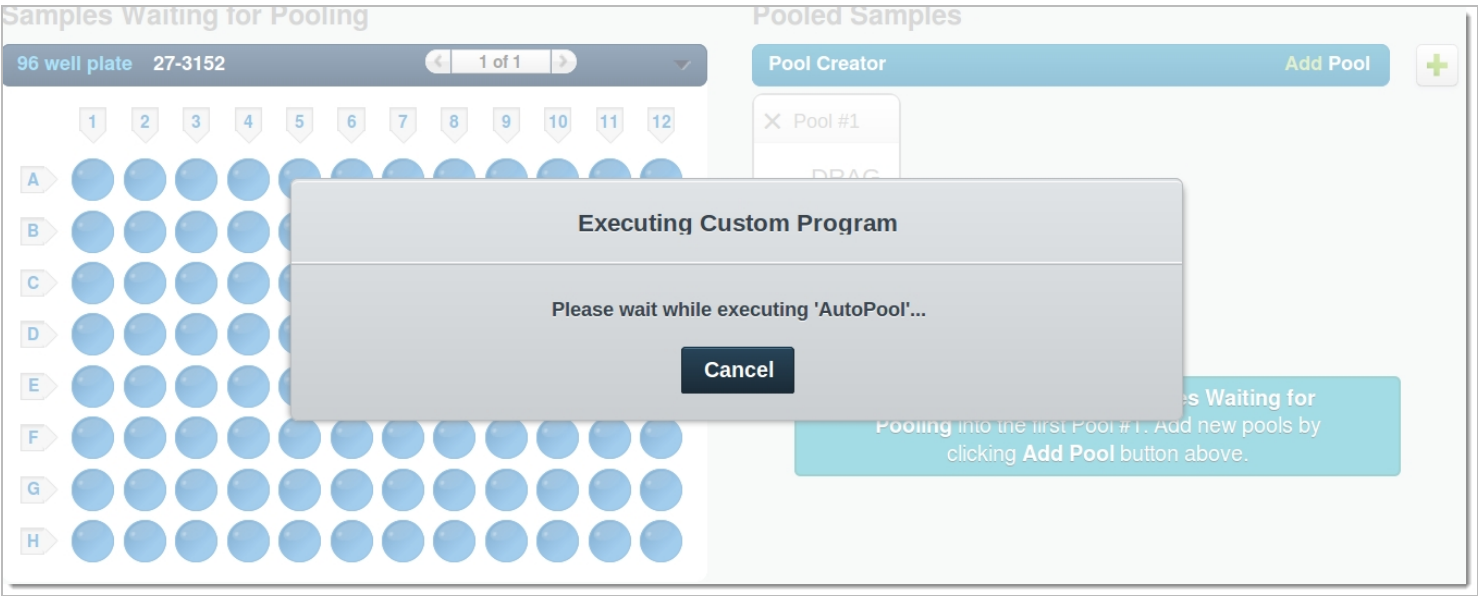
- l The limsid of the process invoking the script (Required)
- u The username of the current user (Required)
- p The password of the current user (Required)
- s The URI of the step that launches the script - the {stepURI:v2:http} token (Required)

An example of the full syntax to invoke the script is as follows:

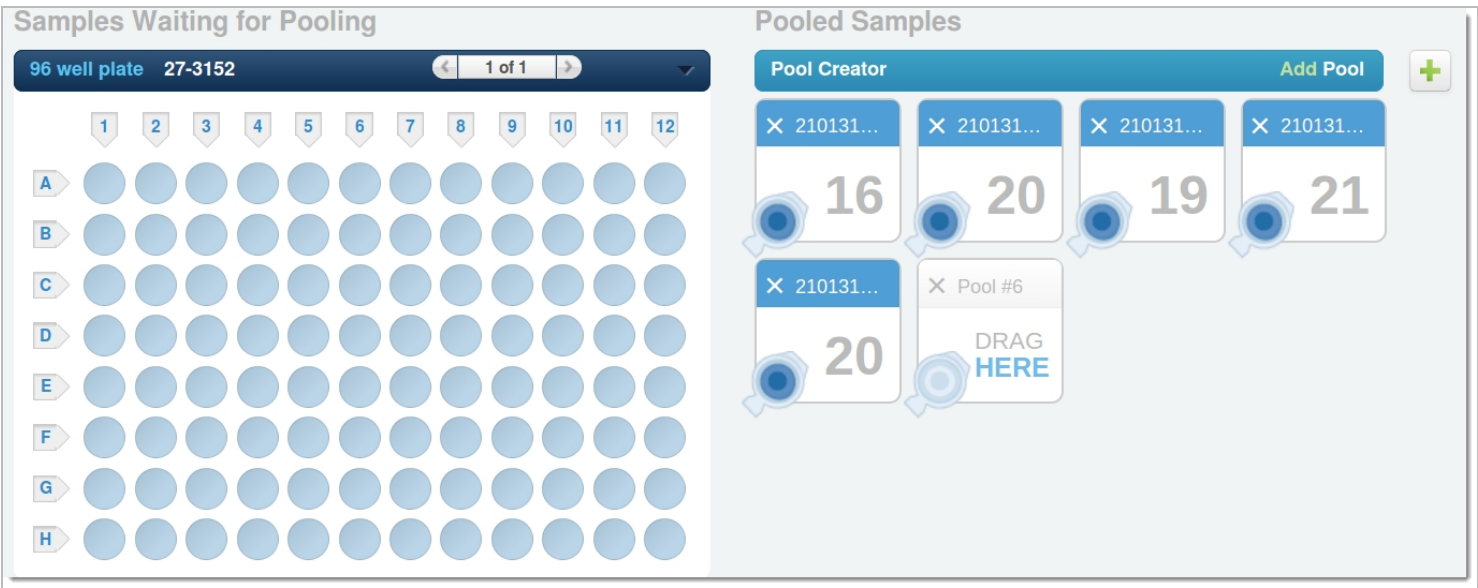
```
/usr/bin/python /opt/gls/clarity/customextensions/autopoolSamples.py -l 122-7953 -u admin -p securepassword -s http://192.168.9.123:8080/api/v2/steps/122-5601
```

User Interaction

When the lab scientist enters the Pooling screen, a message similar to the following displays:



When the script has completed, the rightmost **Placed Samples** area of the Placement screen displays the autocreated pools:



At this point, the lab scientist can review the constituents of each pool, and then complete the protocol step as normal.

About the Code

The main methods of interest are **autoPool()** and **getPoolingGroup()**.

1. The **autoPool()** method harvests information so that subsequent code can retrieve the required objects using the 'batch' API operations. This method involves using additional code to build and manage the cache of objects retrieved in the batch operations, namely:
 - `cacheArtifact()`
 - `prepareCache()`
 - `getArtifact()`
2. After the cache of objects has been built, each artifact is linked to its submitted sample. The **getPoolingGroup** function harvests the **Pooling Group** UDF value of the corresponding submitted sample.
3. The script now understands which artifacts are to be grouped to produce the requested pools. An appropriate XML payload is constructed and then POSTed to the **../steps/<stepID>/placements** API resource.

Assumptions and Notes

- The attached file is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The Python API Library (`glsapiutil.py`) is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest `glsapiutil` library from our [GitHub page](#).
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[autopoolSamples.py.txt](#)

Completing a Step Programmatically

 | This method supersedes the use of the processes API endpoint.

The capacity for completing a step programmatically, without having to open the BaseSpace Clarity LIMS web interface, allows for rapid validation of protocols. This method results in streamlined workflows for highly structured lab environments dealing with high throughput.

This example uses the **/api/v2/steps** endpoint, which allows for more controlled execution of steps. In contrast, a process can be executed using the **api/v2/processes** endpoint with only one POST. This ability is demonstrated in the [Process Execution with EPP/Automation Support](#) example.

Solution

The Clarity LIMS API allows for each aspect of a step to be completed programmatically. Combining the capabilities of the API into one script allows for the completion of a step with one click.

Step Configuration

This example was created for nonpooling, nonindexing process types.

Parameters

The script accepts the following parameters:

iu	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script - the {stepURL:v2:http} token (Required)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/autocomplete-wholestep.py -p apipassword -u apiuser -s https://demo-4-1.claritylims.com/api/v2/
```

User Interaction

The script contains several hardcoded variables, as shown in the following example.

```
step_config_uri = "https://demo-4-1.claritylims.com/api/v2/configuration/protocols/551/steps/1003"step_config = "simple
step"queue = '1003'containerType = "96 well plate"reagentCat = ""replicates = 40nextAction =
'nextstep'nextStepURI = 'https://demo-4-1.claritylims.com/api/v2/configuration/protocols/551/steps/1004'reagent_lot_limsid = ""
```

step_config_uri Is the stage that is automatically completed. Because this script is starting the step, there is no step limsid needed as input parameter to the script. After the script begins the step, it gathers the step limsid from the APIs response to the step-creation post.

```
queue = '1003'
```

About the Code

The **main()** method in the script carries out the following operations:

- startStep()
- addLot()
- addAction()
- addPlacement()
- advanceStep()

Each of these functions creates an XML payload. It interacts with the Clarity LIMS API to complete an activity that you would be doing in the Clarity LIMS interface.

startStep()

This function creates 'stp:step-creation' payload.

As written, the script includes all the analytes in the queue for the specified stage.

addLot()

This function creates a 'stp:lots' payload. This function can be skipped if the process does not require reagent lot selection.

addAction()

This function creates a 'stp:actions' payload. As written, all output analytes are assigned to the same 'next-action'. To see the options available as next actions, refer to the REST API documentation on 'Type action-type'.

 | This example only supports the following next-actions: 'nextstep', 'remove', 'repeat'.

addPlacement()

This function creates a 'stp:placements' payload.

In this example, it is not important where the artifacts are placed, so the analytes are assigned randomly to a well location.

This function relies on the createContainer function, because a step producing replicate analytes may not create enough on-the-fly containers to place all out the output artifacts.

advanceStep()

This function advances the current-state for a step. The **current-state** is an attribute that is found at the **/api/v2/steps/{limsid}** endpoint. It is a representation of the page that you see in the user interface. For more information, refer to the [API Portal](#) and search for the REST API documentation relating to the **/api/v2/steps/{limsid}/advance** endpoint.

This function creates a 'stp:placements' payload. POSTing this payload to **steps/{limsid}/advance** is the API equivalent of moving to the next page of the GUI, with the final advance post completing the step.

Known Issues

- There is a known bug with advance endpoint that prevents a complete programmatic progression through a pooling step.

Assumptions and Notes

- You are running a version of Python supported by Clarity LIMS, as documented in the Clarity LIMS Technical Requirements, within the Clarity LIMS Core Functionality documentation.
- The attached file is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The Python API Library (glsapiutil.py) is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from the [Illumina GitHub resources](#).
- The HOSTNAME global variable must be updated so that it points to your Clarity LIMS server.

- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[autocomplete-wholestep.py.txt](#)

Automatic Sample Placement into Existing Containers

There are often cases where empty containers are received and added into BaseSpace Clarity LIMS before being used in a protocol. This application example describes how to use the API to place samples into existing containers automatically. The application uses a CSV file that describes the mapping between the sample and its destination container.

Furthermore, the API allows accessioning into multiple container categories, something that is not possible through the web interface.

Prerequisites

- If you use Python version 2.6x, you must install the argparse package. Python 2.7 and later include this package by default.
- Also make sure that you have the latest glsapiutil.py Python API library on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from our [GitHub page](#).
- Check the list of allowed containers for the step and make sure that all expected container categories are present. The API cannot place samples into containers that are not allowed for the step!

Structure of the Input File

The suggested input format is a four-column CSV with the following columns:

Sample Name, Container Category, Container Name, Well Position

The sample name must match the name as shown in the **Ice Bucket / Queue** screen.

Step Setup Screen Configuration

First, make sure that the **Step Setup** screen has been activated and is able to accept a file for upload.

EPP / Automation command line

Assuming the file is `compoundOutputFileLuid0`, the EPP / automation command line would be structured as follows:

```
bash -l -c "/usr/bin/env python
/opt/gls/clarity/customextensions/placeSamplesIntoExistingContainers.py -u {username} -p
{password} -s {stepURI:https} -f {compoundOutputFileLuid0}"
```

The automation must be configured to trigger automatically when the **Placement** screen is entered.

« Back Properties **Automation** Next Steps Queue View Step Setup Record Details Next »

Configure trigger timing of automation actions

Name	Behaviour	Stage of Step	Timing
	Automatically initiated	on Placement screen	when screen is entered
	Record Details button	at beginning of step	when screen is entered
	Automatically initiated	on Step Setup screen	when screen is exited
	Not used	on Placement screen	

GenoLogics Life Sciences Software | Support | Operations | Training | Version History

Clarity LIMS v4.x automation trigger configuration

Trigger Location	Trigger Style
Placement >	Automatic upon entry
	Clear Value
	Automatic upon entry
	Automatic upon exit

Clarity LIMS v5.x automation trigger configuration

The Script

i | The attached Python script uses the **prerelease** API endpoint (instead of **v2**), which allows placement of samples into existing containers.

The script performs the following operations:

1. Parses the file and create an internal map (Python dict) between sample name and container details:**Key:** sample name
 - **Value:** (container name, well position) tuple
2. Retrieves the URI of each container.
3. Using a GET request, the script Accesses the 'placements' XML.
4. Performs the following modifications to the XML:

- Populates the **<selected-containers>** node with child nodes for each retrieved container.
 - Populates each **<output>** artifact with a **<location>** node with the container details and well position.
5. PUTs the placement XML back to Clarity LIMS.

After the script runs, the **Placement** screen will show the placements, assuming there were no problems executing the script.

The attached script also contains some minimal bulletproofing for the following cases:

- Container was not found.
- Container is not empty.
- Well position is invalid.
- Sample in the ice bucket does not have a corresponding entry in the uploaded file.
- Sample in the uploaded file is not in the ice bucket.

In all cases, the script reports an error and does not allow the user to proceed.

Attachments

[placeSamplesIntoExistingContainers.py.txt](#)

Routing Output Artifacts to Specific Workflows/Stages

Samples progressing through workflows can branch off and must be directed to different workflows or stages within a workflow.

Example: If it is not known at the initiation of a workflow if a sample is to be sequenced on a HiSeq or MiSeq. Rerouting the derived samples could be necessary.

Solution

This example provides the user with the opportunity to route samples individually to the HiSeq, MiSeq, or both stages from the **Record Details** screen.

Step configuration

The step is configured to display two checkbox analyte UDFs / derived sample custom fields. The fields are used to select the destination workflow/stages for each derived sample. You can choose to queue the sample for HiSeq, MiSeq, or both.

In this example, you select the following:

- Two samples to be queued for HiSeq
- Two samples for MiSeq
- Two that are not routed
- Two samples for both HiSeq and MiSeq

Sample Details *required				
Sample Name ▾	*QC	Go to HiSeq	Go to MiSeq	Go to Co
▼ 96 well plate: Purified plate (8 samples)				
RNA Sample 289		☒	☐	☐
RNA Sample 290		☒	☐	☐
RNA Sample 291		☐	☒	☐
RNA Sample 292		☐	☒	☐
RNA Sample 293		☐	☐	☐
RNA Sample 294		☐	☐	☐
RNA Sample 295		☒	☒	☐
RNA Sample 296		☒	☒	☐

Parameters

The script accepts the following parameters:

-
- u The username of the current user (Required)

 - p The password of the current user (Required)

 - s The URI of the step that launches the script - the {stepURI:v2} token (Required)

An example of the full syntax to invoke the script is as follows:

```
bash -c "/usr/bin/python /opt/gls/clarity/customextensions/Route_to_HiSeq_MiSeq.py -u {username} -p {password} -s {stepURI:v2}"
```

User Interaction

On the **Record Details** screen, you use the analyte UDF / derived sample custom field checkboxes to decide which workflow/stage combination to send each derived sample.

About the Code

The first important piece of information required is the URI of the destination stage.

A stage URI can change across LIMS instances (such as switching from Dev to Prod). Therefore, the script gathers the stage URI from the **workflow and stage name**. This process occurs even when the workflows are identical.

The main method in the script is **routeAnalytes**. This method in turn carries out several operations:

- Gathers the information for the process / master step that triggered the script, including output analytes.
- For each analyte, evaluates which UDFs / custom fields have been set, and adds the analyte to a list of analytes to route.
- Creates the XML message for each stage.
- Does a POST to the REST API to add the analytes to the queue in Clarity LIMS.

Modifications

This script can be modified to look for a process level UDF (master step custom field). Modified in this way, all outputs from the step would be routed to the same step.

This example also serves to demonstrate a more general concept. Routing artifacts is valuable in any situation where a sample must be queued for a stage outside of the usual order of a workflow. This routing approach is valuable even when routing newly submitted samples to the first stage in a workflow.

For more information on using the **artifact/route** endpoint, refer to the REST API Reference, available from the [API Portal](#).

Assumptions and Notes

- You are running a version of Python supported by BaseSpace Clarity LIMS. For details, refer to the *Technical Requirements* provided in the Clarity LIMS Core Functionality documentation section.
- The attached file is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The Python API Library (glsapiutil.py) is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from [Illumina GitHub](#).
- The example code is provided for illustrative purposes only. The example code does not contain sufficient exception handling for use 'as is' in a production environment.
- Samples can be inadvertently duplicated in the next step. This duplication occurs if:
 - The sample is being routed at the last step of a protocol and;
 - If the action of next steps is Mark Protocol as Complete.

This duplication is due to:

- The next step is routing the artifact to its default destination and;
- The script is routing the same artifact.

The solution here is to set the default next steps action to **Remove from Workflow** instead. This solution can be automated using Lab Logic Toolkit or the API.

Attachments

Creating Multiple Containers / Types for Placement

The BaseSpace Clarity LIMS interface offers tremendous flexibility in placing the outputs of a protocol step into new containers.

Sometimes it is necessary that a step produces multiple containers of differing types (for example a 96-well plate **and** a 384-well plate). Such an interaction is not possible without using the Clarity LIMS API.

This example provides a script that creates a 96-well plate and a 384-well plate in preparation for subsequent manual placement of samples.

Solution

In this example, containers are created according to the following logic:

- A 96-well plate is produced along with a 384-well plate.
- Neither plate has a user-specified name. The LIMS names them using the LIMS ID of the plates.

Step configuration

The step is configured:

- To allow samples to be placed in 96 or 384 well plates.
- To invoke the script when the user enters the Sample Placement screen of the step.

Parameters

The EPP / automation command is configured to pass the following parameters:

-l	The limsid of the process invoking the code (Required)	The {processLuid} token
-u	The username of the current user (Required)	The {username} token
-p	The password of the current user (Required)	The {password} token
-s	The URI of the step that launches the script (Required)	The {stepURI:v2:http} token

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/createMultipleContainerTypes.py -l 122-7953 -u admin -p securepassword -s http://192.168.9.123:8080/api/v2/steps/122-5601
```

User Interaction

When the lab scientist enters the Sample Placement screen, the rightmost **Placed Samples** area displays the first container created (1 of 2). By selecting the second container, the display shows the second container (2 of 2).

The screenshot displays the 'ML Create Multiple Container ...' window in the Clarity LIMS interface. It features a top navigation bar with tabs for 'Place Samples', 'Record Details', and 'Assign Next Steps', along with 'Abort', 'Save', and 'Record Details »' buttons. The main area is divided into two panels: 'Samples To Be Placed' and 'Placed Samples'.

The 'Samples To Be Placed' panel shows a 96-well plate layout (8 rows A-H, 12 columns 1-12). Blue circles indicate sample placements at the following coordinates: (A,1), (B,2), (C,3), (C,8), (D,4), (D,7), (D,9), (E,3), (E,6), (E,10), (F,2), (F,5), (F,11), (G,1), (G,4), (G,12), and (H,2), (H,3), (H,12).

The 'Placed Samples' panel shows a 384-well plate layout (8 rows A-H, 48 columns 1-48). The first 12 columns are visible, showing no sample placements. The panel includes a search bar with the value '27-4861' and a '2 of 2' indicator.

The lab scientist manually place the samples into the containers and completes the protocol step as normal.

About the Code

- Two calls to **createContainer()** create the destination 96-well and 384-well plates.
 - To create custom containers, supplement the **createContainer()** method with the configuration details that apply to your instance of Clarity LIMS.
 - To name the container with the value of the second argument, pass a non-empty string as the second argument to **createContainer()** method.
- An XML payload is created containing only the details of the created containers, ready for the user to record the actual placements in the Clarity LIMS user interface.
- This information is POSTed back to the server, in the format required for the placements resource.
- A meaningful message is reported back to the user via the **../steps/<stepID>/programstatus** API resource.

Assumptions and Notes

- The attached file is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The Python API Library (glsapiutil.py) is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder. You can download the latest glsapiutil library from the Illumina [GitHub page](#).
- Update the HOSTNAME global variable so that it points to your Clarity LIMS server.

- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[createMultipleContainerTypes.py.txt](#)

Starting a Protocol Step via the API

In some circumstances, it can be desirable to automate the initiation of a step in BaseSpace Clarity LIMS. In this scenario, the step is thus executed without any user interaction, ie, a liquid-handling robot drives the step to completion. This example provides a solution that allows for automatic invocation of steps via the API.

Solution

Querying the Queues Endpoint

Before we can invoke a step, we must first employ the **queues** endpoint.

Every step displayed in the Clarity LIMS web interface has an associated queue, the contents of which can be queried. This image of the Nextera XT Library Prep shows the samples queued for each step in the **Nextera XT Library Prep** protocol.

Nextera XT Library Prep		Total	50
Step 1 » Tagment DNA (Nextera XT DNA)		10	
Step 2 » PCR Amplification (Nextera DNA) 4.0		10	
Step 3 » PCR Clean-up (Nextera DNA) 4.0		10	
Step 4 » Bead Based Library Normalization		10	
Step 5 » Library Pooling (Nextera XT)		10	

For this example, we investigate the queue for the **Step 1 - Tagment DNA (Nextera XT DNA)** step.

Step 1: Find Step ID

First, we must find the LIMS ID of the step. We query the **configuration/workflows** resource and hone in on the **Nextera XT for MiSeq** protocol:

```
<wkfcnf:workflow status="ACTIVE"
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309" name="Nextera XT for
MiSeq"><protocols> <protocol uri="http://192.168.8.10:8080/api/v2/configuration/protocols/3"
name="DNA Initial QC"/> <protocol
uri="http://192.168.8.10:8080/api/v2/configuration/protocols/302" name="Nextera XT Library
```

```

Prep"/> <protocol uri="http://192.168.8.10:8080/api/v2/configuration/protocols/10"
name="Illumina SBS (MiSeq)"/></protocols><stages> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/690" name="DNA
Initial QC"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/691" name="Tagment
DNA (Nextera XT DNA)"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/692" name="PCR
Amplification (Nextera DNA) 4.0"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/693" name="PCR
Clean-up (Nextera DNA) 4.0"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/694" name="Bead
Based Library Normalization"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/695" name="Library
Pooling (Nextera XT)"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/696" name="Sort
MiSeq Samples (MiSeq) 4.0"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/697" name="Library
Normalization (MiSeq) 4.0"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/698" name="Library
Pooling (MiSeq) 4.0"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/699" name="Denature,
Dilute and Load Sample (MiSeq) 4.0"/> <stage
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/700" name="MiSeq Run
(MiSeq) 4.0"/></stages></wkfcnf:workflow>

```

From the XML returned, we can see that the **Tagment DNA (Nextera XT DNA)** step has an associated stage, with an ID of 691:

```

<stage uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/ 691 "
name="Tagment DNA (Nextera XT DNA)"/>

```

Step 2: Find Stage ID

If we now query this stage ID, we see something similar to the following:

```

<stg:stage index="0" name="Tagment DNA (Nextera XT DNA)"
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309/stages/691"> <workflow
uri="http://192.168.8.10:8080/api/v2/configuration/workflows/309"/> <protocol
uri="http://192.168.8.10:8080/api/v2/configuration/protocols/302"/> <step
uri="http://192.168.8.10:8080/api/v2/configuration/protocols/302/steps/567"/> </stg:stage>

```

We now have the piece of information we need: namely the **ID 567** that is associated with this step.

Step 3: Query the Queues Resource

We can use this ID to query the **queues** resource, which provides us with something similar to the following:

```
<que:queue name="Tagment DNA (Nextera XT DNA)" protocol-step-  
uri="http://192.168.8.10:8080/api/v2/configuration/protocols/302/steps/567"  
uri="http://192.168.8.10:8080/api/v2/queues/567"> <artifacts> <artifact limsid="HES208A1PA1"  
uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A1PA1"> <queue-time>2013-04-  
07T17:01:00.636-07:00</queue-time> <location> <container  
uri="http://192.168.8.10:8080/api/v2/containers/27-2654" limsid="27-2654"/>  
<value>1:1</value> </location> </artifact> <artifact limsid="HES208A2PA1"  
uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A2PA1"></artifact> <artifact  
limsid="HES208A3PA1" uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A3PA1"></artifact>  
<artifact limsid="HES208A4PA1"  
uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A4PA1"></artifact> <artifact  
limsid="HES208A5PA1" uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A5PA1"></artifact>  
<artifact limsid="HES208A6PA1"  
uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A6PA1"></artifact> <artifact  
limsid="HES208A7PA1" uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A7PA1"></artifact>  
<artifact limsid="HES208A8PA1"  
uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A8PA1"></artifact> <artifact  
limsid="HES208A9PA1" uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A9PA1"></artifact>  
<artifact limsid="HES208A10PA1"  
uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A10PA1"></artifact>  
</artifacts></que:queue>
```

This result matches the information displayed in the Clarity LIMS web interface. In the next image, we can see the derived samples awaiting the step.


/ Nextera XT Library Prep / Tagment DNA (Nextera XT DNA)

Queue (10 samples)

Options

0 Samples Selected

#	Sample Name	Well	Container Name	Waiting	
Single Well: Tube 10 Samples Add Group					
1	FFPE_Tumor 1		27-2654	415 days	Add
2	FFPE_Tumor 2		27-2655	415 days	Add
3	FFPE_Tumor 3		27-2656	415 days	Add
4	FFPE_Tumor 4		27-2657	415 days	Add
5	FFPE_Tumor 5		27-2658	415 days	Add
6	FFPE_Tumor 6		27-2659	415 days	Add
7	FFPE_Tumor 7		27-2660	415 days	Add
8	FFPE_Tumor 8		27-2661	415 days	Add
9	FFPE_Tumor 9		27-2662	415 days	Add
10	FFPE_Tumor 10		27-2663	415 days	Add

Initiating the Step

Now that we have the contents of the queue, starting the step programmatically is simple.

All that is required is a POST to the **steps** API endpoint. The XML input payload to the POST request takes the following form:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><tmp:step-creation
xmlns:tmp="http://genologics.com/ri/step"> <configuration
uri="http://192.168.8.10:8080/api/v2/configuration/protocols/302/steps/567"/> <container-
type>96 well plate</container-type> <inputs> <input
uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A1PA1" replicates="1"/>
</inputs></tmp:step-creation>
```

If the POST operation was successful, the API returns XML of the following form (for details, see [About the Code](#) section):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><stp:step
xmlns:stp="http://genologics.com/ri/step" current-state="Placement" limsid="24-19301"
uri="http://192.168.8.10:8080/api/v2/steps/24-19301"> <configuration
uri="http://192.168.8.10:8080/api/v2/configuration/protocols/302/steps/567">Tagment DNA
```

```
(Nextera XT DNA)</configuration> <actions uri="http://192.168.8.10:8080/api/v2/steps/24-19301/actions"/> <placements uri="http://192.168.8.10:8080/api/v2/steps/24-19301/placements"/> <details uri="http://192.168.8.10:8080/api/v2/steps/24-19301/details"/> <available-programs/></stp:step>
```

User Interaction

In the Clarity LIMS web interface, two pieces of evidence indicate that the step has been initiated:

- The partially completed step is displayed in the **Work in Progress** area.
- The **Recent Activities** area shows that the protocol step was started.

WORK IN PROGRESS

Nextera XT Library Prep	Step 1 » Tagment DNA (Nextera XT DNA)	4 minutes ago	1	Continue
Infinium MicroArray	Step 1 » Make MSA3 Plate	19 days ago	14	Continue
Master Mix Protocol	Step 1 » Master Mix Calculation	19 days ago	6 ✓	Continue

AVAILABLE WORK

RECENT ACTIVITIES

- System Administrator Started Step 1 » Tagment DNA (Nextera XT DNA)
Nextera XT Library Prep
3 minutes ago
- System Administrator Started Step 1 » Make MSA3 Plate
Infinium MicroArray
19 days ago
- System Administrator Started Step 1

About the Code

The XML payload POSTed to the steps resource is simple in nature. In fact, there are only three variables within the payload:

1. The step to be initiated:

```
<configuration uri="http://192.168.8.10:8080/api/v2/configuration/protocols/302/steps/567"/>
```

2. The type of output container to be used (if appropriate):

```
<container-type>96 well plate</container-type>
```

3. The URI of the artifacts on which the step runs, along with the number of replicates the step must create (if appropriate):

```
<input uri="http://192.168.8.10:8080/api/v2/artifacts/HES208A1PA1" replicates="1"/>
```

Attachments

None

Setting Quality Control Flags

A key reason to track samples is to monitor their quality. In BaseSpace Clarity LIMS, samples are flagged with a checkmark to indicate good quality (QC Pass) or an X to indicate poor quality (QC Fail).

There are many ways to determine quality, including concentration and DNA 260/280 light absorbance measurements. This example uses a tab-separated value (TSV) results file from a Thermo NanoDrop Spectrophotometer to:

- Record concentration and 260/280 measurements, and;

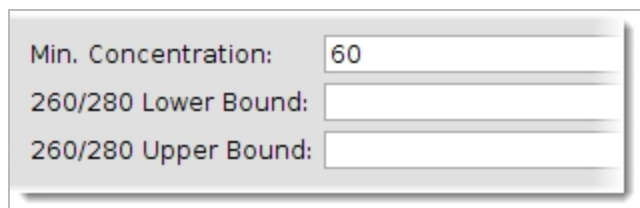
- Set quality flags in the LIMS

In this example, after the script is installed, the user runs the script, records a step, and imports the results file. The EPP / automation script does the rest of the work, by reading the file, capturing the measurements in Clarity LIMS, and setting the QC flags.

i | As spectrophotometers are used for many measurements, within many lab protocols, file formats can vary depending on the instrument software settings. Check your instruments for specific file formats and use the script example to get familiar with the QC example scripts.

User Interaction

1. The user selects samples and runs the **QC Example** step.
2. In the **Operations Interface**(LIMS v4 and earlier), the user sets the required minimum concentration and/or 260/280 lower and upper bounds.

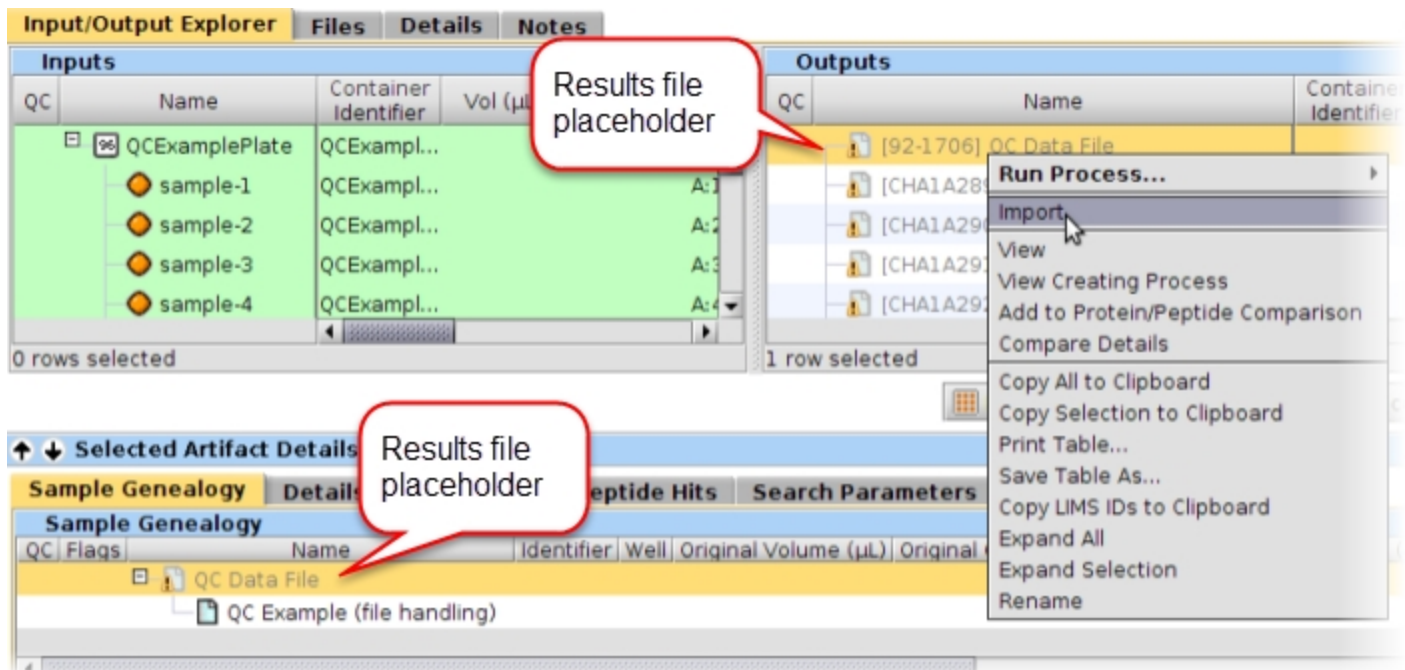


Min. Concentration:

260/280 Lower Bound:

260/280 Upper Bound:

3. The **QC Example** process creates an output artifact (shared ResultsFile) called **QC Data File**. This file is shown in the **Sample Genealogy** and **Outputs** panes. The "!" icon indicates that this entry is a placeholder for a file.
4. The user loads samples onto the spectrophotometer and follows the protocol of the instrument for QC measurement.
5. After the measurements are complete, the user exports the TSV results file created by the spectrophotometer, using the NanoDrop software.



6. The user imports the TSV file into the LIMS:

As Clarity LIMS parses the file, the measurements are captured and stored as output user-defined fields (UDFs). The QC Pass/Fail flags are then set on the process inputs. The flags are set according to whether they meet the concentration and/or 260/280 bounds specified in Step 2.

Inputs				
QC	Name	Container Identifier	Vol (μL)	Conc (μg/μL)
✗	QCExamplePlate	QCExempl...		
✓	sample-1	QCExempl...		
✗	sample-2	QCExempl...		
✓	sample-3	QCExempl...		
✓	sample-4	QCExempl...		

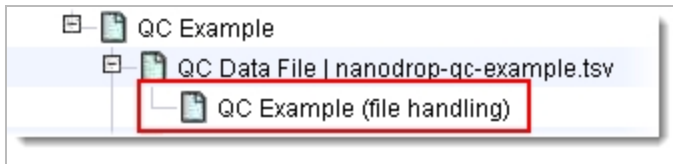
Example

1. The NanoDrop spectrophotometer creates the TSV file. The specific file used in this example is shown:

```
Default 1/6/2011 5:27 PM Sample ID ng/uL A260 260/280 260/230 Constant QCExamplePlate_A01_
DNAIsolate-1 62.49 1.450 1.83 1.86 50 QCExamplePlate_A02_DNAIsolate-2 49.50 1.413 1.99 1.88
50 QCExamplePlate_A03_DNAIsolate-3 70.00 1.198 1.84 2.03 50 QCExamplePlate_A04_DNAIsolate-4
62.49 1.450 1.83 1.86 50 QCExamplePlate_A05_DNAIsolate-5 49.50 1.413 1.99 1.88 50
```

```
QCExamplePlate_A06_DNAIsolate-6 70.00 1.198 1.84 2.03 50 QCExamplePlate_A07_DNAIsolate-7
62.49 1.450 1.83 1.86 50 QCExamplePlate_A08_DNAIsolate-8 49.50 1.413 1.99 1.88 50
QCExamplePlate_A09_DNAIsolate-9 70.00 1.198 1.84 2.03 50
```

- After the TSV file is imported and attached to the **QC Example** process in Clarity LIMS, a second script is automatically called with EPP. This second EPP script is part of a second process, called **QC Example(file handling)**. The file attachment event triggers the second EPP script. You can see the process in the **Sample Genealogy** pane:



- The example uses the TSV file **Sample ID** value to locate the plate and well location to which the QC flag is to be applied. In the example file shown in Step 1: The container is **QCExamplePlate**.
 - The location **A01** maps to the sample on the first well of the container named **QCExamplePlate**.
 - The data contained in the TSV file is captured in Clarity LIMS and can be viewed on the **Details** tab.

↑

↓

Selected Artifact Details

Sample Genealogy

Details

Protein Hits

Peptide Hits

Search Parameters

Details for

Output per Input: Result ▼

Detail Comparison

QC	Name	LIMS ID	Concentration	Absorbance	260/280	260/230	Constant
	Result for sample-2	ACC6A290QC2	49.5	1.41	1.99	1.88	50.00
	Result for sample-1	ACC6A289QC2	62.5	1.45	1.83	1.86	50.00
	Result for sample-3	ACC6A291QC2	70.0	1.20	1.84	2.03	50.00
	Result for sample-4	ACC6A292QC2	62.5	1.45	1.83	1.86	50.00
	Result for sample-5	ACC6A293QC2	49.5	1.41	1.99	1.88	50.00
	Result for sample-6	ACC6A294QC2	70.0	1.20	1.84	2.03	50.00

- i** Notice that the user ran only one process (**QC Example**), but two processes were recorded. The first EPP script created the second process, **QC Example (file handling)**, using the REST API. Using REST to create a process is described in the [Running a Process](#) Cookbook example.
- The NanoDrop QC algorithm in the script compares the concentration and the 260/280 ratio for each sample in the imported TSV file. The values are entered into the process UDFs.
 - A **QC Fail** flag is applied:
 - If the concentration of the sample is less than specified, or;
 - If its 260/280 ratio is outside the bounds given when running the process.
 - A **QC Pass** flag is applied when the sample has values inside the parameters provided.

- Samples with no associated values in the TSV file are unaffected.

In this example, the minimum concentration is set to 60. A QC **Fail** flag is applied to sample-2 because its concentration level does not meet the minimum value specified.

The screenshot shows the 'Input/Output Explorer' and 'Selected Artifact Details' panels. The 'Inputs' table lists samples, with sample-2 marked as a QC failure. The 'Outputs' table lists generated data files. The 'Selected Artifact Details' panel shows a 'Detail Comparison' table with sample data.

QC	Name	Container Identifier	Vol (μL)	Conc (μM)
✗	96 QCExamplePlate	QCExempl...		
✓	sample-1	QCExempl...		
✗	sample-2	QCExempl...		
✓	sample-3	QCExempl...		
✓	sample-4	QCExempl...		

0 rows selected

QC	Name	Container Identifier	Orig
	[92-1706] QC Da...		nan
	[CHA1A289QC2] ...		
	[CHA1A290QC2] ...		
	[CHA1A291QC2] ...		
	[CHA1A292QC2] ...		

96 rows selected

BeadStudio Gene Expr...

Selected Artifact Details

Sample Genealogy Details Protein Hits Peptide Hits Search Parameters

Details for Output per Input: Result Show No

QC	Name	LIMS ID	Concentration	Absorbance	260/280	260/230	Constant
	Result for sample-1	CHA1A289QC2	62.5	1.45	1.83	1.86	50.00
	Result for sample-2	CHA1A290QC2	49.5	1.41	1.99	1.88	50.00
	Result for sample-3	CHA1A291QC2	70.0	1.20	1.84	2.03	50.00
	Result for sample-4	CHA1A292QC2	62.5	1.45	1.83	1.86	50.00

0 rows selected

- i** | Automation/EPP can be used to process files when they are attached to the LIMS. Using file attachment triggers is sometimes called **data analysis pipelining**. Basically, a series of analysis steps across a chain of processes is triggered from one file attachment. For more information, refer to [Triggers and Command Line Calls](#).

Installation

1. Download the zip file to the server (on a non-production server use the **gls** user account).
2. Unzip the file to the following directory: **/opt/gls/clarity/Applications**. The contents of the zip file is installed within that directory, to **CookBook/NanoDropQC/**.

```
unzip -d /opt/gls/clarity/Applications file-qc-2.0-bundle.zip
```

3. Next, unzip the config-slicer-<version>-deployment-bundle.zip in /opt/gls/clarity/Applications/CookBook/NanoDropQC/. Replace <version> with the version number of the included config-slicer.

```
cd /opt/gls/clarity/Applications/CookBook/NanoDropQC/ unzip config-slicer--deployment-bundle.zip
```

4. With Clarity LIMS running, run the following server command-line call to import the required configuration into the server (ie, the process, sample, and fields used by the scripts):

```
java -jar config-slicer-.jar \ -u -p -host \ -o import -k file-qc-package.xml
```

Installation Validation

To confirm that the example is correctly installed, follow these steps to simulate the recording of QC information by a user in the lab:

1. Start the Clarity LIMS **Operations Interface** client.
2. Create a project, and submit a 96-well plate named **QCExamplePlate** full of samples.
3. Select the samples and run the **QC Example** process.
4. Select the **Next** and **Done** buttons to complete the wizard.
5. Next, in the **Input/Output Explorer** of the process summary tab, you see the QC Data File in the **Outputs** pane and in the **Sample Genealogy**. Right-click this placeholder and select **Import**.
6. Import the example TSV **nanodrop-qc-example.tsv** file provided in the zip file.
7. Wait for the QC flags to become visible on a subset of the samples used as inputs to the process (located from A:1 to A:9).

Example Modifications

You can modify the example script to suit the QC experimental methods and calculations of your lab. For example, you can consider the phenotypic information or have extra sample data recorded in the LIMS. Two modifications to the example are described next.

Recording QC measurements on sample inputs instead of file outputs

The example script writes the measurements into user-defined fields (UDFs) associated with **outputs** of the process. This script allows multiple measurements to be recorded for one sample, by running the process multiple times. Each time the process is run on an input sample, a new process with new output results is recorded in the LIMS.

You could write the measurements into UDFs associated with the **input** samples. For example, to keep the data records simple: the greater the number of outputs recorded in the LIMS, the more confusing it becomes for the user to upload files and navigate results. Setting the fields on the inputs provides a single 'golden' value.

To change the configuration and script to set QC flags and field values on inputs:

1. Change the code in **NanoDrop.groovy** so that UDFs are set on inputs instead of outputs. That is, replace this line:

```
setUDF(output, <udf-name>, <value>)
```

with the following:

```
setUDF(input, <udf-name>, <value>)
```

Because you are no longer changing the outputs, you can comment or delete the line where the output is saved:

```
// Save changes client.httpPUT(input.@uri, input) // client.httpPUT(output.@uri, output)
```

2. Run the **QC Example (preparation)** process that was included in the configuration package you imported into your system.

i | The process wizard provides the option for you to either generate a new plate (container) or select a preexisting plate to hold the process outputs. The results of this process are placed into a plate that is different than the one in which you originally the samples.

3. Run the **QC Process** on the plate created in the previous step.
4. Edit the **nanodrop-qc-example.tsv** file to reflect the name of this plate.

i | Remember that the **Sample ID** column in the NanoDrop QC data file depends on the plate name.

To do this edit, for each row replace **QCExamplePlate** with the name of the plate that now holds the outputs of the process-generated plate names. Enter the name in a format similar to "27-124".

5. Import the modified **nanodrop-qc-example.tsv** into the result file placeholder generated by the **QC Example** process.
6. Wait for the QC flags to update on the inputs for which the **nanodrop-qc-example.tsv** file has measurements.
7. This time, the measurements appearing in the outputs of the **QC Example** process are now collected as UDF values on the inputs of that process. To see these measurements, select the outputs of the parent process, **QC Example (preparation)**, and view the output **Details** tab.

i | State on input and output: To ensure you are working with the latest version of an artifact, do not specify a state when retrieving artifacts via the REST API. In the example 'NanoDrop.groovy' script provided, the 'stripStateQuery' closure strips the state from the input and output URIs as it iterates through the input-output-map.

Using Additional Information Stored in Sample Fields to Set the QC Flags

Most labs use multiple factors to determine sample QC flags. These factors might be associated with the submitted sample, multiple instrument measurements, or even the type of project or sample.

To demonstrate how easy it is to aggregate multiple factors into the QC flag logic, a boolean field called **Human** is added to the sample configuration. The script logic is modified to set flags only for human samples.

To change the configuration and script to check for human samples:

1. Change the code in **NanoDrop.groovy** so that we make sure that we are dealing with a **Human** sample. To change the code, modify the loop at the end of the script from this example:

```
// all inputs we have NanoDrop results for exist, we can now assign QC status
nanodropResults.each { def (input, output) = identifiedIOPairs[it.key] // Collect
measurements as UDF values ... // Determine QC flags ... // Save changes ... }
```

to this next example:

```
// All inputs we have NanoDrop results for exist, we can now assign QC status
nanodropResults.each { def (input, output) = identifiedIOPairs[it.key] // Skip non-human
samples def sample = client.httpGET(input.'sample'[0].@uri) def isHuman = getUDF(sample,
'Human', 'false').toBoolean() if (!isHuman) { return } // Collect measurements as UDF values
... // Determine QC flags ... // Save changes ... }
```

2. Configure a **checkbox** UDF on Sample (this action was done for you when you imported the configuration package provided in this application example).
3. Submit a new batch of samples on a 96 well plate.
4. Edit samples on wells from A:1 to A:3 so that the **Human** checkbox field is selected.
5. Run the **QC Example** process on all 96 samples in the plate.
6. In the **nanodrop-qc-example.tsv** file, update the **Sample ID** column with the correct plate name.
7. Import the modified **nanodrop-qc-example.tsv** into the result file placeholder generated by the **QC Example** process.
8. Wait for the QC flags to update on the inputs for which the NanoDrop QC data file has measurements.

i | Only the inputs A:1, A:2 and A:3 have QC flags assigned. The other inputs, including those inputs for which the NanoDrop QC data file has data, are left untouched because of the selection criteria implemented.

Assumptions and Notes

- Clarity LIMS v1 or later (API v2 r14)
- Groovy 1.7.4 or later (expected location: **/opt/gls/groovy/current /**)

i | All prerequisites are preloaded if you install on a non-production server.

Attachments

[file-qc-2.0-bundle.zip](#)

Applying Indexing Patterns to Containers Automatically

The indexing of samples is often performed in patterns, based upon the location of the samples in the container.

This example shows how to automate the default placing of reagents on samples, based on their container position. This greatly reduces the amount of time spent on the **Add Reagents** screen (LIMS v4.x) / **Add Labels** screen (LIMS v5.x) and also reduces user error.

In this example, reagent labels are assigned to samples in a predetermined pattern as the user enters the **Add Reagents** screen. This pattern is applied to all containers entering this stage.

Solution

The example **AssignIndexPathPattern.groovy** script is configured to run on the **Adenylate Ends and Ligase Adapters (TruSeq DNA) 4.0** step.

Parameters

The script accepts the following parameters:

-i	The URI of the step that launches the script (Required)	The {stepURI:v2:http} token, in the form: <i>http://<Hostname>/api/v2/steps/<ProtocolStepLimsid></i>
-u	The username of the API user (Required)	The {username} token
-p	The password of the API user (Required)	The {password} token

An example command line is shown below:

```
bash -c "/opt/gls/groovy/current/bin/groovy -cp /opt/groovy/lib /opt/gls/clarify/customextensions/AssignIndexPathPattern.groovy -i {stepURI:v2:http} -u {username} -p {password}"
```

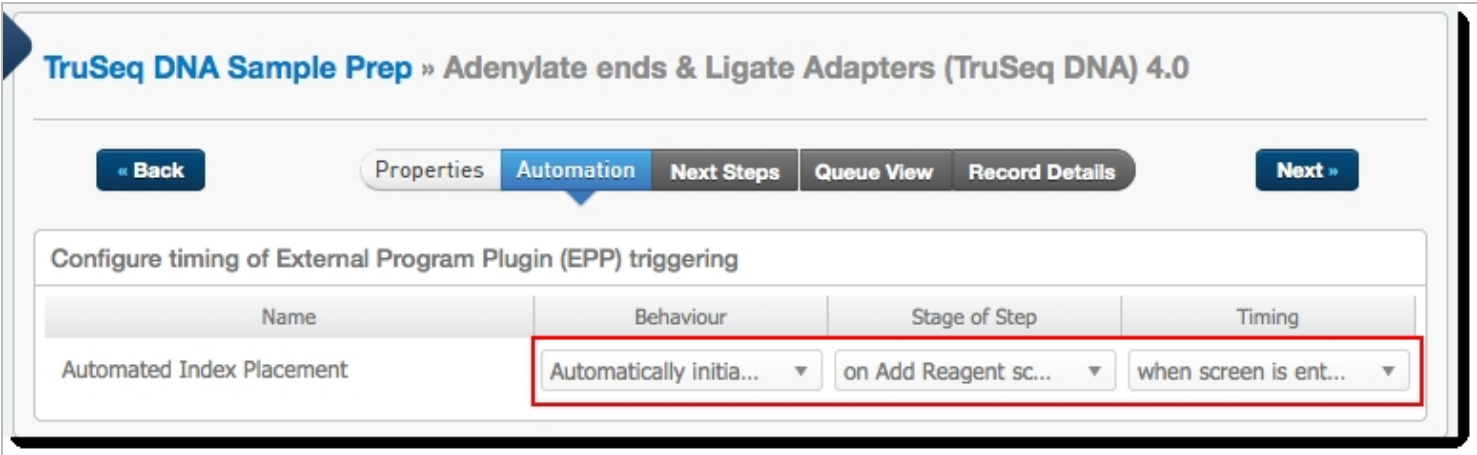
i | The location of Groovy on your server may be different from the one shown in this example. If this is the case, modify the script accordingly.

Step Configuration

In the BaseSpace Clarity LIMS web interface , for the **Adenylate Ends and Ligase Adapters (TruSeq DNA) 4.0** step (in the **TruSeq DNA Sample Prep** protocol), configure **Automation** as follows:

LIMS v4.x

- **Behavior:** Automatically initiated
- **Stage of Step:** On Add Reagent screen
- **Timing:** When screen is entered



Ensure that the protocol is using the **Illumina TruSeq DNA LT** reagent category.

LIMS v5.x

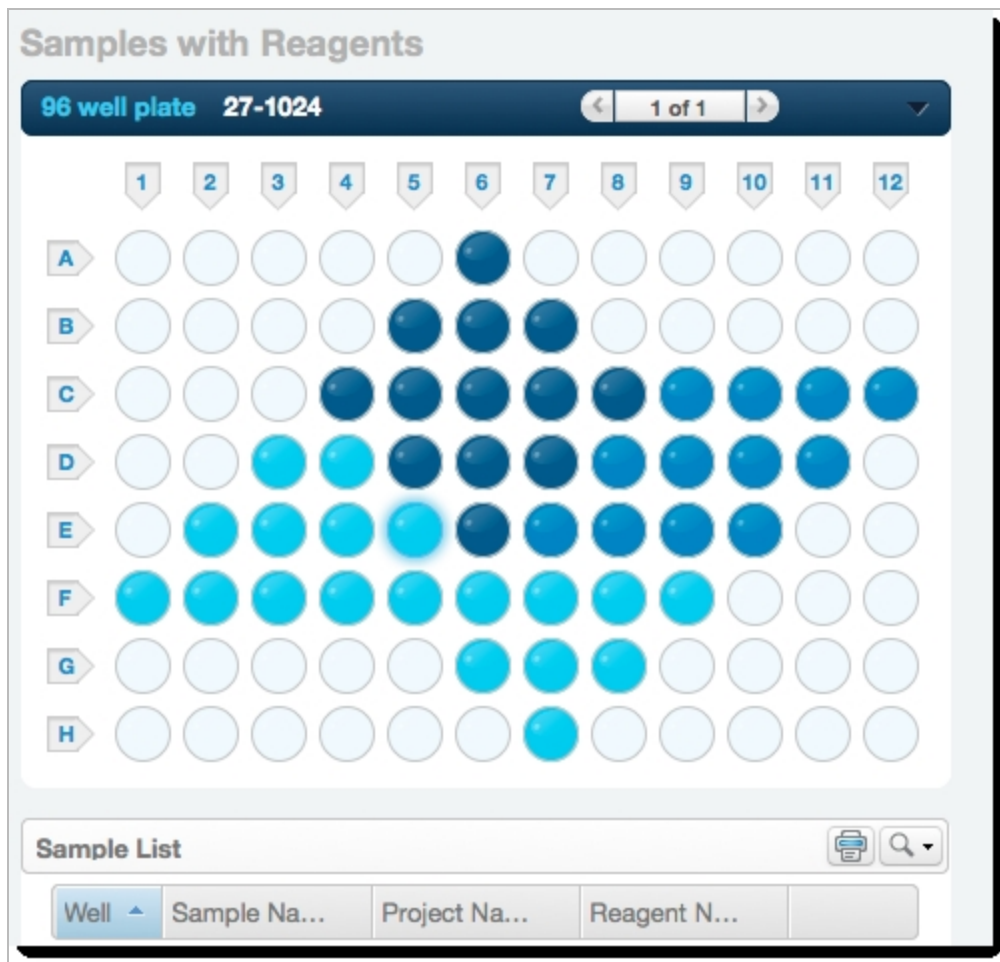
- **Trigger Location:** Add Labels
- **Trigger Style:** Automatic upon entry

Trigger Location	Trigger Style
Add Labels >	Automatic upon entry >

User Interaction

Assuming the user has added 96 samples and has reached the **Adenylate Ends and Ligate Adapters (TruSeq DNA) 4.0** step:

1. The user transfers all 96 samples to a new 96-well plate and proceeds with step.
2. When the user enters the **Add Reagents / Add Labels** screen, the script is initiated. A message box alerts the user that a custom script is in progress.
3. Upon completion, the previously defined success message displays.
4. When the success message is closed, the **Add Reagents / Add Labels** screen loads, and the pattern shown below is applied to samples.



About the Code

Once the script has processed the input and ensured that all the required information is available, we can start applying the reagents to our samples.

- To begin, we need to define the reagents and pattern to apply.
 - The storing of reagents can be accomplished by placing the reagents in a **Map**, comprised of the reagent names indexed by their respective number, i.e. 'AD030' indexed at 30.
 - The pattern can be stored as a **List of Lists**. This can be arranged as a visual representation of the pattern to be applied.

```
// Reagent Labels public static final def REAGENT_MAP = [ (27):'AD027 (ATTCCT)', (23):'AD023
(GAGTGG)', (20):'AD020 (GTGGCC)', (15):'AD015 (ATGTCA)' // Reagent Pattern public static
final def REAGENT_PATTERN = [ [27,27,27,27,27,15,27,27,27,27,27,27],
[27,27,27,27,15,15,15,27,27,27,27,27], [27,27,27,15,15,15,15,15,20,20,20,20],
```

```
[27,27,23,23,15,15,15,20,20,20,20,27], [27,23,23,23,23,15,20,20,20,20,27,27],
[23,23,23,23,23,23,23,23,23,27,27,27], [27,27,27,27,27,23,23,23,27,27,27,27],
[27,27,27,27,27,27,23,27,27,27,27,27] ]
```

- Once we have our reagents and pattern defined, we can start processing the samples:

- We start by retrieving the node from the **reagent setup** endpoint. We use this node as a base for subsequent commands.
- We then gather the unique output artifact URIs and retrieve the output artifacts using **batchGET**:

```
// Retrieve the reagent setup Node reagentSetup = GLSRestApiUtils.httpGET(stepURI +
'/reagents', username, password) // Collect the artifact URIs and retrieve the artifacts def
artifactURIs = reagentSetup.'output-reagents'.'output'.collect { it.@uri }.unique() def
artifacts = GLSRestApiUtils.batchGET(artifactURIs, username, password)
```

- Next, we iterate through our list of output artifacts.
- For each artifact, we determine its position and use its components to index our pattern. This allows us to determine which reagent should be placed on which sample.
- Once we determine the reagent's name, we create a **reagent-label** node with a name attribute equal to the desired reagent name.
- In the list of **output-reagents** in the reagent setup node, we find the **output** that corresponds to the output artifact that we are processing and add our **reagent-label** node to it.

i | We must strip off the state from our artifact's URI. The URIs stored in the step setup node are stateless and will not match the URI returned from our output artifact.

```
// For each artifact, determine its position and set its reagent label accordingly
artifacts.each { artifact -> // Split the position into its two components def
positionIndices = parsePlacement(artifact.'location'[0].value[0].text()) // Using our
relationship maps, determine which reagent should be placed at that position String
reagentName = REAGENT_MAP[((REAGENT_PATTERN[positionIndices[0]])[positionIndices[1]])] < //
Create and attach the reagent-label node to our setup Node reagentNode =
NodeBuilder.newInstance().reagent-label(name:reagentName) reagentSetup.'output-
reagents'.'output'.find { it.@uri == GLSRestApiUtils.stripQuery(artifact.@uri) }.append
(reagentNode) }
```

- Once we have processed all of our output artifacts, we POST our modified setup node to the **reagentSetup** endpoint. This updates the default placement in the API.
- We then define our success message to display to the user upon the script's completion.

```
// Set the reagent setup in the API GLSRestApiUtils.httpPOST(reagentSetup,
reagentSetup.@uri, username, password) // Define the success message to the user
outputMessage = "Script has completed successfully.${LINE_TERMINATOR}" + "Clarity LIMS
reagent pattern has been applied to all containers."
```

Assumptions and Notes

- Your configuration conforms with the script requirements documented in [Solution](#).
- You are running a version of Groovy that is supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The attached Groovy file is placed on the LIMS server, in the following location:
/opt/gls/clarity/customextensions
- **GLSRestApiUtils.groovy** is placed in your Groovy **lib** folder.
- You have imported the attached Reagent XML file into your system using the Config Slicer tool.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[Single Indexing ReagentTypes.xml.txt](#)

[AssignIndexPattern.groovy.txt](#)

Assignment of Sample Next Steps Based On a UDF

In the default configuration of BaseSpace Clarity LIMS, at the end of every step, the user is required to choose where the samples will go next - i.e., the 'next step'.

If samples in the lab follow a logical flow based on business logic, this is an unnecessary manual task. This example shows how to automate this next step selection, to reduce error and user interaction.

This example uses the Automatically Assign Next Protocol Step (Example) step, in the Automation Examples (API Cookbook) protocol. The examples shoes how to:

- Automate the selection of a sample's **Next Steps** , as displayed on the **Assign Next Steps** screen of this step.
- Use the **Pooling** sample UDF / custom field to determine which next step each sample is assigned.

Solution

Step Configuration

The **Automatically Assign Next Protocol Step (Example)** step has two permitted **Next Steps**:

- Confirmation of Low-plexity Pooling (Example)
- Automated Workflow Assignment (Example)

Depending on the value of a sample's **Pooling** UDF / custom field, the sample's **Next Step** will default to one of the permitted next steps:

- If the value of the **Pooling** UDF / custom field is any case combination of **No** or **None** , the sample's next step will default to **Automated Workflow Assignment (Example)**.

- Otherwise, the sample's next step will default to **Confirmation of Low-plexity Pooling (Example)**.

Example showing next step configuration (LIMS v4.x shown)

Automation Examples (API Cookbook) » Automatically Assign Next Protocol Step (Example)

« Back Properties Automation **Next Steps** Queue View Record Details Next »

Configure Permitted Next Steps

Robotic Sample Placement (Example)

Automated DNA Index Assignment (Example)

Automatically Assign Next Protocol Step (Example) - (current step) ←

☒ Confirmation of Low-plexity Pooling (Example)

☒ Automated Workflow Assignment (Example)

Automation is configured as follows:

- Behavior:** Automatically initiated
- Stage of Step:** On **Record Details** screen
- Timing:** When screen is exited

Parameters

The script takes three basic parameters:

-u	The username of the API user (Required)	The {username} token
-p	The password of the API user (Required)	The {password} token
-i	The URI of the step that launches the script (Required)	The {stepURI:v2:http} token, in the form: <i>http://<Hostname>/api/v2/steps/<ProtocolStepLimsid></i>

An example command line is shown below.

(Note: The location of groovy on your server may be different from the one shown in this example. If this is the case, modify the script accordingly.)

```
bash -c "/opt/gls/groovy/current/bin/groovy -cp /opt/groovy/lib
/opt/gls/clarity/customextensions/NextStepAutomation.groovy -u {username} -p {password} -i
{stepURI:v2:http}"
```

User Interaction

Assuming samples have been placed in the protocol and are ready to be processed, the user proceeds as normal:

1. Upon reaching the transition from the **Record Details** screen to the **Assign Next Steps** screen, the script is run. A message box alerts the user that a custom script is in progress.
2. Upon completion of the script, a custom success message is displayed.
3. Once the success message is closed and the screen has transitioned, the default next steps display for the samples.

About the Code

Once the script has processed the input and ensured that all the required information is available, we can start to process the samples to determine their next steps.

1. First, we retrieve the next actions list:

```
// Retrieve the current protocol step String nextActionsURI = stepURI + '/actions' Node
nextActionsList = GLSRestApiUtils.httpGET(nextActionsURI, username, password) String
currentProtocolStepURI = nextActionsList.'configuration'[0].@uri Node currentProtocolStep =
GLSRestApiUtils.httpGET(currentProtocolStepURI, username, password)
```

2. This endpoint contains a list of the step's output analytes, and a link to its parent step configuration. In this case, we want to retrieve the step configuration so that we can collect the URIs of the expected next steps.

```
// Determine the uris of the possible next steps
currentProtocolStep.'transitions'.'transition'.each { if(NEXT_STEPS.containsKey(it.@name))
{ NEXT_STEPS[it.@name] = it.@"next-step-uri" } }
```

3. Once we have retrieved the step configuration, we iterate over its possible next steps, gathering their URIs and storing them by name in a **Map**.
4. Once we have collected the URIs of our destination steps, we can start analyzing each sample to determine what its default should be.
 - For each possible 'next-action', we retrieve the target artifact, which then enables us to retrieve that artifact's parent sample.
 - We then retrieve the value of the sample's **Pooling** UDF / custom field , if it exists. If it doesn't exist, a default value is given.

```
// For each output analyte, set its corresponding next step according to the value of the
UDF 'Pooling' nextActionsList.'next-actions'.'next-action'.each { Node artifact =
GLSRestApiUtils.httpGET(it.@"artifact-uri", username, password) Node sample =
GLSRestApiUtils.httpGET(artifact.'sample'[0].@uri, username, password) String poolingValue =
sample.'udf:field'.find { UDF_NAME == it.@name } ? sample.'udf:field'.find { UDF_NAME ==
it.@name }.value()[0] : 'Default' // If Pooling is a variation of No or None, set to
workflowAssignment, otherwise set to pooling step if(!NO_VALUES.contains
```

```
(poolingValue.toLowerCase())) { it.@action = NEXT_STEP_ACTION it.@'step-uri' = NEXT_STEPS
[POOLING_STEP] poolingSamples++ } else { it.@action = NEXT_STEP_ACTION it.@'step-uri' =
NEXT_STEPS[WORKFLOW_STEP] workflowAssignmentSamples++ } }
```

- To set the next step, we set the **step-uri** attribute of the node to the URI of the expected destination step.
- We also increment counters, so that we can report to the user what actions were taken on the given samples.
- Once this is done, we perform an **httpPUT** on the action list, adding the changes to the API and allowing our defaults to be set.
- Finally, we define the successful output message to the user. This allows the user to check the results.

```
// Update the next steps in the API GLSRestApiUtils.httpPUT(nextActionsList, nextActionsURI,
username, password) // Define the success message to the user outputMessage = "Script has
completed successfully.${LINE_TERMINATOR}" + "Next steps for ${poolingSamples +
workflowAssignmentSamples} samples have been set:${LINE_TERMINATOR}" + "${poolingSamples}
samples set to '${POOLING_STEP}'.${LINE_TERMINATOR}" + "${workflowAssignmentSamples} samples
set to '${WORKFLOW_STEP}'."
```

Assumptions and Notes

- You are running a version of Groovy that is supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The attached Groovy file is placed on the LIMS server, in the folder **/opt/gls/clarity/customextensions**
- **GLSRestApiUtils.groovy** is placed in your Groovy lib folder.
- A single-line text sample UDF named **Pooling** has been created.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[NextStepAutomation.groovy.txt](#)

Parsing Metadata into UDFs (BCL Conversion and Demultiplexing)

This example provides a script that can be used to parse lanebarcode.html files from demultiplexing. This script is written to be easily used with the out of the box **Bcl Conversion and Demultiplexing (HiSeq 3000/4000)** protocol.

- Result values are associated with a barcode sequence as well as lane.
- Values are attached to the result file output in BaseSpace Clarity LIMS, with matching barcode sequence (index on derived sample input) and lane (container placement of derived sample input).
- Script modifications may be needed to match the format of index in Clarity LIMS to the index in the HTML result file.

Parameters

The script accepts the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-o	The limsid of the result file artifact with attached lanebarcode.html file (Required)
-s	The LIMS IDs of the individual result files. (Required)

An example of the full syntax to invoke the script is as follows:

```
bash -l -c "/usr/bin/python /opt/gls/clarity/customextensions/demux_stats_parser.py -s {stepURI:v2} -o {compoundOutputFileLuid0} -u {username} -p {password}"
```

Configuration

Defining the UDFs/custom fields

All user defined fields (UDFs) / custom fields must first be defined in the script. Within the UDF / custom field dictionary, the name of the field as it appears in Clarity LIMS (the key) must be associated with the field from the result file (the value).

The fields should be preconfigured in Clarity LIMS for result file outputs.

```
udfs_in_clarity = {"Yield PF (Gb)": "Yield (Mbases)" "%PF": "% PF Clusters", "% One Mismatch Reads (Index)": "% One mismatch barcode", "% Bases >=Q30": "% andgt;= Q30 bases", "Ave Q Score": "Mean Quality Score", "% Perfect Index Read": "% Perfect barcode", "# Reads": "PF Clusters", "% of Raw Clusters Per Lane": "% of the lane"}
```

Modifying individual UDFs/custom fields

The UDF / custom field values can be modified before being brought into Clarity LIMS. In the following example, the value in megabases is modified to gigabases.

```
if clarity_udf == 'Yield PF (Gb)': yieldmb = udf_value yieldmb = yieldmb.replace(",", "")  
yieldgb = float(yieldmb)*.001 udf_value = yieldgb
```

Checking for matching flow cell ID

The script currently checks the flow cell ID for the projects in Clarity LIMS against the flow cell IS in the result file.

 The script will still complete and attach UDF / custom field values. You may wish to modify the script to not attach the field values if the flow cell ID does not match.

Assumptions and Notes

- Your configuration conforms with the script's requirements, as documented in the Configuration section of this document.
- You are running a version of Python that is supported by Clarity LIMS, as documented in the [Technical Requirements](#).

- The attached Python file is placed on the LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The glsapiutil file is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[demux_stats_parser.py.txt](#)
[demux_stats_parser_4.py.txt](#)

Scripts That Validate Step Contents

- [Validating Process/Step Level UDFs](#)
- [Checking That Containers Are Named Appropriately](#)
- [Checking for Index Clashes Based on Index Sequence](#)
- [Validating Illumina TruSeq Index Adapter Combinations](#)

Validating Process/Step Level UDFs

In this example, the protocol step is configured to invoke a script that is triggered when the user exits the step's **Record Details** screen.

Parameters

The EPP command is configured to pass the following parameters to the script:

-u	The username of the current user (Required)	
-p	The password of the current user (Required)	
-f	The names(s) of the UDF(s) that should be considered mandatory	Multiple UDF names should be separated by a comma

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/checkprocessFields.py -l 122-7953 -u admin -p securepassword -f 'field Name 1, field Name 2, field Name N'
```

User Interaction

When the lab scientist attempts to exit the **Record Details** screen, the script is invoked and the UDF names specified in the -f parameter will be checked to see that they have been populated. If all the specified UDFs have been populated as expected the script issues no message, and the process will continue as normal. If some of the specified UDFs have not been populated a message will be displayed to the user indicating which mandatory fields need to be populated. The user will be unable to proceed until all the mandatory fields have been populated.

About the Code

The method of central interest in the script is **checkFields()**. The method in turn carries out several operations:

1. The list of fields passed to the script *via* the -f parameter is broken into its component UDF names
2. For each UDF name in the list, the corresponding value is checked via the API, and if the value is empty, the UDF name is noted as absent
3. If any UDF names are noted as being absent, an error dialog will be displayed to the user.

Assumptions and Notes

- Both of the attached files are placed on the LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The HOSTNAME global variable needs to be updated so that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.
- If required, this script could be enhanced so that it not only checked to see that the UDFs had been populated, but that their values also match a regexp pattern for additional validation.

Attachments

[checkprocessFields.py.txt](#)

Checking That Containers Are Named Appropriately

When samples are placed onto containers in BaseSpace Clarity LIMS, the default operation is that containers produced by the system are named with their LIMS ID (for example 27-1234 etc). The expectation is that after samples have been placed, the user will rename the containers with the barcode that is on the plate, chip, or other container.

There are places in the workflow in which issues will occur with integrations if the user renames the container incorrectly.

For example, when loading libraries onto a flow cell for sequencing, the flow cell container must be renamed with the actual ID/barcode of the flow cell in order for the Illumina integration to complete its work successfully.

This example provides a short script that checks to see if the user has correctly renamed containers after the placement of samples. It also shows what occurs in the Clarity LIMS Web Interface when the script encounters a naming issue.

Solution

Protocol step configuration

In this example, the protocol step is configured to invoke the script when the user exits the step's Sample Placement screen.

Parameters

The EPP command is configured to pass the following parameters:

-l	The lmsid of the process invoking the script (Required)
-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script (Required)

An example of the full syntax to invoke the script is as follows:

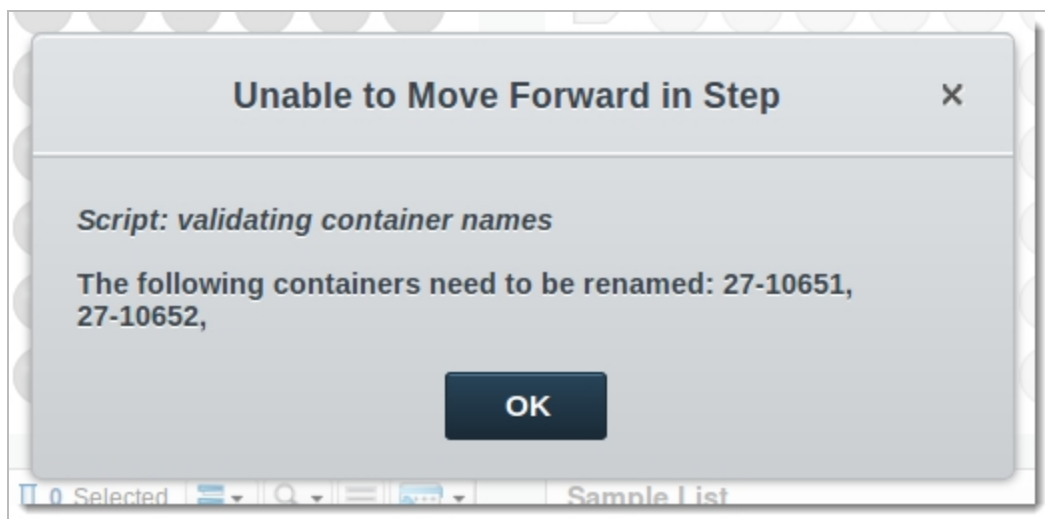
```
/usr/bin/python /opt/gls/clarity/customextensions/validateContainerNames.py -l 122-7953 -u admin -p securepassword -s http://192.168.9.123:8080/api/v2/steps/122-5601
```

User Interaction

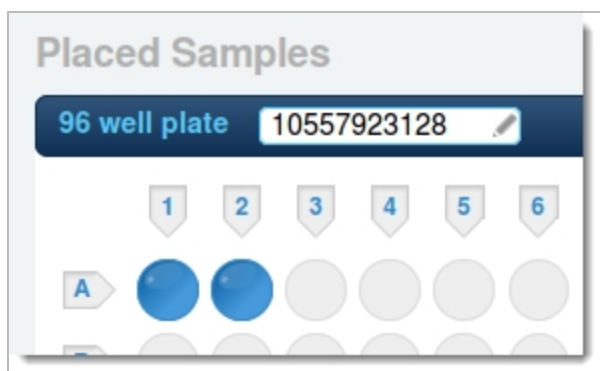
When the user enters the Sample Placement screen, the rightmost **Placed Samples** area will show the containers created with their default, system-assigned names:

When the user tries to leave the Sample Placement screen, the script is invoked:

If the script finds any containers that still have their default, system-assigned names, an error message is generated:



To complete the protocol step, the user must first rename the containers:



About the Code

The main method of interest is **validateContainerNames()**.

1. This method queries the **placements** resource for the current step, and gathers the LIMS IDs of the selected containers.
2. For each selected container associated with the protocol step, the **containers** resource is called:
 - The container name is compared to the LIMS ID.
 - If the values are identical, an error message is generated.
3. **Additional validation:** Ideally, this script is set up to also validate the renamed containers against a specific goal. For example, the attached example script checks to see that the new name is at least 10 characters in length. You may choose to replace or supplement this optional, additional validation to provide specific logic to suit your business needs.

```
if cName == cLUID: containers.append( cName ) else: ## optional, additional validation here
if len(cName) < 10: containers.append( cName )
```

Assumptions and Notes

- Both of the attached files are placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- You will need to update the HOSTNAME global variable such that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[validateContainerNames.py.txt](#)

Checking for Index Clashes Based on Index Sequence

While not a common scenario, it is possible to have two molecular barcodes/indices in BaseSpace Clarity LIMS that share the same sequence.

Within Clarity LIMS, there is logic that can be enabled to prevent pooling of samples with the same index name. However, this check is based upon the **index name**, not the sequence. In the case of identical indexes with differing names, additional logic is required to prevent index clashes based upon the **index sequence**.

This example provides a solution that looks for index clashes based upon sequence.

Solution

In this example, the protocol step that is responsible for pooling is configured to invoke the script as soon as the user exits the step's Pooling screen.

Parameters

The EPP command is configured to pass the following parameters:

-l	The limsid of the process invoking the code (Required)	The {processLuid} token
-u	The username of the current user (Required)	The {username} token
-p	The password of the current user (Required)	The {password} token
-s	The URI of the step that launches the script (Required)	The {stepURI:v2:http} token

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/checkIndexes.py -l 122-7953 -u admin -p  
securepassword -s http://192.168.9.123:8080/api/v2/steps/122-5601
```

User Interaction

When the user exits the Pooling screen, the script is invoked.

- If no index clashes are found , the step proceeds and the script does not report back to the user.

- If an index clash is found, a message is returned to the user, informing them of the components causing the problem within the affected pool. The user is not able to continue the step without first re-constituting the pools.

About the Code

The main method in the script is **checkIndexes()**. The method in turn carries out several operations:

1. The step's **pools** resource is queried.
2. The constituent artifacts of each pool are identified, along with the name of the reagent-label applied.
3. For each reagent-label:
 - The associated sequence is identified.
 - A dictionary is constructed using the sequence as key, the value being the name of the derived sample(s) associated with the sequence
4. If any sequence in the dictionary is associated with more than one derived sample, an error message is constructed and is reported to the user once all pools have been considered.

Assumptions and Notes

- Both of the attached files are placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- You will need to update the HOSTNAME global variable such that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

 | The current script uses caching to avoid unnecessary GETs, but does not use batch transactions. In the case of large levels of multiplexing, the script could be reworked to gather all of the constituent artifacts in a single batch transaction.

Attachments

[checkIndexes.py.txt](#)

Validating Illumina TruSeq Index Adapter Combinations

When pooling samples, there are often numerous complex rules and restrictions regarding which combinations of adapters are acceptable.

As a method of applying custom business logic, it is possible to automate the verification of your pools using BaseSpace Clarity LIMS.

This example shows how to confirm the composition of pools before they are created, allowing the lab scientist to alter the composition of pools that have caused an error.

Solution

In this example, we will enforce the following **Illumina TruSeq DNA LT** adapter tube pooling guidelines:

Process

The example script is configured to run on the **Library Pooling (Illumina SBS) 4.0** process.

Parameters

The EPP command is configured to pass the following parameters to the script:

-i	The URI of the step that launches the script (Required)	The {stepURI:v2:http} token - in the form <i>http://<Hostname>/api/v2/steps/<ProtocolStepLimsid></i>
-u	The username of the current user (Required)	The {username} token
-p	The password of the current user (Required)	The {password} token

An example of the full syntax to invoke the script is as follows:

```
bash -c "/opt/gls/groovy/current/bin/groovy -cp /opt/groovy/lib  
/opt/gls/clarity/customextensions/ConfirmationOfPoolComposition.groovy -i {stepURI:v2:http}  
-u {username} -p {password}"
```

 The location of Groovy on your server may be different from the one shown in this example. If this is the case, modify the script accordingly.

Protocol step configuration

In this example, in the **Illumina SBS (HiSeq GAllx)** protocol, the **Library Pooling (Illumina SBS) 4.0** protocol step is configured as follows:

- **Behavior:** Automatically initiated
- **Stage of Step:** On Pooling screen
- **Timing:** When screen is exited

Name	Behaviour	Stage of Step	Timing
Confirm Pool Compositions	Automatically initia... ▼	on Pooling screen ▼	when screen is exi... ▼

Ensure that the **Illumina TruSeq DNA LT** adapters are present and applied upstream of the **Library Pooling (Illumina SBS) 4.0** protocol step.

User Interaction

1. Assuming samples have been worked through the protocol and have reached the **Library Pooling (Illumina SBS) 4.0** protocol step, the user pools the samples following the specified guidelines.
2. When the pools are created, the user attempts to proceed to the next page.

3. A message box displays alerting the user that a custom program is executing.
4. On successful completion, a success message displays.

About the Code

Once the script has processed the input and ensured that all the required information is available, we process the pools to determine if they meet the required specifications.

1. The first challenge is to represent the adapter combinations in the script.
 - This is accomplished by a **map** comprised of the adapter names, indexed by their respective number, ie. AD001 indexed at 1.
2. Next, we define the three combination groups: 2 plex, 3 plex, and 4 plex.
 - This is achieved by creating a **List of Lists**, with the inner lists representing our combinations.
 - To facilitate the required fallback through lower plexity combinations, we store the combinations groups in a list, in ascending plexity.

```
// Reagent Adapters public static final def ADAPTERS = [ (1): 'AD001 (ATCACG)', (2): 'AD002 (CGATGT)', (3): 'AD003 (TTAGGC)', (4): 'AD004 (TGACCA)', (5): 'AD005 (ACAGTG)', (6): 'AD006 (GCCAAT)', (7): 'AD007 (CAGATC)', (8): 'AD008 (ACTTGA)', (9): 'AD009 (GATCAG)'...] // Pooling Combinations public static final def TWO_PLEX_PATTERNS = [ [ADAPTERS[6], ADAPTERS[12]], // Combination 1 [ADAPTERS[5], ADAPTERS[19]] // Combination 2 ] .... public static final def PATTERNS = [TWO_PLEX_PATTERNS, THREE_PLEX_PATTERNS, FOUR_PLEX_PATTERNS]
```

3. Once the combinations are defined, we need to create a method which will compare the actual combination of adapters in a pool with our ideal combinations. There are two cases we need to handle:
 - When we are comparing two equal plexity combinations.
 - When we are comparing a higher plexity pool to a lower plexity combination.
4. To handle the first case, we create a function that takes in our actual combination, and the ideal combination.
5. If the actual combination contains the entire combination, we remove those adapters. We then ensure that the leftover adapters are not in our Illumina TruSeq DNA LT adapter.

```
Boolean noWildcardsMatch(List disposableList, def combination) { if (disposableList.containsAll(combination)) { disposableList.removeAll(combination) // Ensure that there are no other reagent-labels in the pool that may conflict with the pattern return !disposableList.find { leftoverReagent -> ADAPTERS.containsValue(leftoverReagent) } } return false }
```

6. The second case is similar to the first.
 - We create a function that takes in our actual combination, the ideal combination, and the amount of wildcards. A **wildcard** represents an 'any adapter' condition in Illumina's TruSeq DNA LT adapter tube pooling guidelines.
 - Like the first case, we ensure that the actual list contains the entire ideal combination.

- After removing the ideal adapters, we ensure that the amount of leftover Illumina TruSeq DNA LT adapters is equal to the amount of wildcards.

```
Boolean wildCardMatch(List disposableList, def combination, int wildcards) { // If the
reagents contain the entire combination if(disposableList.containsAll(combination)) {
disposableList.removeAll(combination) // If there are not more reagents found in the pool
than there are wildcards, return true return (disposableList.findAll { leftoverReagent ->
ADAPTERS.containsValue(leftoverReagent) }.size() == wildcards) } return false }
```

7. To represent the adapter combination fallbacks, we require a method which will attempt to match the highest possible plexity for a given list of adapters. If it cannot do this, it will attempt to match it with a lower plexity combination with a wildcard.

- To achieve this, we define a recursive function that handles both the exact and wildcard cases. The ideal combination plexity will be chosen by the **patternIndex** input.
- If no wildcards are present, we check each combination in the designated plexity.
- If a match is not found, we call the **match** function again. This time, we increase the amount of wildcards by 1 and reduce the plexity of the combinations by 1. The function will now compare the adapter list using the **wildCardMatch** function. If a match is found, the function will exit and return true.

```
Boolean match(List reagents, int patternIndex, int wildcards = 0) { Boolean matches = false
// If there are wild cards, handle them separately if(wildcards == 0) { // For each
combination, check for a match PATTERNS[patternIndex].each { if(noWildcardsMatch(reagents,
it)) { matches = true } } } else { PATTERNS[patternIndex].each { if(wildCardMatch(reagents,
it, wildcards)) { matches = true } } } // If there was no match, determine if a match is
found with an easier set of restrictions if(!matches && patternIndex != 0) { matches = match
(reagents, patternIndex - 1, wildcards + 1) } return matches }
```

8. Now, with our supporting functions defined, we can start processing our pools.

- First we retrieve the definitions of the pools from the API. This node contains a list of the output pools, in addition to what input each pool contains.
- Using this list, we create a map that stores the URIs of the output pools and the amount of inputs to each pool.
- We then retrieve the output pools using a **batchGET**.

```
// Retrieve the pooling information Node pooling = GLSRestApiUtils.httpGET(stepURI +
'/pools', username, password) // Collect all of the unique potential output pools and their
number of input samples Map poolURIs = [:] pooling.'pooled-inputs'.each { String
poolURI = it.'output-uri' poolURIs[poolURI] = it.'input'.size() } // Retrieve the pool
artifacts def poolNodes = GLSRestApiUtils.batchGET(poolURIs.keySet(), username, password)
```

9. Once we have the pools, we iterate through the list.

- If a pool is valid, we increment a counter which will be used in our success message.
- If invalid, we set the script outcome to failure, and append to the failure message.

- The script continues searching for other issues and adding their information to the failure message.

```
// Verify that each pool contains acceptable reagent labels Boolean failure = false String
errorMessage = '' poolNodes.each { Boolean accepted = verifyReagents(it, poolURIs
[GLSRestApiUtils.stripQuery(it.@uri)]) if(accepted) { validatedPools++ } else { failure =
true errorMessage = errorMessage + "${it.'name'[0].text()} has an invalid combination of
Illumina Adapters.${LINE_TERMINATOR}" } }
```

10. After each pool has been checked, we determine how to alert the user of the script's completion.

- If a pool is invalid, an error is thrown containing the list of failures and a recommendation to review the Illumina pooling guidelines.
- If all pools are valid, we alert the user of a success.

```
// If a pool failed validation, report the message to the user and halt their progress if
(failure) { throw new Exception(errorMessage + 'Please consult the Illumina TruSeq adapter
pooling guidelines.') } // Define the success message to the user outputMessage = "Script
has completed successfully.${LINE_TERMINATOR}All ${validatedPools} pools passed Illumina
low-plexity pooling guidelines."
```

Assumptions and Notes

- Your configuration conforms with the script's requirements(see the Solution section of this document).
- You are running a version of Groovy that is supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The attached Groovy file is placed on the LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- **GLSRestApiUtils.groovy** is placed in your Groovy **lib** folder.
- You have imported the attached Reagent XML file into your system using the Config Slicer tool.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[ConfirmationOfPoolComposition.groovy.txt](#)

[Single Indexing ReagentTypes.xml](#)

Scripts Triggered Outside of Workflows/Steps

[Repurposing a Process to Upload Indexes](#)

[Adding Users in Bulk](#)

[Moving Reagent Kits & Lots to New Clarity LIMS Server](#)

[Programmatically Importing the Sample Submission Excel File](#)

[Assigning Samples to New Workflows](#)

Repurposing a Process to Upload Indexes

Compatibility: API version 2

Uploading indexes (reagent types) into BaseSpace Clarity LIMScan be done 3 different ways:

- 1. Manually adding the indexes one at a time using the Operations interface.
- 2. Uploading an XML data file using the config slicer.
- 3. Using the API directly.

There is no "out of the box" method to quickly and easily upload a CSV file of many indexes to the LIMS. Adding the indexes one at a time is not quick, and using the config slicer requires admin privileges, knowledge of the correct XML file format, and the command line.

Solution

This example enables Clarity LIMS lab users to upload a CSV file containing new indexes and, through an EPP trigger, instantly create the indexes in Clarity LIMS. This example provides the provides the functionality included with the config slice, including preventing the indexes from being created if there is already an index with the same name in the system as well as ensuring the sequence only contains a valid nucleotide sequence (composed of A,G,T and C).

Protocol Step Configuration

In this example, a protocol step is repurposed as a mechanism to upload indexes. The protocol is configured to not produce any analyte outputs, and a single shared output file is used as the placeholder to upload the CSV file containing the indexes. The step is configured to permit use of a control sample, which allows the user to begin the step without any sample inputs.

Parameters

The script accepts the following parameters:

-f	The LUID of the file placeholder for the CSV file (Required)
-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script - the {stepURI:v2} token (Required)
-l	The LUID of the file placeholder for the Log file (Required)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/addindexescsv.py -u {username} -p {password} -s {stepURI:v2} -f {compoundOutputFileLuid0} -l {compoundOutputFileLuid1}
```

User Interaction

The lab scientist begins the step with only a control in the ice bucket. The user uploads the CSV file to the placeholder on the record details screen, and pushes the button to trigger the script.

About the Code

The main method in the script is `importIndexes()`. This method first calls the function `downloadfile()` which finds the location of the CSV file on the Clarity LIMS server and reads the content into memory.

The script then generates an XML payload for each index, checks the nucleotide sequence against the acceptable characters, and POSTs to the `/api/v2/reagenttypes` endpoint. If an index with the same name is already present in the LIMS, the log will be appended with the exception.

This example script produces a log which tracks which indexes were not added to the system as well as the reason. The log displays how many indexes were successfully added to the system and the total number of indexes included in the CSV file.

Assumptions and Notes

- You are running a version of Python that is supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The attached files are placed on the LIMS server, in the `/opt/gls/clarity/customextensions` folder.
- You will need to update the `HOSTNAME` global variable such that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[addindexescsv.py.txt](#)

[example_Indexes.csv.txt](#)

Adding Users in Bulk

When BaseSpace Clarity LIMS is replacing an existing LIMS, or if the LDAP integration is not being utilized, a list of current LIMS users and collaborators often already exists in a file.

In this scenario, it is useful to use this file to create users in Clarity LIMS. This topic provides an example script and outlines a strategy for the bulk import of users from a simple CSV file.

Solution

The attached script parses a CSV file containing user information. Based on the columns of data present in the file, the script then creates users and their associated labs in Clarity LIMS.

Parameters

Since this operation is independent of workflow and sample processing, the parameters used to invoke the script differ from those typically used:

-h	The URI of the server to which you want to upload the users (Required)
-u	The username of a user with administrative privileges (Required)
-p	The password of the user (Required)
-f	The path to the CSV file containing the user-related data (Required)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/userImport.py -h http://192.168.8.10:8080  
-u admin -p securepassword -f ./user_list.csv
```

File Format

The format of the file is very flexible. The order of the columns is not relevant.

If the names of your columns do not match the column names in the example file, modify the script so that the column names match yours.

```
uNameIndex = COLS[ "User" ] fnameIndex = COLS[ "First Name" ] lNameIndex = COLS[ "Last Name" ]  
eMailIndex = COLS[ "E-mail" ] labIndex = COLS[ "Institution" ]
```

Attached to this topic, you'll find a CSV file containing test data that illustrates the format further. The structure of this file is shown in this example:

	A	B	C	D	E	F	G	H	
1	Generated by Geospiza GeneSifter Lab Edition								
2									
3	User	Active	First Name	Last Name	E-mail	Institution	Role	Groups	Lab C
4	abroccoli	YES	Albert	Broccoli	abroccoli@test.com		Collaborator		
5	cdusthoover	YES	Cedric	Dusthoover	cdusthoover@test.com	Simpkins Lab	Collaborator		
6	efreezing	YES	Ethel	Freezing	efreezing@test.com	Freezing Cold	Collaborator		
7	gheartburn	YES	George	Heartburn	gheartburn@test.com	Coronary Lab	Collaborator		
8	ijalopy	YES	Ivor	Jalopy	ijalopy@test.com		Collaborator		
9									

Notice that the first two rows do not contain data of interest, and so the script ignores them. This is controlled by line 38 of the script, which specifies the location of the header row:

```
headerRow = 3
```

About the Code

The main method of interest is **importData()**. After parsing the file into data structures (COLS and DATA), the data is processed one line at a time using the following pseudocode:

```
does a user with this username already exist?if NO: does the lab exist? if NO: createLab
createUser
```

To reduce the number of queries the script makes back to the server, each time a new lab is created it is added to the cache of existing labs that is stored in the **LABS** dictionary.

Finally, if you set the DEBUG variable to **true**, the script will stop processing the file after the first successful creation of a user. This is useful as it allows you to test your script using just one line of the CSV file at a time.

Assumptions and Notes

- Both of the attached *.py script files are placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The users will be created with the same hard-coded password (abcd1234). It is possible to have the script create a unique password for each user. If this is required, consider having the script e-mail the user with a 'welcome' e-mail outlining their username and password.
- The users will be created with permissions to access the **Collaborations Interface** only. This can be modified as required.
- The contents of the ' **Institution** ' column will be used to associate the user with a ' **Lab** ', if a value for this column is provided.
- The example code provided is for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[user_List_Test.csv](#)

[userImport.py.txt](#)

Moving Reagent Kits & Lots to New Clarity LIMS Server

Whereas config slicer is a useful tool for moving configuration data from one server to another and, strictly speaking, reagent lot data is not considered configuration, there is a need for an easy way to transfer reagent lot data to a new BaseSpace Clarity LIMSserver.

Solution

The preventative solution is to not create production reagent kits on the development server. However, if you're reading this it might be too late for that.

This example demonstrates how to use the Clarity LIMS API to extract reagent kit and reagent lot data from one server, transfer the data as an XML file and create the equivalent reagent kits and/or lots on the new server.

This example contains two scripts which can be run on the CLI, one which exports data from the dev server and generates a txt file. The second which uses the exported data to create the kits/ lots on the prod server.

Parameters

Export

The first script accepts the following parameters:

-s	The hostname of the Clarity LIMS server API and version from which to export (Required)
-u	The username of the current user (Required)
-p	The password of the current user (Required)
-d	The directory to write the txt file to (Optional, default writes to current working directory)
--skipLots	The script does not export reagent lot data (Optional)
--skipKits	The script does not export reagent kit data (Optional)
-k	A list of reagent kits to limit the export to, only reagent lots for these kits will be exported (Optional, by default will export all reagent kits)

An example of the full syntax to invoke the first script is as follows:

```
python reagents_export.py -s https://demo-4-1.claritylims.com/api/v2 -u apiuser -p ***** -d /Users/dcrawford/Documents --skipKits -k "AMPure XP Beads, Covaris Snap-cap microTUBE"
```

Import

The second script accepts the following parameters:

-s	The hostname of the Clarity LIMS server API and version to import into (Required)
-u	The username of the current user (Required)
-p	The password of the current user (Required)
-d	The directory to write the txt file from (Optional, default expects file to be in current working directory)
--checkforKits	The script checks the clarity server for existing kits of the same name. (Optional, Recommended) If this parameter is used, the script does not create duplicate reagent kits.

An example of the full syntax to invoke the first script is as follows:

```
python reagents_import.py -s https://demo-4-1.claritylims.com/api/v2 -u apiuser -p ***** -d /Users/dcrawford/Documents --checkforKits
```

reagents_export.py

The main method in the script searches the reagent kits and reagent lots endpoints for all the kits and lots or, if the -k parameter is used, for the kits and lots belonging to kits specified in the -k parameter.

The script writes the XML data for each kit and lot to the XML data file.

reagents_import.py

The main method of this script creates the reagent kits and reagent lots in CClarity LIMS. In the case of duplicate reagent kits, the reagent lots are associated with the newly created kits. If the `--checkforKits` parameter is included, the script does not create kits with duplicate names, and associates the reagent lots with the preexisting kits with the matching reagent lot name.

Assumptions & Notes

- You are running a version of Python supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

- Script Updates

Jan 2018: reagents_export_jan2018.py and reagents_import_jan2018.py

- Script updated to enable import/export of reagent kits with special characters.

Aug 2018: reagents_import_Aug2018.py

- Script checks for preexisting reagent lots.
- `--checkforkits` prevents duplicate reagent lots. Lots with non-matching status are considered to already exist (if duplicate kit, name & number).

Attachments

[reagents_import.py.txt](#)

[reagents_export.py.txt](#)

[reagents_import_jan2018.py.txt](#)

[reagents_export_jan2018.py.txt](#)

[reagents_import_Aug2018.py.txt](#)

Programatically Importing the Sample Submission Excel File

Compatibility: API version 2

The BaseSpace Clarity LIMSweb interface provides an example Sample Import Excel file which can be manually uploaded to submit new samples to a selected project within the LIMS.

This application example shows how the Sample Import Sheet can be uploaded programatically with just one change to its format.

Solution

This example uses the same Sample Sheet with an additional column 'Project/Name'. The script processes the Sample Sheet, creates the samples and adds the samples to their respective projects.

This script leverages a python module xlrd, which is not included in the standard python library. It is used to extract data from .xls and .xlsx excel files.

Parameters

The script accepts the following parameters:

-f	The full path to the location of the excel file. (Required)
-g	The full path to the location of the log file. (Optional)

An example of the full syntax to invoke the script is as follows:

```
python /path/to/file/SampleSheetImporter.py -f /Users/mywd/ClaritySampleSheetprojects.xlsx -g /Users/logs/samplesubmission.log
```

About the Code

parseFile

This method carries out several operations:

- Opens the excel file and reads the text
- Stores the column headers in a dictionary variable called COLS
- Stores the row data accessibly in an array variable called ROWS

createProject

This method in turn carries out several operations:

- For each project name the script encounters it searches the LIMS to identify if a project with this name has already been created.
- If the project does not exist, the script will create the project. This example script is easily modifiable, however as written:
 - Project researcher is System Administrator
 - Project open date is today
 - No project level UDFs are created

processRows

This method prepares the data needed to create a sample in LIMS:

- Assembles the UDF values, the project ID and container ID.
- For each non-tube container the script encounters it searches the LIMS to identify if a container with this name already exists.
- If the container does not exist the script will create the container.
 - If container type is not specified, TUBE will be assumed.
 - For TUBE, well location will always be 1:1

The script contains additional supporting methods to generate XML which is POSTED to the API.

Assumptions and Notes

- The UDFs in the Sample Sheet header have been configured in LIMS prior to submission.
- The following column headers are required: Sample/Name, Project/Name and any sample-level UDFs that are mandatory within your system.
- The script need not be run on the Clarity server, however it must have a connection to the Clarity LIMS API.
- You are using Python version 2.6 or 2.7.
- The Python installation contains the **non-standard** xlrd library.
- The _auth_tokens.py file has been updated to include the information for your installation.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[_auth_tokens.py.txt](#)

[ClaritySampleSheetprojects.xlsx](#)

[SampleSheetImporter.py.txt](#)

Generating an MS Excel Sample Submission Spreadsheet

An easy way to get sample data into BaseSpace Clarity LIMS is to import a Microsoft® Excel® sample spreadsheet. However, manually configuring the spreadsheet can be time-consuming and error-prone.

Use this Python application example to generate a custom Excel sample submission spreadsheet that meets the specific needs of your lab.

Solution

Parameters

The script accepts the following parameters:

-a	The base URL to the REST API - no trailing slash (Required)	The URL that points to your main API endpoint
-u	The username of the admin user (Required)	The {username} token
-p	The password of the admin user (Required)	The {password} token

The **SampleSubmissionXLSGenerator.py** application script is run as follows:

```
python SampleSubmissionXLSGenerator.py -a http://<hostname or IP address of server>:<port>/api/v2 -u admin_user -p admin_user_pass
```

An XLS workbook containing the following worksheets is generated:

- **Sample Import Sheet:** Contains columns for all Sample UDFs/UDTs.
- **Container Type Names:** Contains a list of all container type names.

User Interaction

1. In the Clarity LIMS **Operations Interface**, the lab manager configures **Container Types** and **Sample UDFs/UDTs**.
2. In the Clarity LIMS **Web Interface**, the lab manager or lab scientist runs the **SampleSubmissionXLSGenerator.py** application script, providing the required parameters.
3. An Excel workbook containing the **Sample Import Sheet** and **Container Type Names** worksheets is generated.
4. The **Sample Import Sheet** is provided to lab scientists for use when creating lists of samples to import; it may also be used by collaborators to import samples via the LabLink **Collaborations Interface**.

Using the Generated Spreadsheet

1. The spreadsheet will contain red and green column headers. Populate the spreadsheet with sample data:
 - **Red headers:** These columns must contain data.
 - **Green headers:** These columns may be left empty.
2. Import the spreadsheet into Clarity LIMS.

 | If there are no sample user-defined fields (UDFs) or user-defined types (UDTs) in the system, the generated spreadsheet will only contain four columns. After configuring the UDFs/UDTs, you can re-run the script to add columns to the spreadsheet that reflect the updated configuration.

Example Modifications

You can edit the Python application to include supplementary functions. For example, you may want to use other attributes from the resulting XML to generate additional data entry columns.

Inserting Additional Non-required and Non-configured Columns

- The **SampleSubmissionXLSGenerator.py** Python application script adds **Sample/Volume** and **Sample/Concentration** columns to the spreadsheet.
- The script includes a 'commented' section. You can remove the **### NON-REQUIRED COLUMN MODIFICATION** **###** comments and use this section to add your own columns.

Assumptions and Notes

- You have downloaded the attached zip file to the server. On a non-production server use the **glsai** user account.
- Unzip the file to the following directory: **/opt/glsclarity/customextensions**. The contents of the zip file will be installed within that directory, to **/CookBook/SpreadSheetGenerator/**
- Python 2.7 is installed and configured on the system path.

- You can run the **SampleSubmissionXLSGenerator.py** Python application script on any system running Python, provided it has web access to the Clarity LIMS REST API.
- This script is not compatible with the 3.x branch of Python.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

All dependencies are preloaded if you install on a non-production server.

Attachments

[python-xls-generator-2.0-bundle.zip](#)

Assigning Samples to New Workflows

It is sometimes necessary to assign a sample to a new workflow from within another workflow.

You can do this in the BaseSpace Clarity LIMS **Operations Interface**, by manually adding the sample to the desired workflow. However, it is also possible to perform this action through the API.

Solution

This example shows how to use the API to automate the addition of samples to a specified workflow, based on a UDF value.

- The script can be run off any protocol step whose underlying process is configured with **Analyte inputs** and a **single per-input ResultFile output**.
- The process may have any number of per-all-input ResultFile outputs, as they will be ignored by the script.
- A Result File UDF, named **Validate** , will control which samples will be added to the specified workflow.

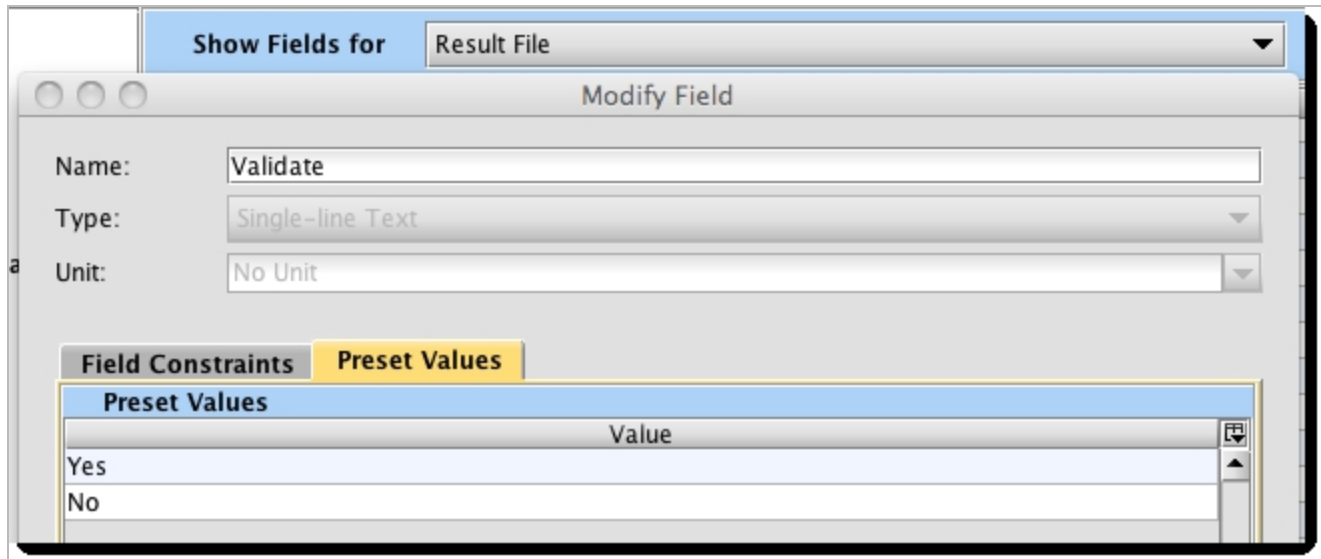
Parameters

The script accepts the following parameters:

-i	The limsid of the process invoking the script (Required)	The {processLuid} token
-s	The URI of the step that launches the script (Required)	The {stepURI:v2:http} token (in the form http://<YourIP>/api/v2/steps/<ProtocolStepLimsid>)
-u	The username of the current user (Required)	The {username} token
-p	The password of the current user (Required)	The {password} token
-w	The name of the destination workflow (Required)	The {workflow} token

Step 1: Create Result File UDF

Before the example script can be used, first create the ResultFile's **Validate** UDF in the **Operations Interface**. This is a single-line text UDF with preset values of 'Yes' and 'No'.



The screenshot shows the 'Modify Field' dialog box. At the top, there's a 'Show Fields for' dropdown menu set to 'Result File'. Below this, the 'Name' field contains 'Validate', the 'Type' dropdown is set to 'Single-line Text', and the 'Unit' dropdown is set to 'No Unit'. There are two tabs: 'Field Constraints' and 'Preset Values', with the latter being selected. The 'Preset Values' tab displays a table with a single column header 'Value' and two rows of values: 'Yes' and 'No'.

Value
Yes
No

Step 2: Create and Configure Process Type

In the **Operations Interface**, create a new process type named **Cookbook Workflow Addition**.

This process type must:

- Have Analyte inputs.
- Have a single per-input ResultFile output.
- Apply the **Validate** UDF on its ResultFile outputs.

Step 3: Configure an EPP call on this process type as follows:

```
bash -c "/opt/gls/groovy/current/bin/groovy -cp /opt/groovy/lib  
/opt/gls/clarity/customextensions/SwitchingWorkflows.groovy -u {username} -p {password} -s  
{stepURI:v2:http} -i {processURI:v2:http} -w 'Destination Workflow'"
```

Step 4: Modify the file paths to suit your server's Groovy installation.

Step 5: Create Protocol

Once the process type is created, in the Clarity LIMSWeb Interface, create a protocol named **Cookbook Workflow Addition Protocol**.

This protocol should have one the protocol step - **Cookbook Workflow Addition**.

Cookbook Workflow Addition Protocol Library Prep Capacity 0 ☐ Hidden

Select a Step to Add to your Protocol Add Step +

1 Cookbook Workflow Addition

Step 6: Configure EPP

Configure the EPP script to automatically initiate at the end of the **Cookbook Workflow Addition** step:

Configure timing of External Program Plugin (EPP) triggering

Name	Behaviour	Stage of Step	Timing
Automated Workflow Addition	Automatically initia...	at end of step	

Step 7: Create Workflows

To finish configuration, create two workflows:

- **Destination Workflow:** This workflow should contain the **DNA Initial QC** protocol only.
- **Sending Workflow:** This workflow should contain the new **Cookbook Workflow Addition Protocol**.

Sending Workflow Archive

Workflow Status: **Active**

1 Cookbook Workflow Addition Prot...

Destination Workflow Archive

Workflow Status: **Active**

1 DNA Initial QC

About the Code

Once the script has processed the input parameters and ensured that all the required information is available, we can start processing the samples to determine if they should be assigned to the new workflow.

1. To begin, we retrieve the process from the API. This gives us access to the input-output maps of the process. These will be used to determine which ResultFiles we will examine.
2. Next, we retrieve the protocol step action list. This contains a list of the input analytes' URIs and their next steps.
3. We then search this list for and collect all analyte URIs whose next action has been set to **Mark as protocol complete**.

```
// Retrieve the process def process = GLSRestApiUtils.httpGET(processURI, username,
password) // Retrieve the analytes which have been set to complete def actionsList =
GLSRestApiUtils.httpGET(stepURI + '/actions', username, password) def completedAnalyteURIs =
actionsList.'next-actions'.'next-action'.findAll { it.@action == COMPLETE }.collect {
it.'artifact-uri' }
```

4. Next, we gather the per-input ResultFile input-output maps. We can collect the ResultFile URIs of those related to the analytes that have been marked as complete.

 It is important that any extra state information is stripped from the URIs. The URIs found in the next action list do not contain any state information and, when compared against a non-stripped URI, will return 'false'.

5. Once we have the ResultFile URIs, we can retrieve them with batchGET. It is important that the list contains unique URIs, as the batchGET will fail otherwise.

```
// Gather the PerInput ResultFile mappings def inputOutputMaps = process.'input-output-
map'.findAll { it.'output'[0].@'output-generation-type' == PER_INPUT } // Determine which
ResultFiles to examine and retrieve them def resultFilesURIs = inputOutputMaps.findAll {
completedAnalyteURIs.contains(GLSRestApiUtils.stripQuery(it.'input'[0].@uri)) }.collect {
it.'output'[0].@uri }.unique() def resultFiles = GLSRestApiUtils.batchGET(resultFilesURIs,
username, password)
```

6. After we have retrieved the ResultFiles, we can iterate through the list, adding the parent sample's URI to our list of sample URIs if the ResultFile's **Validate** UDF is set to **Yes**. We also increment a counter which will allow us to report to the user how many samples were assigned to the new workflow.

```
// Determine which artifacts should be assigned List sampleURIs = [] resultFiles.each { if
(it.'udf:field'.find { VALIDATION_UDF == it.@name }.value()[0] == YES) { sampleURIs.add
(it.'sample'[0].@uri) switched++ } }
```

7. Since we don't assign samples themselves to workflows, we first need to retrieve the samples' derived artifacts. We can do this by iterating through each sample URI, retrieving it, and adding its artifact's URI to a list.

```
// Gather the sample artifacts URIs List artifactsToAssignURIs = [] sampleURIs.each { Node
sample = GLSRestApiUtils.httpGET(it, username, password) artifactsToAssignURIs.add
(sample.'artifact'[0].@uri) }
```

8. Before we can add the artifacts to the workflow, we need to determine the destination workflow's URI. By retrieving a list of all the workflows in the system, we can find the one that matches our input workflow name.

```
// Retrieve workflow URI def workflowList = GLSRestApiUtils.httpGET(baseURI +
'/configuration/workflows', username, password) String workflowURI = workflowList.find {
it.@name == workflow }.@uri
```

9. Assigning artifacts to workflows requires the posting of a routing command to the routing endpoint.

- We first generate the required XML by using a Streaming Markup Builder.
- We then dynamically build our XML by looping inside of the markup declaration.

```
// Create a new routing assignment using the Markup Builder def assignmentOrder =
builder.bind { mkp.xmlDeclaration() mkp.declareNamespace(rt:
'http://genologics.com/ri/routing') 'rt:routing' { if(artifactURIsToNewWorkflow.size() != 0)
{ 'assign'('workflow-uri': workflowURI) { artifactURIsToNewWorkflow.each { 'artifact'(uri:
it) } } } } } return GLSRestApiUtils.xmlStringToNode(assignmentOrder.toString())
```

10. To create our routing command, we pass the workflow URI and the artifact URIs that we wish to assign to the workflow to a method containing the above code. This will generate the required node.

11. We then perform an httpPOST to the routing endpoint to perform the action.

```
// Create and post the assignment Node assignmentNode = createAssignmentNode(workflowURI,
artifactsToAssignURIs) GLSRestApiUtils.httpPOST(assignmentNode, baseURI +
'/route/artifacts/', username, password)
```

12. Finally, we define our success message to the user. This will allow us to inform the user of the results of the script.

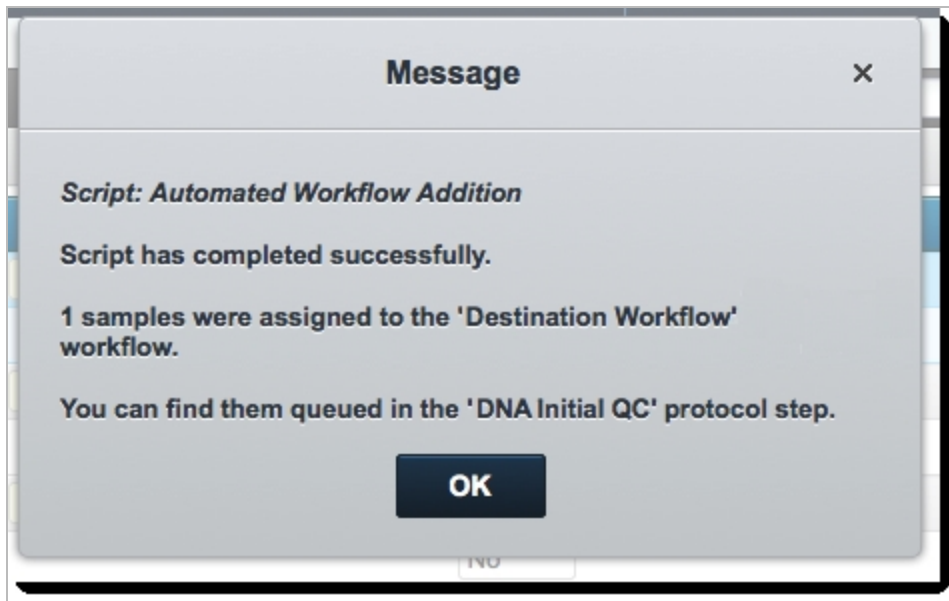
```
// Define the success message to the user outputMessage = "Script has completed
successfully.${LINE_TERMINATOR}" + "${switched} samples were assigned to the '${workflow}'
workflow.${LINE_TERMINATOR}" + "You can find them queued in the 'DNA Initial QC' protocol."
```

User Interaction

1. Assuming samples have been placed in the **Switching Workflow** , the user proceeds as normal through the protocol step.
2. In the **Record Details** screen, the user enters **Validate** values in the ResultFile UDFs.
3. The user then proceeds to the **Assign Next Steps** screen, provides a variety of **Next Steps** , and completes the protocol step.

Sample Name	*Next Step	Validate	Container Name	Well	LIMS ID (Container)	Project Name
Single Well: Tube (3 samples)						
WorkflowMoveSample-3	Mark protocol as complete	Yes	27-5953	1:1	27-5953	Dan's Test V
WorkflowMoveSample-4	Repeat this step	Yes	27-5954	1:1	27-5954	Dan's Test V
WorkflowMoveSample-5	Mark protocol as complete	No	27-5955	1:1	27-5955	Dan's Test V

4. A message displays, alerting the user of the execution of a custom script.
5. When the script completes, a success message displays and the samples are added to the specified workflow.



Assumptions and Notes

- The attached file is placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- GLSRestApiUtils.groovy is placed in the Groovy **lib** folder.
- The required configuration has been set up, as described in **Configuration**.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[SwitchingWorkflows.groovy.txt](#)

Miscellaneous Scripts

[Generating a Hierarchical Sample History](#)

[Illumina LIMS Integration](#)

[Customizing the Graphical Well-Map](#)

[Protocol-based Permissions](#)

[Self-Incremental Counters](#)

[Generic CSV Parser Template \(Python\)](#)

[Renaming Samples to Add an Internal ID](#)

[Creating Custom Sample Sheets](#)

[Copying Output UDFs to Submitted Samples](#)

- [Generating a Custom HiSeq Sample Sheet](#)
- [Parsing Sequencing Meta-Data into Clarity LIMS](#)
- [Submit to a Compute Cluster via PBS](#)
- [Downloading a File and PDF Image Extraction](#)

Illumina LIMS Integration

This combination of configuration and python script can be used to set up an integration between **BaseSpace Clarity LIMS** and **Illumina LIMS**. There are 2 main parts to this integration:

1. Generating a sample manifest from **BaseSpace Clarity LIMS** to import the samples into **Illumina LIMS**.
2. Once the analysis is completed, automatically parsing in the results from **Illumina LIMS** into **BaseSpace Clarity LIMS**.

Disclaimer: This application example is provided as is, with the assumption that anyone deploying this to their LIMS server will own the testing and customization of the configuration and scripts provided.

Installation

Protocol and Workflow Import

Using the config-slicer tool, import the configuration file attached (IlluminaLIMSIntegration.xml) as the glsjboss user with the following command:

```
java -jar /opt/gls/clarity/tools/config-slicer/config-slicer-3.<x>.jar -o import -k IlluminaLIMSIntegration.xml -u <user> -p <password> -a https://<hostname>/api
```

Script and Configuration Setup

1. As the glsjboss user on the **Basespace Clarity LIMS** server, copy the Illumina LIMS manifest template file (IlluminaLIMS_Manifest_Template.csv) attached below to the following folder:
`/opt/gls/clarity/customextensions/IlluminaLIMS`
2. On the **Illumina LIMS** Windows/Linux workstation create a folder called Clarity_gtc_Parser and do the following :
 - a. Copy the clarity_gtc_parser_v2.py file into this folder and update the following configuration parameters:

Parameter	Description
USERNAME = <APIUSERNAME>	Clarity user with API access
PASSWORD = <APIPASSWORD>	Password for that user
uri = 'https://<DEMAINNAME>/api/v2/artifacts'	URI to the artifact API endpoint on Clarity
path = '/<PATH>/IlluminaLIMS/gtc_folder_v3/'	Path to gtc files
gtcVersion = 3	gtcfile version

 | This script supports the current LIMS gtc version (3) and will be compatible with version 5 when available.

- b. Download and copy the IlluminaBeadArrayFiles.py to the same folder (also available [here](#)). Edit the file with variables: API, gtc file path and username/password for clarity API for the relevant server.
- c. Create an empty file called processed_gtc.txt in the gtc files directory.
- d. Setup a [scheduled task](#) (Windows) or [cronjob](#) (Linux) to run this python script every 10 minutes (assuming Python version 2.7.1 is installed and available on the workstation).

Workflow

The configuration attached to this page contains an example protocol with two steps.

Prerequisites:

Samples have been accessioned into **BaseSpace Clarity LIMS** with the following sample metadata as Submitted Sample UDFs:

- Is Control
- Institute Sample Label
- Species
- Sex
- Comments
- Volume (ul)
- Conc (ng/ul)
- Extraction Method
- Parent 1
- Parent 2
- Replicate(s)
- WGA Method (if applicable)
- Mass of DNA used in
- WGA
- Tissue Source

Protocol Step: IlluminaLIMS Sample Prep

This manual step is meant to be merged into the last step of a Sample Prep Protocol. It has the configuration to generate a Derived Sample with the LIMSID in the name so that the name can be unique, and used to match data back using the data parser to the next step.

Protocol Step: IlluminaLIMS Manifest and Analysis

This requires the user to perform the following steps:

1. Generate the Illumina LIMS Manifest using the button provided called "Generate Illumina LIMS Manifest".
2. Download the manifest and import this to IlluminaLIMS Project Manager under the correct institution.
3. Run the appropriate lab workflow on Illumina LIMS
4. After the Illumina LIMS analysis is complete, allow 10 minutes and come back to Clarity LIMS to find the step in progress. Make sure the following derived sample UDFs are populated:
 - Autocall Version
 - Call Rate
 - Cluster File
 - GC 10
 - GC 50
 - Gender
 - Imaging Date
 - LogR dev
 - Number of Calls
 - Number of No Calls
 - SNP Manifest
 - Sample Plate
 - Sample Well
 - 50th Percentiles in X:
 - 50th Percentiles in Y:
 - 5th Percentiles in X:
 - 5th Percentiles in Y:
 - 95th Percentiles in X:
 - 95th Percentiles in Y:
 - Number of Calls Number of Intensity Only Calls:
 - Number of No Calls

Attachments

[clarity_gtc_parser_v2.py.txt](#)

[IlluminaBeadArrayFiles.py.txt](#)

[IlluminaLIMSIntegration.xml](#)

[IlluminaLIMS_Manifest_Template.csv](#)

Generating a Hierarchical Sample History

Often, workflows do not have a linear configuration. Even in those cases, samples progressing through workflows may be re-queued, replicated, or submitted into a number of parallel workflows.

Attempts to align downstream results with submitted samples may get hindered when trying to account for sample replicates or the dynamic decisions made in the lab.

A visual representation of a samples complete history presented in a clear hierarchical format allows for a digestible report of the work done on the sample. This format provides at-a-glance understanding of the any of the branching or re-queuing of a sample.

Solution

This example describes a python script which, given an artifact, recursively finds all the processes for which that artifact was an input, then finds all the associated output artifacts of that process. This continues for all processes all the way down to the most downstream artifact.

A clear visual representation of the entire history of work on a sample, similar to what was available in the Ops interface, can allow a user to see all the processes and derivations of a sample. This is especially of use for troublesome samples that have branched into numerous downstream replicates which may end up in the same or different sequencing protocol.

Parameters

The script accepts the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The API steps URI {stepURI:v2}. (Required)

An example of the full syntax to invoke the script is as follows:

```
bash -l -c "/usr/bin/python /opt/gls/clarity/customextensions/sample_history.py -a DAN2A4PA1 -s {stepURI:v2} -u {username} -p {password}"
```

Script output example:

Sibling artifacts will appear aligned vertically with the same indentation. In the above example, 122-1650 Library Pooling (MiSeq) 5.0 created two replicate Analytes, 2-4110 and 2-4111. Analyte 2-4111 was the input to the subsequent step (24-1952) and no additional work was performed on 2-4110.

Processes performed on an artifact will appear underneath with a tab indentation. In the above example, the first 4 processes (3 QC processes and Fragment DNA) are all using the Root Analyte (CRA201A1PA1) as an input.

Adding Colors With termcolor

Install package termcolor for color printing support. Entity colors can be configured within the script. Globally turn off colors by changing the variable use_colours to False (line 16).

Assumptions and Notes

- Your configuration conforms with the script's requirements, as documented in [Solution](#).
- You are running a version of Python supported by BaseSpace Clarity LIMS, as documented in the [Technical Requirements](#).
- The glsapiutil.py file is placed in the working directory.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[sample_history.py.txt](#)

[sample_history_colours.py.txt](#)

Protocol-based Permissions

Laboratories may want to limit what steps Researchers can start. At the time of writing, BaseSpace Clarity LIMS does not natively support protocol-based permissions. However, with an EPP at the beginning of the step we can check to see if the technician/researcher starting the step has been given approval to start the step, and halt the step from starting if they do not have permission. There are several ways this can be done, but special considerations to how these permissions are administered need to be made.

Solution

In order to allow an administrator to easily maintain permissions we will assign users to groups in a config file and our EPP will consume this information. One parameter of the EPP is the groups that are permitted to run the step. When the script is triggered at the start of the step, it will look for the name of the technician starting the step in the config file and determine if the technician is:

- Included in the config file and,
- Has been assigned to a group that is permitted to run the step

It is important to remember that by exiting a script with a negative number an EPP will fail and the user will not be able to move forward in the step. We will take advantage of this EPP feature and if the technician/researcher is part of a permitted group the step would start as expected. But, if they are not part of a permitted group, entry into the step will be halted and an error box will appear with whatever the last print message was in the script.

```
try: # is the technicians name a key in the dictionary created from the config file # if so
find the groups the technician has been assigned in the config config_groups = (configDict
[first,last]).split(",") step_approved = [y.strip() for y in (args["groups"].split(","))] if
```

```
bool(set(config_groups) and set(step_approved)) is False: #fail script, stop user from
moving forward and have the last print statement appear in message print "Nice try %s %s,
but you have not been approved to run this step % (first, last ) exit (-1)except: print
"This technician's name has not been included in the config file " exit (-1
```

Parameters

The EPP command is configured to pass the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script - the {stepURI:v2:http} token (Required)
-g	The name of the permitted groups. The permitted groups should be separated by a commas, and passed as one string (enclosed in double quotes)

An example of the full syntax to invoke the script is as follows:

```
python /opt/gls/clarity/customextensions/Group_Permissions.py -u {username} -p {password} -s
{stepURI:v2} -g "GroupD, GroupE"
```

User interaction

- The config file can reside in any directory that the EPP script will have access to.
- The config file that is used in this example has tab delimited columns of Last Name, First Name, and Groups. The permitted groups need to be separated by commas (see the attached example config file). The script can be easily modified if a different format is desire for the config file
- The EPP should be "automatically initiated" at "the beginning of the step"

If the user is not allowed to move forward a message box displays and the step is aborted.

Assumptions and Notes

- You are running a version of Python that is supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- Both of the attached files are placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[config.txt](#)
[Group_Permissions.py](#)

Self-Incremental Counters

Sequential numbers are sometimes needed for naming conventions and require self-incrementing counters be created and maintained. We do not recommend using the BaseSpace Clarity LIMSDatabase for this. However the Unix “(n)dbm” library provides an easy way to create and manage counters by creating Dbm objects that behave like mappings (dictionaries).

Solution

The way this would work is the attached script (and the counters file it creates and manages) would live on the Clarity LIMSServer. Other scripts would depend upon that script and use code similar to below whenever a sequential number was needed. While the script is written in Python and uses the dbm module, there is nothing inherently Pythonic about this code that couldn't be reimplemented in another language. However, more information on the Python dbm module can be found at: <https://docs.python.org/2/library/dbm.html>

```
def test1(): cm = counterManager() cm.setPath( "./test" ) if cm.setup() is True : print (
"INFO: setup Counter Manager" ) print ( "INFO: attempting to call getNextValue for: testA
..." ) tmp = cm.getNextValue( "testA" ) print ( "INFO: getNextValue returned:" + str (tmp) )
else : print ( "ERROR: Failed to setup Counter Manager")
```

User interaction

- The counters live in a file, the path to which is defined in the cm.setPath() command. The file will be created if it doesn't exist.
- The file can contain as many counters as you wish (it's better to have many counters in one file than many files each with only one counter)
- The name of the counter is passed to the function cm.getNextValue(). If this is the first time the counter has been used, it will be created and added to the file.
- Each time you want the next value just call cm.getNextValue() for that counter and you will be given the next value.
- The counters and the file will look after themselves, you don't need to explicitly update or save them – this is all handled behind the scenes.

Assumptions and Notes

- You are running a version of Python supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[clarityCounters.py.txt](#)

Generic CSV Parser Template (Python)

Compatibility: API version 2

Many different types of CSV files are attached to BaseSpace Clarity LIMS. This example provides a template for a script that can parse a wide range of simple CSV files into Clarity LIMS. The user can change parameters to read the format of the file to be parsed.

i | The Lab Instrument Tool Kit includes the parseCSV script, which allows for parsing of CSV files. However, this tool has strict limitations in its strategy for mapping data from the file to corresponding samples in Clarity LIMS. For information on the parseCSV script, refer to the [Clarity LIMS Integrations and Tool Kits](#) documentation.

Solution

Protocol step configuration

CSV files are attached to a protocol step. Artifact UDFs, where data will be written to, need to be configured for artifacts and for the protocol step.

Parameters

The script accepts the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-s	The URI of the step that launches the script - the {stepURI:v2:http} token (Required)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/genericParser.py -u {username} -p {password} -s {stepURI:v2} -r {compoundOutputFileLuid0}
```

About the Code

The script contains an area with a number of configurable variables. This allows an FAS or bioinformatician to customize the script to parse their specific file. The following variables within the script are configurable:

MAPPING MODE	What will the script use to map the measurements to the artifacts in LIMS? (see Generic CSV Parser Template (Python) for more detail)
artifactUDFMap	A Python dictionary where the key is the name of a column in the txt file, and the value is a UDF in Clarity LIMS. (see Generic CSV Parser Template (Python) for more detail)
delim	How is the file delimited? (ex. ',' for .commas or '\t' for tabs)

MAPPING MODES

There are many attributes of samples that can be used to map the data in the text file with the corresponding derived samples in Clarity LIMS. The script should be configured such that one of these modes is set to True.

Three modes are available:

MapTo_WellLocation	The data will be associated with the well locations of the output artifacts for the given step.
MapTo_WellLocation	The data will be associated with the well locations of the output artifacts for the given step.
MapTo_UDFValue	The data will be associated with the value of a specified UDF of the output artifacts.

For any of the three modes, a mapping column value must be explicitly given. The value is the index of the column containing the mapping data (either artifact name, well location, or UDF value).

If using the mode MapTo_UDFValue, a UDFName must be given. This is the name of the UDF in clarity which will be used to match the value found in the mapping column.

artifactUDFMap

```
artifactUDFMap = { "Concentration" : "Concentration", "Avg. Size" : "Average Size" }
```

This Python dictionary maps the name of columns in the txt file to artifact UDFs for the outputs of the step. The data from these columns in the file will be written to these UDFs for the output artifacts. The dictionary can contain an unlimited number of UDFs. The dictionary keys, (left side), are the names of the columns in the txt file, and the dictionary values, (right side), are the names of the UDFs as configured for the artifacts.

Assumptions and Notes

- You are running a version of Python that is supported by Clarity LIMS, as documented in the [Technical Requirements](#).
- The attached files are placed on the LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[genericParser.py.txt](#)

[glsfileutil.py.txt](#)

Renaming Samples to Add an Internal ID

When a lab takes in samples, they are named by the lab scientist who supplies them. As such, samples may be named in any way imaginable. Conversely, the research facilities processing the samples often have strict naming conventions.

When these two situations occur, there is a need for the sample to be renamed with the strict nomenclature of the processing lab. However, in order for the final data to be meaningful for the scientist who supplied the samples, the original name must also be retained.

Solution

In this example, the attached script may be used to rename samples, while retaining their original names in a separate field.

- The original name of the sample is saved in a user-defined field (UDF) called **Customer's Sample Name** , .
 - The **Sample Name** for the submitted sample is overwritten, using the specific naming convention of the customer.
- It is recommended that the script be launched by a process/protocol step as early as possible in the sample lifecycle.

Parameters

The script is invoked with just three parameters:

-l	The limsid of the process invoking the script (Required)
-u	The username of the current user (Required)
-p	The password of the current user (Required)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/renameSamples.py -l 2-1234 -u admin -p securepassword
```

About the Code

Once the command-line parameters have been harvested, and the API object set up ready to handle API requests, the **renameSamples()** method is called.

This method implements the following pseudo code:

```
step 1: get the inputs to this process for each input: step 2: get the URI of the submitted sample associated with the inputs step 3: get the sample step 4: update the sample step 5: save the sample
```

Assumptions and Notes

- Both of the attached files are placed on the BaseSpace Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- A user defined field, named **Customer's Sample Name** has been created at the analyte (sample) level.
- The **Customer's Sample Name** field is visible within the LabLink **Collaborations Interface**.
- You will need to update the HOSTNAME global variable such that it points to your Clarity LIMS server.
- You will need to implement your own logic to apply the new sample name in Step 4 of the script.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[renameSamples.py.txt](#)

Creating Custom Sample Sheets

BaseSpace Clarity LIMS can create Illumina-based MiSeq and HiSeq 'flavoured' sample sheets. However, if you are using algorithms or indexes outside of those suggested by Illumina, you may be required to produce your own 'custom' sample sheet.

This example script provides an algorithm that harvests the contents of a flow cell (or any container) that may contain pooled samples, and uses the resulting information to output a custom sample sheet.

Solution

The attached script uses aggressive caching in order to execute as quickly as possible. When extreme levels of multiplexing are involved, the cache size could consume considerable quantities of memory, which may be counter-productive.

The algorithm has been tested on the following unpooled analytes; pooled analytes; and 'pools of pools' — in which multiple homogeneous or heterogeneous pools are themselves combined to produce a new pool

In these tests, the algorithm behaved as expected. If you find this is not the case please contact us.

1. The algorithm uses recursion to determine the individual analytes (samples) and their indexes that are located on the flow cell lane(s).
2. To determine whether an analyte constitutes a pool or not, the script looks at the number of submitted samples with which the analyte is associated.
 - If the answer is 1, the analyte is not a pool.
 - If the answer is greater than 1, the analyte is considered to be a pool.
3. If a pooled analyte is discovered, the inputs of the process that produced the pooled analyte are gathered and the same test is used to see if they themselves are pools.
4. This gathering of ancestor analytes continues until the contents of each pool have been resolved, at which point the script produces some example output.
 - Note that while it is expected that you will augment this section of the script with the fields you need for your custom sample sheet, the logic to recursively identify analytes that are not themselves pools should be applicable to all.

Parameters

The script is invoked with just three parameters:

-l	The luid of the flow cell / container of interest (Required)
-u	The username of the current user (Required)
-p	The password of the current user (Required)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/flowcellContents.py -l 27-1234 -u admin -p securepassword
```

Assumptions and Notes

- Both of the attached files are placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- You will need to implement your own logic to gather the fields required for your specific sample sheet.
- You will need to update the HOSTNAME global variable such that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[flowcellContents.py.txt](#)

Copying Output UDFs to Submitted Samples

It may at times be desirable to take key data derived during a workflow and copy it to the submitted sample. There are several reasons why this could be useful:

- All key data is combined with all of the submitted sample's data, and becomes available on a single object.
- Key data determined during a workflow can be made immediately available to external collaborators via the LabLink **Collaborations Interface**, since these users have access to their submitted samples.
- Searching for data becomes easier as the data is not spread over several entities.

This example provides a script to allow the copying to occur, and describes how the script can be triggered.

Solution

To illustrate the script, we will copy a user-defined field (UDF) that is collected on the outputs of a QC type protocol step.

- This UDF is named **Concentration**, and it is stored on the individual ResultFile entities associated with the analytes that went through the QC protocol step.
- Once the QC protocol step has completed, the **Concentration** UDF values are copied to a UDF on the submitted Samples, which is called **Sample Conc**.
- The QC protocol step is configured to invoke the script from a button on the step's **Record Details** screen.

Parameters

The EPP command is configured to pass the following parameters:

-l	The limsid of the process invoking the script (Required)	
-u	The username of the current user (Required)	
-p	The password of the current user (Required)	
-f	The UDF on the submitted sample to be copied to (Required)	To copy multiple UDF values from outputs to submitted samples, a list of comma-separated values may be provided.
-t	The type of output that has the UDF to be copied from (Required)	For example, Analyte, ResultFile, Pool, etc.
-v	The name of the UDF to be copied from (Required)	To copy multiple UDF values from outputs to submitted samples, a list of comma-separated values may be provided.

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/setUDFonSample.py -l MCL-SA1-131211-24-6813 -u admin -p securepassword -f "Sample Conc., Units" -t ResultFile -v "Concentration, Conc.Units"
```

User Interaction

Once the script has copied the UDF values from the output to the submitted samples, the values are visible in the **Submitted Samples** view of the **Operations Interface**:

SamplesFilesDetailsPermissionsNotes

✱ Navigator

Submitted Samples

Containers

Samples

✱ Look for:

Find

Clear

☐ Exact matches only

☐ Highlight matches

QC	Sample Name	Workflows	Date Received	Date Completed	Container	Well (row:column)	Sample LIMS ID	Project	Type	Progress	Sample Conc.	Sec
	SG-1564-1	ML Point Billing	10-Dec-2013		27-2851	1:1	ADM1351A1	MCL UDF Testing			10.00	
	SG-1564-2	ML Point Billing	10-Dec-2013		27-2852	1:1	ADM1351A2	MCL UDF Testing			11.00	
	SG-1564-3	ML Point Billing	10-Dec-2013		27-2853	1:1	ADM1351A3	MCL UDF Testing			22.00	
	SG-1564-4	ML Point Billing	10-Dec-2013		27-2854	1:1	ADM1351A4	MCL UDF Testing			13.00	
	SG-1564-5	ML Point Billing	10-Dec-2013		27-2855	1:1	ADM1351A5	MCL UDF Testing			14.00	

Similarly, assuming that the **Sample Conc.** UDF is set to be visible within LabLink Collaborations Interface, collaborators are able to see these values in their interface:

Samples Files Notes										
View samples with status: Received					Apply					
Name	LIMS ID	Status	Project	Date Received	Last Note	Sample Type	Progress	Sample Conc.	Category	
SG-1564-1	ADM1351A1	Received	MCL UDF Testing	Dec 11, 2013				10.00		
SG-1564-2	ADM1351A2	Received	MCL UDF Testing	Dec 11, 2013				11.00		
SG-1564-3	ADM1351A3	Received	MCL UDF Testing	Dec 11, 2013				22.00		
SG-1564-4	ADM1351A4	Received	MCL UDF Testing	Dec 11, 2013				13.00		
SG-1564-5	ADM1351A5	Received	MCL UDF Testing	Dec 11, 2013				14.00		

About the Code

The main method of interest is **setUDFs()**. This method carries out several operations:

1. It harvests just enough information so that the objects required by the subsequent code can retrieve the required artifacts using the 'batch' API operations. This involves using some additional code to build and manage the cache of artifacts retrieved in the batch operations, namely:
 - `cacheArtifact()`
 - `prepareCache()`
 - `getArtifact()`
2. The cached artifacts are then accessed, and for each one:
 - The corresponding sample is retrieved via the API.
 - The sample XML is updated such that the UDF value is obtained from the artifact by calling **api.getUDF()**, and stored on the sample by calling **api.setUDF()**.
 - The sample XML is saved by calling **api.updateObject()**.
3. Finally, a meaningful message is reported back to the user via the contents of the **successMsg** and/or **failMsg** variables.

Assumptions and Notes

- Both of the attached files are placed on the BaseSpace Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- You will need to update the `HOSTNAME` global variable such that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[setUDFonSample.py](#)

Generating a Custom HiSeq Sample Sheet

In most cases, the **SampleSheet.csv** file that is created by BaseSpace Clarity LIMS for Illumina HiSeq instruments can be used without modification. However, it may sometimes be necessary to modify this sample sheet.

For example, you may want to:

- Use the Library Name in place of the Sample Name.
- Use the Sample LIMSID in place of the Sample Name.
- Use the Project LIMSID in place of the Project Name.

This example demonstrates how to achieve this.

Solution

You can use the attached script in place of the existing Clarity LIMS SampleSheet generation script, to generate a sample sheet for Illumina HiSeq instruments.

Parameters

The EPP command is configured to pass the following parameters:

-l	The limsid of the process invoking the script - the process that created the flow cell (Required)	The {processLuid} token
-u	The username of the current user (Required)	The {username} token
-p	The password of the current user (Required)	The {password} token
-o	The name of the file produced	Typically the {compoundOutputFileLuidX}.csv token, where X reflects the index of the SampleSheet output of the process)

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/HiSeqSampleSheet.py -l 122-7953 -u admin -p securepassword -o 92-1234.csv
```

About the Code

The main method of interest is **getFlowCellContents()**.

This method does the following:

1. Harvests the flow cell produced by the process.
2. For each lane, gathers the contents of the lane and demultiplexes any pooled libraries. (This demultiplexing of pools reuses the code and logic covered in the [Creating Custom Sample Sheets](#) application example.)

3. Once the indexed constituents of each pool have been retrieved, the script then:

- Gathers the associated sample and project level information to be included in the sample sheet.
- Creates the sample sheet file.

Creating the file

The following line of the script assembles the data needed for each row of the sample sheet file:

```
line = cName + ',' + str(WP) + ',' + sLimsID + ',,' + index + ',' + sName + ',N,,Script,'" +  
pName + '''
```

The following examples show how this line may be edited to accommodate modifications required in the generated file.

Example 1 – Here, the Library Name is used in place of the Sample Name:

```
line = cName + ',' + str(WP) + ',' + sLimsID + ',,' + index + ',' + aName + ',N,,Script,'" +  
+ pName + '''
```

Example 2– Here, the the Project LIMS ID is used in place of the Project Name:

```
line = cName + ',' + str(WP) + ',' + sLimsID + ',,' + index + ',' + sName + ',N,,Script,'" +  
pLimsID + '''
```

Assumptions and Notes

- Both of the attached files are placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- You will need to update the HOSTNAME global variable such that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[HiSeqSampleSheet.py.txt](#)

Parsing Sequencing Meta-Data into Clarity LIMS

Once a sequencing run has occurred, there is often a requirement to store the locations of the FASTQ / BAM files in BaseSpace Clarity LIMS.

For paired-end sequencing, it is likely that the meta-data file that describes the locations of these files will contain two rows for each sample sequenced: one for the first read, and another for the second read.

Such a file is illustrated here:

Project	Sample ID	Fastq File
MCL External Sequencing 001	ADM1901A1PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1238H1/NGS0099-1238H1_GACGCTAT_L002_R1_001.fastq.gz
MCL External Sequencing 001	ADM1901A1PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1238H1/NGS0099-1238H1_GACGCTAT_L002_R2_001.fastq.gz
MCL External Sequencing 001	ADM1901A2PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1239H2/NGS0099-1239H2_AGTATCGC_L002_R1_001.fastq.gz
MCL External Sequencing 001	ADM1901A2PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1239H2/NGS0099-1239H2_AGTATCGC_L002_R2_001.fastq.gz
MCL External Sequencing 001	ADM1901A3PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1240H3/NGS0099-1240H3_GAATCTCG_L002_R1_001.fastq.gz
MCL External Sequencing 001	ADM1901A3PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1240H3/NGS0099-1240H3_GAATCTCG_L002_R2_001.fastq.gz
MCL External Sequencing 001	ADM1901A4PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1241H4/NGS0099-1241H4_CGTATGCA_L002_R1_001.fastq.gz
MCL External Sequencing 001	ADM1901A4PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1241H4/NGS0099-1241H4_CGTATGCA_L002_R2_001.fastq.gz
MCL External Sequencing 001	ADM1901A5PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1242H5/NGS0099-1242H5_GCACGTAT_L002_R1_001.fastq.gz
MCL External Sequencing 001	ADM1901A5PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sample_NGS0099-1242H5/NGS0099-1242H5_GCACGTAT_L002_R2_001.fastq.gz

Column 2 of the file, **Sample ID**, contains the LIMS IDs of the artifacts for which we want to store the FASTQ file values listed in column 3 (**Fastq File**).

This example discusses the strategy for parsing and storing data against process inputs, when that data is represented by multiple lines in a data file.

Solution

The attached script will parse a data file containing multiple lines of FASTQ file values, and will store the locations of those FASTQ files in user-defined fields.

Process Configuration

In this example, the process is configured to have a single shared ResultFile output.

Parameters

The EPP command is configured to pass the following parameters:

-u	The username of the current user (Required)	The {username} token
-p	The password of the current user (Required)	The {password} token
-s	The URI of the step that launches the script (Required)	The {stepURI:v2:http} token
-i	The limsid of the shared ResultFile output (Required)	The {compoundOutputFileLuid0} token

An example of the full syntax to invoke the script is as follows:

```
/usr/bin/python /opt/gls/clarity/customextensions/parseMetadata.py -l 24-9953 -u admin -p
securepassword -s http://192.168.9.123:8080/api/v2/steps/24-9953 -i 92-20553
```

User Interaction

The user-interaction is comprised of the following steps:

Step 1

The user runs the process up to the **Record Details** screen as shown in the following image. Note that initially:

- The Sequencing meta-data file is still to be uploaded.
- The values for the **R1Filename** and **R2 Filename** fields are empty.

Upload attach the file containing the meta-data, then press the button

Please review your parameters prior to clicking Parse Meta-data File.

LIMS ID (Process)

24-9953

Parse Meta-data File

Run

Files

Sequencing MetaData File

Upload file

Sample Details

R2 Filename

Apply

Sample Name	LIMS ID (Derived Sample)	R1 Filename	R2 Filename
Single Well: Tube (5 samples)			
Ex-Seq-1	ADM1901A1PA1		
Ex-Seq-2	ADM1901A2PA1		
Ex-Seq-3	ADM1901A3PA1		
Ex-Seq-4	ADM1901A4PA1		
Ex-Seq-5	ADM1901A5PA1		

Step 2

The user clicks **Upload file** and attaches the meta-data file. Once attached, the user's screen will resemble this:

Upload attach the file containing the meta-data, then press the button

Please review your parameters prior to clicking Parse Meta-data File.

LIMS ID (Process)

24-9953

Parse Meta-data File

Run

Files

Sequencing MetaData File

× Demo-MetaDataSheet.csv

Sample Details

R2 Filename

Apply

Sample Name	LIMS ID (Derived Sample)	R1 Filename	R2 Filename
Single Well: Tube (5 samples)			
Ex-Seq-1	ADM1901A1PA1		
Ex-Seq-2	ADM1901A2PA1		
Ex-Seq-3	ADM1901A3PA1		
Ex-Seq-4	ADM1901A4PA1		
Ex-Seq-5	ADM1901A5PA1		

Step 3

Now that the meta-data file is attached, the user clicks **Parse Meta-data File**. This invokes the parsing script. If parsing was successful, the user's screen will resemble the next image.

Note that the values for the **R1Filenames** and **R2 Filenames** have been parsed from the file and will be stored in Clarity LIMS.

Upload attach the file containing the meta-data, then press the button

Parse Meta-data File successful! March 24, 2014 - 2:43 pm
Parsing completed Successfully

LIMS ID (Process)
24-9953

Parse Meta-data File Run

Files

Sequencing MetaData File x 92-20553

Sample Details R2 Filename Apply

Sample Name	LIMS ID (Derived Sample)	R1 Filename	R2 Filename
Single Well: Tube...			
Ex-Seq-1	ADM1901A1PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sa...	140115_D00232_0064_AC3A60ACXX/Unaligned
Ex-Seq-2	ADM1901A2PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sa...	140115_D00232_0064_AC3A60ACXX/Unaligned
Ex-Seq-3	ADM1901A3PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sa...	140115_D00232_0064_AC3A60ACXX/Unaligned
Ex-Seq-4	ADM1901A4PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sa...	140115_D00232_0064_AC3A60ACXX/Unaligned
Ex-Seq-5	ADM1901A5PA1	140115_D00232_0064_AC3A60ACXX/Unaligned/Sa...	140115_D00232_0064_AC3A60ACXX/Unaligned

About the Code

The key methods of interest are **main()**, **parseFile()** and **fetchFile()**. The **main()** method calls **parseFile()**, which in turn calls **fetchFile()**.

fetchFile() method

The **fetchFile()** method relies upon the fact that the script is running on the Clarity LIMS server, and as such has access to the local file system in which the file (uploaded in Step 2) now resides.

Thus, **fetchFile()** can use the API to:

1. Convert the LIMSID of the file to the location on disk.
2. Copy the file to the local working directory, ready to be parsed by **parseFile()**.

parseFile() method

1. The **parseFile()** method creates two data structures that are used in the subsequent code within the script:
 - The **COLS** dictionary has the column names from the first line of the file as its key, and the index of the column as the value.

- The **DATA** array contains each subsequent line of the file as a single element. Note that this parsing logic is overly-simplistic, and would need to be supplemented in a production environment. For example, if the CSV file being parsed does not have the column names in the first row, then exceptions would likely occur. Similarly, we assume the file being parsed is CSV, and likewise any data elements which themselves contain commas would likely cause a problem. For the sake of clarity such exception handling has been omitted from the script.
2. Once **parseFile()** has executed successfully the inputs to the process that has invoked the script are gathered, and 'batch' functionality is used to gather all of the artifacts in a single batch-retrieve transaction.
 3. All that remains is to step through the elements within the DATA array, and for each line, gather the values of the **Fastq File** and **Sample ID** columns. For each **Sample ID** value:
 - The corresponding artifact is retrieved.
 - Depending on whether the value of the **Fastq File** column represents the filename for the first or second read, either the **R1 Filename** or the **R2 Filename** user-defined field is updated.
 4. Once the modified artifacts have been saved, the values will display in the Clarity LIMS Web Interface, as shown in Figure 4.

Assumptions and Notes

- Both of the attached files are placed on the Clarity LIMS server, in the **/opt/gls/clarity/customextensions** folder.
- You will need to update the HOSTNAME global variable such that it points to your Clarity LIMS server.
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[NextStepAutomation.groovy.txt](#)

Submit to a Compute Cluster via PBS

There are some algorithms which work well on massively parallel compute clusters. BCL Conversion is such an example and is the basis for this application example.

The concepts illustrated here are not limited to BCL Conversion; as such, they may also be applied in other scenarios. For instance the example PBS script below uses syntax for illumina's CASAVA tool, but could easily be re-purposed for the bcl2fastq tool.

Also, in this example, **Portable Batch System** (PBS) is used as the job submission mechanism to the compute cluster, which has read/write access to the storage system holding the data to be converted.

Example PBS file

For illustrative purposes, an example PBS file is shown here. (As there are many ways to configure PBS, it is likely that the content of your PBS file(s) will differ from the example provided.)

```
#!/bin/bash #PBS -N run_casava #PBS -q himem #PBS -l nodes=1:ppn=20 export RUN_
DIR=/data/instrument_data/120210_SN1026_0092_BXXXXXXXXX export OUTPUT_DIR=/data/processed_
data/processed_data.1.8.2/120210_SN1026_0092_BXXXXXXXXX export SAMPLE_
SHEET=/data/SampleSheets/samplesheet.csv cd $PBS_O_WORKDIR source /etc/profile.d/modules.sh
module load casava-1.8.2 export TMPDIR=/scratch/ export NUM_PROCESSORS=$((PBS_NUM_NODES*PBS_
NUM_PPN)) configureBclToFastq.pl --input-dir $RUN_DIR/Data/Intensities/BaseCalls --output-
dir $OUTPUT_DIR --sample-sheet $SAMPLE_SHEET --force --ignore-missing-bcl --ignore-missing-
stats --use-bases-mask y*,I6,y* --mismatches 1 cd $OUTPUT_DIR make -j $NUM_PROCESSORS
```

Solution

Process Configuration

In this example, the **BCL Conversion** process is configured to:

- Accept a ResultFile input.
- Produce at least two ResultFile outputs.

The process is configured with the following process level UDFs:

Input/Output ExplorerFilesDetailsNotes

Details

Run Name:

Run42

Number of Cores:

20

Number of mismatches:

1

Bases mask:

y*,I6,y*

The syntax for the external program parameter is as follows:

```
python /opt/gls/clarity/customextensions/ClusterBCL.py -l {processLuid} -u {username} -p
{password} -c {udf:Number of Cores} -m {udf:Number of mismatches} -b "{udf:Bases mask}" -a
{compoundOutputFileLuid0}.txt -e {compoundOutputFileLuid1}.txt -r "{udf:Run Name}"
```

Parameters

-l	The lmsid of the process invoking the code (Required)	The {processLuid} token
-u	The username of the current user (Required)	The {username} token
-p	The password of the current user (Required)	The {password} token
-c	The number of CPUs to dedicate to the run (Required)	The {udf:Number of Cores} token

-l	The lmsid of the process invoking the code (Required)	The {processLuid} token
-m	The number of mismatches (Required)	The {udf:Number of mismatches} token
-b	The bases mask to be used (Required)	The {udf:Bases mask} token
-a	The name of the first output file produced (Required)	The {compoundOutputFileLuid0} token
-e	The name of the second output file produced (Required)	The {compoundOutputFileLuid1} token
-r	The name of the run (Required)	The {udf:Run Name} token

User Interaction and Results

1. The user runs the **BCL Conversion** process on the output of the **Illumina Sequencing** process. The sequencing process is aware of the Run ID, as this information is stored as a process level user-defined field (UDF).
2. The user supplies the following information, which is stored as process level UDFs on the **BCL Conversion** process:
 - The name of the folder in which the converted data should be stored.
 - The bases mask to be used.
 - The number of mismatches.
 - The number of CPUs to dedicate to the job.
3. The **BCL Conversion** process launches a script (via the EPP node on the Clarity LIMS server) which does the following:
 - Builds the PBS file based on the user's input.
 - Submits the job by invoking the ' *qsub* ' command along with the PBS file.

Assumptions and Notes

- Portable Batch System (PBS) is used as the job submission mechanism to the compute cluster.
- The compute cluster has read/write access to the storage system holding the data to be converted.
- There is an EPP node running on the Clarity LIMS server.
- The PBS client tools have been installed and configured on the Clarity LIMS server, such that the ' *qsub* ' command can be launched directly from the server.
- When the ' *qsub* ' command is invoked, a PBS file is referenced; this file contains the job description and parameters.

- The script was written in Python (version 2.7) and relies upon the **GLSRestApiUtil.py** module. Both files are attached below. The required Python utility is available for download at [Using the Utility Classes](#) .
- The example code is provided for illustrative purposes only. It does not contain sufficient exception handling for use 'as is' in a production environment.

Attachments

[ClusterBCL.py.txt](#)

Downloading a File and PDF Image Extraction

Compatibility: API version 2 revision 21

A lab user can attach a PDF file containing multiple images to a result file placeholder, the script extracts the images which are automatically attached to corresponding samples as individual result files.

Images within a PDF may be in a number of formats, and will usually be .ppm or .jpeg. The example script includes additional code to convert .ppm images to .jpeg.

Prerequisites

- You have the package 'poppler' (linux) installed.
- You have defined a process with analytes (samples) as inputs, and outputs that generate the following:
 - A single shared result file output.
 - A result file output per input.
- You have added samples to BaseSpace Clarity LIMS.
- You have uploaded the Results PDF to Clarity LIMS during 'Step Setup'.
- Optionally, if you wish to convert other file types to .jpeg, installation of ImageMagick (Linux package).

Code Example

How it works:

1. The lab scientist runs a process/protocol step and attaches the PDF in BaseSpace Clarity LIMS.
2. When run, the scrip uses the API and the 'requests' package available in Python to locate and retrieve the PDF.
3. The script generates a file for each image.
4. Files are named with LUIDs and well location.
5. The images are attached to the ResultFile placeholder. ****The file names must begin with the {outputFileLuids} for automatic attachment.****

Additionally, this script converts the images to JPEG format for compatibility with other LIMS features.

Step 1. Create the script

Part 1 - Downloading a file using the API

The script finds and gets the content of the PDF through two separate GET requests:

1. Following the artifact URI using the **{compoundOutputFile0}** to identify the LUID of the PDF file.
2. Using the `~/api/v2/files/{luid}/download` endpoint to save the file to the temporary working directory.

```
def dl_pdf( artifactluid_ofpdf ): ### finds the file LUID from artifact LUID of the PDF
artif_URI = BASE_URI + "artifacts/" + artifactluid_ofpdf
artGET = requests.get(artif_URI,
auth=(args[ "username" ],args[ "password" ]))
root = ET.fromstring(artXML)
for id in root.findall("{http://genologics.com/ri/file}file"):
fileLUID = id.get("limsid")
file_URI = BASE_URI + "files/" + fileLUID + "/download"
fileGET = requests.get(file_URI, auth=(args[ "username" ],args[ "password" ]))
with open("frag.pdf", 'wb') as fd:
for chunk in fileGET.iter_content():
fd.write(chunk)
```

The PDF is written to the temporary directory.

The script performs a batch retrieval of the artifact XML for all samples. Subsequently a python dictionary is created defining which LIMS id corresponds to a given well location.

Part 2 - Extracting images as individual results files

The script uses the `pdfimages` function to extract the images from the PDF. This function is from a linux package and can be called using the `os.system()` function.

This example script extracts an image from each page, beginning with page 10. Files are named with LUIDs and well location. The file names must begin with the **{outputFileLuids}** for automatic attachment.

```
page = 10
for each in range(len(wells)):
well_loci = wells[each]
if well_loci in well_map.keys():
limsid = well_map[well_loci]
filename = limsid + "_" + well_loci
command = 'pdfimages ' + 'frag.pdf' + ' -j -f ' + str(page) + ' -l ' + str(page) + ' ' + filename
os.system(command)
```

Additionally, the cookbook example script converts the image files to JPEG for compatibility with other features in Clarity LIMS. The script uses 'convert', a function from the linux package called 'ImageMagick'. Like the 'pdfimages' function, 'convert' can be called in a python script through the `os.system()` function.

Step 2. Configure the Process

The steps required to configure a process to run EPP are described in the [Process Execution with EPP/Automation Support](#) example, namely:

1. Configure the inputs and outputs.
2. On the **External Programs** tab, select the check box to associate the process with an external program.

Parameters

The process parameter string for the external program is as follows:

```
bash -c "/usr/bin/python /opt/gls/clarity/customextensions/pdfimages.py -a
{compoundOutputFileLuid0} -u{username} -p {password} -f '{outputFileLuids}'"
```

The EPP command is configured to pass the following parameters:

-u	The username of the current user (Required)
-p	The password of the current user (Required)
-f	A list of the output LUIDS (required, must be in quotes "")

Step 3. Run the Process

Record Details page in Clarity LIMS

The placeholder where the lab scientist can upload the PDF.

External Process ready to be run. ' *Script generated*' message marking individual result file placeholder.

Expected Output and Results

External program was run successfully. Individual result files named with artifact LUID and well location.

Attachments

[pdfimages.py.txt](#)

Resources and Reference

Additional reference information for BaseSpace Clarity LIMS.

Revision History

Document	Date	Description of Change
Document # 1000000150201 v02	August 2023	Updated API Portal page for REST AP v2 r32 and v2 r33 (for Clarity LIMS v6.1 and v6.2).
Document # 1000000150201 v01	March 2022	Updated API Portal page for REST API v2 r31. Added Project Automation Tokens section to support Clarity LIMS v6. In Step Automation Tokens, removed extra token name on 'response / output' example. Noted that parent process step type must produce derived samples for the token to be evaluated. In Email Notifications, updated Mail Server Configuration section. Updated example command line and script. Moved Resources & Reference section from Overview to top level in navigation. Added revision history.
Document # 1000000150201 v00	January 2021	Initial release in new help format.

Understanding LIMS ID Prefixes

Some of the more common LIMS ID prefixes are listed in the following table.

Prefix	Object
1	Address
2	Analyte (derived sample)
24	Standard Step (not indexing—ie, not Add Labels or Pooling)
27	Container
40	File
92	ResultFile (artifact), both per-sample and step-level
120	Reagent type
122	Pooling step
151	Indexing step (ie, Add Labels)
152	Reagent label

 | QC steps do not have a special LIMS ID prefix.

In addition, there are also some esoteric and deprecated LIMS ID prefixes. The complete list of prefixes can be found in the class index database table.

Translating a Step in the Web Interface to a Step URI in the API

Assuming you have a started a step, your browser address will look something like this:

```
https://demo.claritylims.com/clarity/work-placement/21816
```

The important piece is the suffix: 21816.

To convert to a step URI:

- 1. Note the type of step you are running. Is it a Standard step, an Indexing (Add Labels) step, or a Pooling step?
- 2. Add the corresponding hyphenated prefix using the previous list, to construct the step ID. For example, if it is an Indexing step, 21816 becomes 151-21816.

3. Build the step URI:

```
https://hostname/api/version/steps/151-21816.
```

Translating a Step URI in the API to a Step in the Web Interface

To perform the reverse action and find a step in the web interface for a given a step URI (or a process URI):

- 1. Get the step ID. Using the example above, the step ID would be 151-21816.
- 2. Strip out the hyphenated prefix to give: 21816.
- 3. Build the web address:

```
https://hostname/clarity/work/21816.
```

BaseSpace Clarity LIMS translates the word work to the correct address for the screen the step is on. For example, if the step is completed, work becomes work-complete.

Container States

The Container database table includes the StateId column. Unlike the states of artifacts, container stateids can be easily deciphered. (In Clarity LIMS, the stateids correspond to the states displayed in the BaseSpace Clarity LIMS Operations Interface).

As there is no table in the database that defines these stateid values, you will need to hard code the values in your queries based on the information provided in the following table:

StateId	State	Container Type
1	Empty	Normal
2	Populated	Normal
3	Depleted	Normal
4	Discarded	Normal
5	Reagent-Only	Normal

StateId	State	Container Type
6	New	Array*
7	Hybridized	Array*
8	Scanned	Array*
9	Discarded	Array*

* Since the concept of arrays does not apply to the BaseSpace Clarity LIMS Web Interface, only states 1-5 apply.

Example 1

To search for discarded and depleted containers, use the following example:

```
SELECT * FROM container WHERE stateid IN (3,4);
```

Example 2

To see all the arrays in the system that are new and still need to be scanned, use the following example:

```
SELECT * FROM container WHERE stateid = 6;
```

Useful Tools

To help with learning, BaseSpace Clarity LIMS documentation includes:

- An introduction to the API architecture (see [REST General Concepts](#)).
- Details of the methods and interfaces (see REST API documentation and [Automation Overview](#)).
- Example code (see [About the Cookbook](#)).

However, there is more to the Clarity LIMS Rapid Scripting™ API package than these resources.

Details on tools, training, and best practices are covered in the following sections and in [Development Prerequisites](#).

Nonproduction Scripting Sandbox Server

Nonproduction server licenses and training are essential. A nonproduction scripting sandbox server is set up with Groovy, BaseSpace Clarity LIMS, and automation ready to be used with the scripts provided in the [About the Cookbook](#) and with the [Clarity LIMS API Application Examples](#).

- Have the nonproduction servers set up before learning to script (see [Development Prerequisites](#)).
- All the examples in the Cookbook are intended to be used with the nonproduction scripting sandbox server. This server has all the software packaged up to start scripting with the API.
- The Cookbook examples do not explain how to configure the servers for process configuration. Server installation and server-side configuration instructions are covered during implementation, as part of the installation and training services.

- For more information and to understand the importance of a dedicated scripting server, see [Development Prerequisites](#).

Cookbook Examples for Learning the API on Nonproduction Servers

The nonproduction server license and the Cookbook examples are not for production use. It is not practical to integrate actual instruments to this server permanently, but it does provide a great way to learn to script.

The nonproduction scripting sandbox server provides a safe place to learn the API, using the Cookbook example scripts as a guide. The typical installation includes the Clarity LIMS software, the Automation Informatics node (referred to as the automation worker in Clarity LIMS v5), and Groovy.

Installation and Administration Training

Installation and administration training enables IT administrators to support multiple script writers with multiple nonproduction servers for development and testing. Administrators learn about server-side tools and procedures, such as the backup and restore procedure. This procedure moves scripts between systems.

- The production and nonproduction server installation training is provided to IT administrators at the start of each implementation or when the Rapid Scripting API package is purchased.
- Automation Worker (AW) node training is also included. This component can be installed on multiple computers using the Automation Worker software package.
- Explore other training, consulting, and issue management services available from Illumina in case the packaged services are exceeded.

Contact Illumina for additional training or support installing and maintaining servers and AW nodes.

API Training and Consulting

The script training includes instructor-led, hands-on sessions and the training included on this site.

Instructors start from a basic Hello World script kicked off from within Clarity LIMS using automation, then progress to more in-depth concepts.

Also provided in the package is on-demand consulting. For quality service, all requests are handled via the Support team. However, expert script writers are available in technical services to consult by email, phone, or web-based sessions.

Script Deployment

Illumina recommends placing scripts in subdirectories starting from a common root, namely `/opt/gls/clarity/customextensions/`.

This makes moving scripts between development, test, and production systems much easier and less error-prone. This directory is assumed by the backup and restore process.

Use a directory structure beneath this root to expand the deployed scripts over time. This customextensions folder is reserved exclusively for your custom scripts.

At Illumina, this practice is called deploying on the rails. The following directories illustrate how this method is used to deploy multiple instrument integrations.

```
/opt/gls/clarity/extensions/  
/opt/gls/clarity/extensions/Illumina/HiSeq/v1.1  
/opt/gls/clarity/extensions/ABI/SOLiD/v1.0  
/opt/gls/clarity/customextensions/'YourRootDirectory'  
- root dir for all automation scripts  
- Integration from Illumina, do not modify  
- Another integration for Illumina  
- Scripts created by you
```

Putting all the scripts into a common root directory also allows multiple teams to work on the implementation independently. For example, when Illumina provides custom integrations and you are scripting at the same time. The backup and restore process allows you to transfer all files in the root directory between systems. It is important to not place server-specific settings, such as IP addresses or login details, in these directories.

To learn more about the backup and restore process, talk to the IT Administrator (who attended the installation and administration training) and refer to the Clarity LIMS Installation Procedure documentation (accessible from the Installation section of the Clarity LIMS documentation).

Groovy

The Cookbook how-to examples and the nonproduction server use scripts written in Groovy, a Java-based scripting language.

You can use other languages with the API. However, Groovy is easy to understand. Groovy scripts look similar to pseudocode.

In addition, Groovy is particularly suited to real scripting because it handles and manipulates the major inputs, outputs, and interfaces within Clarity LIMS well.

Related Resources

- [Development Prerequisites](#)
- [Integrating Scripts](#)
- [Suggested Reading](#)

Unsupported Artifact Types

An artifact is defined as something that is produced by a process or protocol step. It is the output of a process/protocol step run in Clarity LIMS. For example, analytes and result files are all artifacts.

Currently, the REST API does not support the following artifacts:

- **Reagents**—While reagents are produced by some processes in the Clarity LIMS client, they are not exposed as artifacts in the REST API. Reagents are an entity type of their own available through the list and single reagents resources.
- **Subartifacts**—When a process produces shared outputs, the system creates subartifacts to support the REST data model. While they exist as entities in the system, subartifacts are transparent and meaningless to a user and are therefore not supported.

Unsupported Process Types

Currently, the REST API does not support the following process types:

- Batch Gel Image Analysis Run (known at Gel Analysis Experiment in the client)
- Batch Gel Run (multiple 1D Gel Run or 2D Gel Run processes run at the same time)
- Ettan Spot Handling
- Gel 1D Run (known at 1D Gel Run in the client)
- Gel 2D Run (known at 2D Gel Run in the client)
- Gel Image Analysis Run (known as Image Annotation in the client)
- Gel Imaging Run
- Sample*

i | When adding a sample to Clarity LIMS, the system actually sees the action of add sample as a process. No information pertaining to this process is made available to the user through the client. As a result, it is not supported through the REST API.

Suggested Reading

The following resources provide additional information on the tools and technologies used in the API and in the [About the Cookbook](#) examples.

REST

The Clarity LIMS Rapid Scripting API uses a RESTful web service with XML over HTTP. For more information about REST, see the following:

- http://en.wikipedia.org/wiki/Representational_State_Transfer
- <http://www.xfront.com/REST-Web-Services.html>

Linux and Common File Formats

Many of the scripts used in the Cookbook examples assume a knowledge of Linux and text file formats.

Review bash shell execution command syntax and file formats such as comma-separated values (CSV) and Extensible Markup Language (XML):

- http://en.wikipedia.org/wiki/Comma-separated_values
- <http://en.wikipedia.org/wiki/XML>

Groovy

For its clarity and readability, we have used Groovy for the Cookbook examples. The [Apache Groovy](#) website provides useful resources for learning and working with Groovy:

- <http://groovy-lang.org/learn.html>
- <http://groovy-lang.org/documentation.html>

Additional Resources

- [Useful Tools](#)
- [Integrating Scripts](#)

API Training Videos

To help those who prefer learning at their own pace, we have created video materials that supplement our API training materials. These videos correlate directly to the materials covered in the slides for each module, and in most cases we have also provided videos that illustrate the solution to the exercises.

These videos refer to resource definitions and the API Cookbook:

- Resource Definitions can be accessed through the [API Portal](#).
- The [About the Cookbook](#) is a set of example scripts intended to help you learn how to work with REST and EPP (automation) scripts.

Module 1: Overview of the API

This module outlines the course structure of the API Training Series of modules and provides a general strategy for being successful using the API.

Module 2: General Concepts of the API

This module explores the basic principles of interacting with the API. A browser plugin such as XHR for Chrome or Poster for Firefox is used and illustrates where specific LIMS data are accessible via the API. The PUT, POST, GET, and DELETE actions are explained as to how these operations are supported through the API.

Understanding the concept of a genealogy and how user defined fields are configured and used in the system is uncovered. Completing the exercises for this module is critical before proceeding to Module 3.

Module 2: Example Exercise 1

Module 2: Example Exercise 2

Module 2: Example Exercise 3

Module 3: Groovy

This module explores setting up Groovy and interacting with data in the Clarity LIMS database via the API. Common script examples from the Cookbook section are used to show how to use a scripting language to interact with the API. API example videos are also provided as part of module 3.

Module 3: Example Exercise 1

Module 3: Example Exercise 2

Module 4: Process Inputs and Outputs

This module focuses on how the data for processes that have been carried out in the LIMS are represented in XML format.

Module 4: Example Exercises

Module 5a: EPP in Operations Interface

This module focuses on how to configure a script to be run in the LIMS using the External Program Plugin tab of the Process configuration wizard in the Operations Interface (available in LIMS v4.x and earlier only).

Module 5a: Example Exercises

Module 5b: EPP/Automation in Lab Scientist Interface

This module covers enabling and invoking EPP / automation scripts from the Lab Scientist Interface. In particular, we focus on the possibilities available for both manual and automatic invocation.

Module 5b: Example Exercises