Typefi Publish

Web Services Interface 5.1.x

January 2013

Typéfi®
typefi.com

This document was created with Typefi Publish 6.

WSI Guide 5.1.x

Contents

Introduction

The Web Service Interface (WSI) provides integrators with a standards-based method of interacting with Typefi Publish® from their own applications. The focus of the WSI is to allow for the running of jobs, getting and putting of assets, and sending of messages. Additional servlets allow for getting information about projects, engines, job options, and assets.

Protocol

The WSI is based on the REST (REpresentational State Transfer) web service architecture. All access is via HTTP and XML is used for all data transfer. In short, a client application makes an HTTP request. If the response to that request is informational in nature, the returned data is delivered as XML. If instead the request is for a particular resource, the resource itself is returned. The client application may be as simple as a web browser. Keep in mind, however, that different browsers behave differently when it comes to displaying XML.

Authentication to all WSI servlets is handled with Basic Authentication (see *RFC 2617*).

POST requests that furnish data to a WSI servlet should set the `Content-Type` HTTP header field for the request to `multipart/form-data` (see *RFC 2616*). The `Content-Type` for any provided MIME parts, on the other hand, should always be set specific to that part (`text/xml`). See *"MIME Structure"*.

The examples provided with the WSI distribution are also an excellent resource and may even form the basis for your own application.

Resources

- Basic Authentication (*http://www.ietf.org/rfc/rfc2617.txt*)
- HTTP Header Field Definitions (*http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html*)
- REST (*http://en.wikipedia.org/wiki/Representational_State_Transfer*)

MIME Structure

Many of the servlets require data to be sent as multipart MIME requests. This provides compatibility with HTML forms and makes it fairly straightforward to embed multiple types of data in a single request.

A sample multipart MIME request looks like:

```
POST /put HTTP/1.1
Pragma: no-cache
Content-Type: multipart/form-data; boundary=XYTxYDxxYYYApEf
Host: 192.168.1.147:8080
Content-Length: 10268
Connection: Keep-Alive

--XYTxYDxxYYYApEf
Content-Disposition: form-data; name="File"; filename="=?ISO-8859-1?Q?/Test/
Content/ParallelsTest1.doc?="
Content-Type: application/octet-stream

{\rtf1 .... File contents .... }
--XYTxYDxxYYYApEf--
```

The request contains the following information:

```
POST /put HTTP/1.1
```

POST is the HTTP method to use to transmit the data. HTTP/1.1 declares this as compatible with the HTTP 1.1 spec.

```
Pragma: no-cache
```

This indicates that the results of the request should not be cached. That is, every request will recalculate the results.

```
Content-Type: multipart/form-data; boundary=XYTxYDxxYYYApEf
```

The Content-Type: multipart/form-data indicates that what follows this header information is a multipart MIME data. The boundary statement lists the unique text string that will be used to separate the multiple parts of the MIME body.

```
Host: 192.168.1.147:8080
```

This is the IP or host name of the Typefi Publish server the client is talking to.

```
Content-Length: 10268
```

This is the total length in bytes of the message body. This includes the MIME boundary separators but does not include the HTTP header information.

```
Connection: Keep-Alive
```

This indicates that the client may reuse the authorization information for multiple requests.

```
--XYTxYDxxYYYApEf
```

The start of the message body. The start must include a CRLF and 2 dashes preceding the boundary string.

```
Content-Disposition: form-data; name="File"; filename="=?ISO-8859-1?Q?/Test/
Content/ParallelsTest1.doc?="
Content-Type: application/octet-stream
```

Content-Disposition describes this MIME part. This should be set to form-data. The name of the part is servlet specific. In this example, the filename attribute has been Q-Encoded. The Content-Type field describes the data that follows. In this case it is the contents of a file.

```
{\rtf1 .... File contents .... }
--XYTxYDxxYYYApEf--
```

After Content-Disposition is another blank line (a CRLF) and then the MIME part data. After the file contents, another boundary string is written preceded by the double-dashes. A double-dash at the end of the boundary string indicates the end of the MIME content.

Here is a sample with 2 mime parts:

```
POST /put HTTP/1.1
Pragma: no-cache
Content-Type: multipart/form-data; boundary=XYTxYDxxYYYApEf
Host: 192.168.1.147:8080
Content-Length: 10268
Connection: Keep-Alive

--XYTxYDxxYYYApEf
```

You must set the Content-Disposition to `form-data` and the `filename` parameter is required. It's best to Q-encode your filename to avoid issues with non-ASCII characters.

See *http://www.rfc-editor.org/rfc/rfc2047.txt*.

```
Content-Disposition: form-data; name="File"; filename="=?ISO-8859-1?Q?/Test/
Content/ParallelsTest1.doc?="
Content-Type: application/octet-stream

{\rtf1 .... File contents .... }
--XYTxYDxxYYYApEf
Content-Disposition: form-data; name="File"; filename="=?ISO-8859-1?Q?/Test/
Content/ParallelsTest2.doc?="
Content-Type: application/octet-stream

{\rtf1 .... File contents .... }
--XYTxYDxxYYYApEf—
```

Changelog

Below is a general overview of changes made to the Typefi Publish web services interface over time. This is not meant to be an svn or cvs commit log, rather a straightforward description of the overall changes you will notice when working our web services interface. Please report any problems or suggestions to ***support@typefi.com***.

5.1.2 (January 2013)

Supports Typefi Publish 5.1.x

- **MOD**: corrected example code for */runjob-xml* related to method of submitting XSL

5.1 (September 2011)

Supports Typefi Publish 5.1.x

- **MOD**: */runjob-xml* for running jobs with an applied XSLT (also supported for Typefi Publish 5.0.x)
- **MOD**: */listserver* has a new parameter (extendedInfo) to optionally return usage information for Typefi Publish

5.0 (February 2011)

Supports the initial release of Typefi Publish 5

- **NEW**: */runjob-raw* for running jobs that are not associated with a project
- **NEW**: */runjob-xml* can be used to run jobs with external XML and an XSL transform to CXML
- **NEW**: */exportproject* exports one or more projects
- **MOD**: */importproject* now supports multi-project archives (.tzips)
- **MOD**: */listproject* has been updated to include XSL information associated with the project
- **NEW**: a Java example project has been added to demonstrate calling */runjob*, */runjob-cxml*, and */runjob-raw*.

4.1 (February 2010)

Supports Typefi Publish 4.x.

- **MOD**: update some inaccuracies in the documentation

4.0 (June 2009)

Supports the initial release of Typefi Publish 4

- **MOD**: */runjob* and */runjob-cxml* now return the job folder name as part of the response
- **MOD**: */put* was incorrectly using MIME part name for the file name instead of the 'filename' attribute.
- **NEW**: a .NET example has been provided to demonstrate calling some of the servlets.

/authenticate

Description

As with all servlets, ***/authenticate*** uses Basic Authentication to acquire and check a user's credentials. While a call to ***/authenticate*** is not required in order to use other WSI servlets, it can be helpful for applications wishing to explicitly log users in.

Syntax

```
/authenticate
```

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/canceljob

Description

Cancels a submitted job.

Syntax

```
/canceljob?jobId=123
```

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/deleteasset

Description

Deletes the specified asset.

Syntax

```
/deleteasset {/My Project/Images/Chesire.jpg | ?assetId=123}
```

The asset can be identified by id or path.

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Examples

DELETE ASSET BY PATH:

```
/deleteasset/My Project/Images/Cheshire.jpg
```

DELETE ASSET BY ID:

```
/deleteasset?assetId=123
```

/deletejoboption

Description

Deletes a job option.

Syntax

```
/deletejoboption?
  {jobOptionId=123 |
  {jobOption=My Job Option&{projectId=123 | project=My Project}}}
```

Examples

DELETE JOB OPTION BY ID:

```
/deletejoboption?jobOptionId=123
```

DELETE JOB OPTION BY NAME:

```
/deletejoboption?jobOption=My Job Option&project=My Project
```

Response

SUCCESS

```
<response success="true">
  <jobOption id="123"/>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/deletemessage

Description

Deletes a message.

Syntax

```
/deletemessage?messageId=123
```

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/deletemessages

Description

Deletes messages. Any number of message ids may be provided.

Syntax

```
/deletemessages?messageIds=123,124,125
```

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/deleteproject

Description

Deletes a project. Only admin users can delete projects.

Syntax

```
/deleteproject?{projectId=”123” | project=”My Project”}
```

Response

SUCCESS

```
<response success=”true”>
  <project id=”123”/>
</response>
```

FAILURE

```
<response success=”false”>
  <error code=”<error code>”>
    <language type=”en-US”>Error Text</language>
  </error>
</response>
```

/exportproject

Description

Exports one or more projects.

Syntax

```
/exportproject
  {[/My Project] | [?projectIds=123,234,345...]}
  [&inline={true|false*}]
```

A single project can be exported by specifying its name in the URL path or by specifying a single project id. Multiple projects can be exported into a single archive by including a comma separated list of project ids. When exporting multiple projects the archive will be named `Projects.tzips`. Single project archives will be named after the project name.

Response

SUCCESS

The tzip as a MIME part.

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/get

Description

Gets an asset.

Syntax

```
/get
  {/My Project/Images/Cheshire.jpg | ?assetId=123}
  [&inline={true|false*}]
  [&checkout={true|false*}]
```

The asset can be identified by its path or by its id.

The `inline` parameter is used to set the Content Disposition MIME header (see *RFC 1806*) of the returned file. If `inline` is true, the Content-Disposition will be set to `inline`. If `inline` is false, the Content-Disposition will be set to `attachment`. This may be significant to the software initiating the request. A browser displaying a PDF will use the Content-Disposition to determine whether to show the PDF in the current window or to prompt the user to save it to the file system.

The `checkout` parameter, if true, will cause the asset to be checked out to the current user.

Examples

GET ASSET BY PATH:

```
/get/My Project/Images/My Image.jpg
```

GET ASSET BY ID:

```
/get?assetId=123
```

GET AND CHECKOUT ASSET:

```
/get/My Project/Images/My Image.jpg?checkout=true
```

GET AND CHECKOUT THE .PDF ASSET AND DISPLAY IT INLINE:

```
/get/My Project/Editions/My Edition/02 Aug 2005 16_47_31/Print.
pdf?inline=true&checkout=true
```

Response

SUCCESS

The asset as a MIME part named with the asset's file name.

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/getmessage

Description

Gets a message by its id. This is useful if you know the id of a message, perhaps from a ***/getmessages*** call, and wish to get the complete message information. Use ***/deletemessage*** to delete a message from the database.

Syntax

```
/getmessage
  ?messageId=123
  [&verbosity=01]
```

The `verbosity` parameter can be used to control how much information is returned. It is provided as a binary string where each bit represents an on/off switch. Bits are numbered from right to left and have the following meanings:

- bit 0: Show ID attribute.
- bit 1: Show child data elements.

The default verbosity is `11`.

Response

SUCCESS

```
<response success="true">
  <message id="1234">
    <!-- Note: always GMT - convert to user's timezone if displaying -->
    <date>Feb 27, 2006 12:21:41 GMT</date>
    <subject>
      <language type="en-US">Message subject here.</language>
    </subject>
    <body>
      <language type="en-US">Message text here.</language>
```

```
    </body>
    <type>{info | warn | error}</type>
    <destination>{console | other}</destination>

    <!-- <refresh> indicates whether or not the Console should attempt a page
refresh before or after showing the message. <refreshBefore> indicates when
the refresh should occur (true = before, false = after). <refreshed> indicates
whether or not the Console has already been refreshed. -->
    <refresh>{true | false}</refresh>
    <refreshBefore>{true | false}</refreshBefore>
    <refreshed>{true | false}</refreshed>

    <!-- Whether or not the message has been sent to the user. -->
    <sent>{true | false}</sent>
  </message>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/getmessages

Description

Get messages for the authenticated user. Messages will stay in the database until they are actively deleted (see */deletemessages*).

Syntax

```
/getmessages
  [?type={info | warn | error | all*}]
  [&dest={console | other* | all}]
  [&verbosity=01]
```

The `type` parameter can be used to narrow retrieved messages by their type. See */sendmessage* for type descriptions.

`dest` narrows the messages by their destination. See */sendmessage* for a description of `console` and `other` destination values. `all` gets messages for both destinations. Typically, an application will omit this parameter (defaults to `other`) to fetch just those messages not intended for the Console.

The `verbosity` parameter is identical to that used with the */getmessage* servlet. Please see the description from that servlet.

Response

SUCCESS

```
<response success="true">
  <messages>
    <!-- Stucture of <message> elements is identical to that for /getmessage.
Please see the /getmessage servlet. -->
    <message id="1234">
      ...
    </message>
```

```
    ...
  </messages>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/importproject

Description

Imports one or more projects.

Syntax

```
/importproject
  [{/My Project/Templates/BusinessLetter.tzip | ?assetId=123}]
  [&name=My New Project]
  [&descr=Project Description]
  [&errorIfExists={true|false*}]
  [&verbosity=0101010]
<File MIME Part: tzip>
```

The .tzip file can either be supplied as a MIME part, identified by a path or by an asset id. The name and description of the project is taken from the properties within the tzip file. A multiple project archive (a .tzips file) can also be imported. When importing a single project, `name` and `descr` can be specified to override the values inside the tzip file. `Name` and `descr` are ignored when importing a multi-project archive.

The `verbosity` parameter is used to control how much information about the imported project is returned. Minimally, the project id and name are returned. This is the default behavior (verbosity="0000001"). See the description of the `verbosity` parameter in the ***/listproject*** servlet for a complete list of verbosity controls.

If `errorIfExists` is true then the servlet will return an error if a project already exists with the specified name. If false projects will be given unique names in the case of a conflict.

There are several ways to add .tzip files to the repository for reference by path or asset id. Using Add Files will place them in the Images directory. Adding them directly to the filestore and using ***/synchfs*** allows them to be added to any directory. The ***/put*** servlet can also be used to place them anywhere in the repository.

Response

SUCCESS

```
<response success="true">
  <project id="123" name="My New Project"/>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listasset

Description

Returns details for the specified asset.

Syntax

```
/listasset
  {/My Project/Images/Chesire.jpg | ?assetId=123}
  [&rawSize={true|false*}]
```

The asset can be identified by id or path.

The `rawSize` parameter only applies to .sxml and .cxml assets. If false or omitted, the size reported for the asset (<size>) will be the size of the asset after cross-references are resolved. This is the default behavior. If `rawSize` is true, the size of the file before cross-references are resolved will be used.

Response

SUCCESS

```
<response success="true">
  <asset id="123" path="/My Project/Content/Sections/sadf23fawefa1382.sxml">
    <!-- applies to sections (.sxml) only -->
    <position>2</position>
    <owner id="123" username="admin"/>
    <creationDate>2006-05-17 01:07:42.155 GMT</creationDate>
    <modifiedDate>2006-05-17 02:17:14.301 GMT</modifiedDate>
    <checkedOut val="{true | false}">
      <!-- only present if val = true -->
      <user id="123" username="admin"/>
    </checkedOut>
    <readOnly val="{true | false}">
      <!-- only present if val = true -->
```

```
      <user id="123" username="admin"/>
    </readOnly>
    <size>23100</size>
  </asset>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Examples

LIST ASSET BY PATH:

```
/listasset/My Project/Images/Cheshire.jpg
```

LIST ASSET BY ID:

```
/listasset?assetId=123
```

LIST ASSET BY ID AND GETRAW SIZE INSTEAD OF THE SIZE AFTER CROSS-REFERENCES ARE RESOLVED (.SXML AND .CXML ONLY):

```
/listasset?assetId=123&rawSize=true
```

/listassets

Description

Lists the assets at the provided project path.

Syntax

```
/listassets
  /My Project/Images/*.jpg
  [?inclSubdirs={true|false*}]
  [&rawSize={true|false*}]
  [&verbosity=1]
```

The provided path indicates where to search for assets. The `*` wildcard may be used to indicate a sequence of zero or more unspecified characters.

`inclSubdirs` specifies whether or not to include subdirectories when searching for assets.

The `rawSize` parameter only applies to .sxml and .cxml assets. If false or omitted, the size reported for each asset (<size>) will be the size of the asset after cross-references are resolved. This is the default behavior. If `rawSize` is true, the size of the file before cross-references are resolved will be used.

The `verbosity` parameter can be used to control how much information is returned. It is provided as a binary string where each bit represents an on/off switch. Bits are numbered from right to left and have the following meanings:

- bit 0: Expand assets.

The default verbosity is `1`.

Job assets are ignored by this servlet to prevent against potentially huge amounts of return data. See ***/listjobs*** and ***/listjob*** for querying jobs and their assets.

Only assets belonging to projects that the current user is a member of are returned.

Response

SUCCESS

```
<response success="true">
  <assets>
    <!-- Stucture of <asset> elements is identical to that for /listasset. Please
see the /listasset servlet. -->
    <asset id="123" path="/My Project/Content/Sections/sadf23fawefa1382.sxml">
      ...
    </asset>
    ...
  </assets>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Examples

FIND ALL .GIF ASSETS IN THE IMAGES SUB-FOLDER OF 'MY PROJECT'

```
/listassets/My Project/Images/*.gif
```

FIND ALL .INDD ASSETS IN ANY PROJECT

```
/listassets/*/Templates/*.indd
```

FIND ALL .DOC ASSETS IN 'MY PROJECT'

```
/listassets/My Project/*.doc?inclSubdirs=true
```

FIND ALL CONTENT ASSETS IN 'MY PROJECT'

```
/listassets/My Project/Content/*
```

FIND A SPECIFIC ASSET IN 'MY PROJECT' (ALSO SEE */LISTASSET*)

```
/listassets/My Project/Images/icon.jpg
```

/listdocs

Description

Lists the docs in the Content folder of a given project.

Syntax

```
/listdocs
  ?{projectId=123 | project=My Project}
  [&pattern=<file pattern>]
  [&verbosity=1]
```

`pattern`, if provided, can be used to match the filenames to return. The `*` wildcard may be used to indicate a sequence of zero or more unspecified characters. For example, `*.rtf` would return only files with an .rtf extension.

The `verbosity` parameter is identical to that used with the ***/listassets*** servlet. Please see the description for that servlet.

Response

Servlet output is identical to that for ***/listassets***. Please see that servlet.

Examples

FIND ALL DOCUMENTS IN 'MY PROJECT'

```
/listdocs?project=My Project
```

FIND ONLY .RTF DOCUMENTS IN 'MY PROJECT'

```
/listdocs?project=My Project&pattern=*.rtf
```

/listengine

Description

Lists information about the specified engine.

Syntax

```
/listengine
  ?{engineId=123 | engine=Sample Engine}
  [&verbosity=01010]
```

The engine to list is specified by its id or name. Only engines that the authenticated user has permissions to use can be listed. Administrative users are exempted from this rule and can list any engine.

The `verbosity` parameter can be used to control how much information is returned. It is provided as a binary string where each bit represents an on/off switch. Bits are numbered from right to left and have the following meanings:

- bit 0: Show id and name attributes.
- bit 1: Show child data elements.
- bit 2: Expand reserved user.
- bit 3: Expand engine type.
- bit 4: Expand current job, if any.

The default verbosity is `00011`.

Response

SUCCESS

```
<response success="true">
  <engine id="123" name="My Engine">
    <host>127.0.0.1</host>
    <port>8470</port>
```

```
    <designerPort>8471</port>
    <engineType id="123" name="Generic"/>
    <reservedUser id="123" username="joe"/>
    <filestorePath>C:/Program Files/Typefi/Publish/Filestore</filestorePath>
    <enabled>true</enabled>

    <!-- The current job, if any. The existence of this element can also be used
to determine whether or not the engine is busy. -->
    <currentJob id="123"/>
  </engine>
  ...
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listengines

Description

Lists information about engines.

Syntax

```
/listengines
  [?{engineTypeId=123 | engineType=Generic}]
  [&{jobOptionId=123 |
  {jobOption=My Job Option&{projectId=123 | project=My Project}}}]
  [&enabledOnly={true | false*}]
  [&showAll={true | false*}]
  [&verbosity=01010]
```

If no parameters are supplied, */listengines* will return all engines that the authenticated user has permission to use. Alternatively, if an engine type is specified, only engines of that type will be returned. If a job option is specified, only engines capable of running that job option will be returned. Specifying both an engine type and a job option further restricts the results. If `enabledOnly` is true, only enabled engines are returned. Finally, the `showAll` parameter is honored for administrative users only and allows all engines to be returned regardless of user permissions.

The `verbosity` parameter is identical to that used with the */listengine* servlet. Please see the description for that servlet.

Response

SUCCESS

```
<response success="true">
  <engines>
    <!-- Stucture of <engine> elements is identical to that for /listengine.
Please see the /listengine servlet. -->
```

```
    <engine id="123" name="My Engine"/>
      ...
    </engine>
    ...
  </engines>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Examples

AS A USER, LIST ALL ACCESSIBLE ENGINES

```
/listengines
```

AS A USER, LIST ALL ACCESSIBLE ENGINES FOR A GIVEN JOB OPTION

```
/listengines?project=My Project&jobOption=My Job Option
```

AS AN ADMIN USER, LIST ALL 'GENERIC' ENGINES

```
/listengines?engineType=Generic&showAll=true
```

AS AN ADMIN USER, LIST ALL ENABLED ENGINES

```
/listengines?enabledOnly=true&showAll=true
```

/listenginetype

Description

Lists information about the specified engine type.

Syntax

```
/listenginetype
  ?{engineTypeId=123 | engineType=My Engine Type}
  [&verbosity=010]
```

The engine type to list is specified by its id or name.

The `verbosity` parameter can be used to control how much information is returned. It is provided as a binary string where each bit represents an on/off switch. Bits are numbered from right to left and have the following meanings:

- bit 0: Show id and name attributes.
- bit 1: Show child data elements.
- bit 2: Expand PDF presets.

The default verbosity is `011`.

Response

SUCCESS

```
<response success="true">
  <engineType id="123" name="My Engine Type">
    <descr>Engine Type Description</descr>
    <pdfPresets>
      <pdfPreset id="123" name="[High Quality Print]"/>
      <pdfPreset id="124" name="[Smallest File Size]"/>
    </pdfPresets>
  </engineType>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listenginetypes

Description

Lists information about engine types.

Syntax

```
/listenginetypes[?verbosity=010]
```

The `verbosity` parameter is identical to that used with the ***/listenginetype*** servlet. Please see the description for that servlet.

Response

SUCCESS

```
<response success="true">
  <engineTypes>
    <engineType id="123" name="My Engine Type">
      <descr>Engine Type Description</descr>
      <pdfPresets>
        <pdfPreset id="123" name="[High Quality Print]"/>
        <pdfPreset id="124" name="[Smallest File Size]"/>
      </pdfPresets>
    </engineType>
    ...
  </engineTypes>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listimages

Description

Lists the images in the Images folder of a given project.

Syntax

```
/listimages
  ?{projectId=123 | project=My Project}
  [&pattern=<file pattern>]
  [&verbosity=1]
```

`pattern`, if provided, can be used to match the filenames to return. The `*` wildcard may be used to indicate a sequence of zero or more unspecified characters. For example, `*.jpg` would return only files with a .jpg extension.

The `verbosity` parameter is identical to that used with the ***/listassets*** servlet. Please see the description for that servlet.

Response

Servlet output is identical to that for ***/listassets***. Please see that servlet.

Examples

FIND ALL IMAGES IN 'MY PROJECT'

```
/listimages?project=My Project
```

FIND ALL .JPG IMAGES IN 'MY PROJECT'

```
/listimages?project=My Project&pattern=*.jpg
```

/listjob

Description

Lists information about a job.

Syntax

```
/listjob
  ?jobId=123
  [&rawSize={true|false*}]
  [&verbosity=0101010]
```

The `rawSize` parameter only applies to content.cxml assets and only when the verbosity indicates that output assets should be expanded. If false or omitted, the size reported for content.cxml (<size>) will be the size of the asset after cross-references are resolved. This is the default behavior. If `rawSize` is true, the size of the file before cross-references are resolved will be used.

The `verbosity` parameter can be used to control how much information is returned. It is provided as a binary string where each bit represents an on/off switch. Bits are numbered from right to left and have the following meanings:

- bit 0: Show ID attribute.
- bit 1: Show child data elements.
- bit 2: Expand owner.
- bit 3: Expand project.
- bit 4: Expand edition.
- bit 5: Expand engine.
- bit 6: Expand output's assets.

The default verbosity is `0000011`.

Response

SUCCESS

```
<response success="true">
  <job id="123">
    <name>12 May 2006 10_35_21</name>
    <label>My Job</label>
    <type>{compose | roundtrip}</type>
    <environ>{publish | print}</environ>
    <transient>{true | false}</transient>
    <owner id="123" username="admin"/>
    <project id="123" name="My Project"/>
    <jobOption id="123" name="My Job Option"/>
    <status>
      <state>
        {initialized | pending | inprogress | done | error | cancelled}
      </state>

      <!-- Only returned if there is a message. -->
      <message>
        <language type="en-US">
          Status Message Text
        </language>
      </message>
      <percentComplete>70</percentComplete>

      <!-- Only returned when job is "pending". Indicates the job's position in
the wait queue.-->
      <positionInQueue>5</positionInQueue>

      <!-- Only returned when job is complete. Indicates the severity of the most
serious entry in this job's Engine log. -->
      <logSeverity>{info|warn|error}</logSeverity>
    </status>

    <!-- waitTime indicates how long job waited in queue -->
    <waitTime>5s</waitTime>
    <submitTime>2006-05-17 10:25:03.84 GMT </submitTime>
    <startTime>2006-05-17 10:25:22.100 GMT</startTime>
    <endTime>2006-05-17 10:25:34.364 GMT</endTime>

    <!-- Engine used to run the job. -->
    <engine id="123" name="My Engine"/>

    <!-- Only sent once the job is complete. -->
    <output>
```

```
      <asset id="123" path="/My Project/Editions/My Job Option/02 Aug 2005
16_47_31/Print.pdf"/>
      <asset id="124" path="/My Project/Editions/My Job Option/02 Aug 2005
16_47_31/Print.log"/>
      <asset id="125" path="/My Project/Editions/My Job Option/02 Aug 2005
16_47_31/Print.indd"/>
      <asset id="126" path="/My Project/Editions/My Job Option/02 Aug 2005
16_47_31/content.cxml"/>
    </output>
  </job>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listjoboption

Description

Lists information about a job option.

Syntax

```
/listjoboption?
  {jobOptionId=123 |
  {jobOption=My Job Option&{projectId=123 | project=My Project}}}
  [&verbosity=010101]
```

The `verbosity` parameter can be used to control how much information is returned. It is provided as a binary string where each bit represents an on/off switch. Bits are numbered from right to left and have the following meanings:

- bit 0: Show identifying attributes.
- bit 1: Show child data elements.
- bit 2: Expand owner.
- bit 3: Expand engine type.
- bit 4: Expand engine.
- bit 5: Expand content sections.

The default verbosity is `00001`".

Response

SUCCESS

```
<response success="true">
  <jobOption id="123" name="My Job Option" projectId="123" projectName="My
Project">
    <descr>My Job Option's Description</descr>
    <owner id="123" username="admin"/>
    <creationDate>2006-05-17 03:12:29.120 GMT</creationDate>
```

```
    <engineSelection>
      <overrideProject>{true | false}</overrideProject>
      <engineSpec>
        <engineType id="123" name="My Engine Type"/>
        <engine id="123" name="My Engine"/> <!-- only exists if a specific engine
is identified -->
      </engineSpec>
    </engineSelection>
    <engineCfg>
      <maxTimePerPage>30</maxTimePerPage>
      <maxLayoutsPerPage>100</maxLayoutsPerPage>
    </engineCfg>
    <createPdf presetId="123" presetName="[High Quality Print]">
      {true | false}
    </createPdf>
    <createSingleDoc>{true | false}</createSingleDoc>
    <firstPageNum>1</firstPageNum>
    <xrefs>

      <!-- If present, replace the xref text with that provided. Otherwise, keep
the last known text. -->
      <handleUnresolvable>
        <replaceText>My Replacement Text</replaceText>
      </handleUnresolvable>
    </xrefs>
    <labelJob fieldId="123" fieldName="DocumentTitle">
      {true | false}
    </labelJob>
    <conditions>
      <condition name="US English" ID="123">
        <descr>Condition Description</descr>
        <value>{true | false}</value>
      </condition>
      ...
    </conditions>
    <template id="123" name="My Template"/>
    <content>
      <section id="123"/>
      ...
    </content>
    <storeJob>{true | false}</storeJob>
    <scriptOptions>

      <!-- Whether or not these options should be used to override the System
defaults. -->
```

```
      <overrideSystem>{true | false}</overrideSystem>

      <!-- Script set to use if overriding. -->
      <scriptSet>
        <params>My Params</params>
        <script eventId="job.start">MyJobStartScript</script>
        <script eventId="job.end">MyJobEndScript</script>
        <script eventId="book.start">MyBookStartScript</script>
        <script eventId="book.end">MyBookEndScript</script>
        <script eventId="document.start">MyDocumentStartScript</script>
        <script eventId="document.end">MyDocumentEndScript</script>
        <script eventId="section.start">MySectionStartScript</script>
        <script eventId="section.end">MySectionEndScript</script>
        <script eventId="spread.start">MySpreadStartScript</script>
        <script eventId="spread.end">MySpreadEndScript</script>
        <script eventId="page.start">MyPageStartScript</script>
        <script eventId="page.end">MyPageEndScript</script>
        <script eventId="roundtrip.start">MyRoundtripStartScript</script>
        <script eventId="roundtrip.end">MyRoundtripEndScript</script>
        <!-- There is no spill.start event. -->
        <script eventId="spill.end">MySpillEndScript</script>
      </scriptSet>
    </scriptOptions>
  </jobOption>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listjoboptions

Description

Lists information about all job options for the project.

Syntax

```
/listjoboptions
  ?{projectId=123 | project=My Project}
  [&verbosity=010101]
```

The `verbosity` parameter is identical to that used with the ***/listjoboption*** servlet. Please see the description for that servlet.

Response

SUCCESS

```
<response success="true">
  <jobOptions>
    <!-- Stucture of <jobOption> elements is identical to that for
/listjoboption. Please see the /listjoboption servlet. -->
    <jobOption id="123" name="My Job Option" projectId="123" projectName="My
Project">
      ...
    </jobOption>
    ...
  </jobOptions>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
```

```
</response>
```

/listjobs

Description

Lists jobs. Jobs can be narrowed by project, job option, and/or user. They can also optionally include complete, incomplete, roundtrip, compose, publish, and print jobs. The order of returned jobs depends on the `order` parameter and defaults to reverse chronological.

Syntax

```
/listjobs?
  [{projectId=123 | project=My Project} |
  {jobOptionId=123 | {jobOption=My Job Option&{projectId=123 | project=My
Project}}}]
  [&{userId=123 | user=admin}]
  [&inclComplete={true*|false}]
  [&inclIncomplete={true*|false}]
  [&inclRoundtrip={true|false*}]
  [&inclCompose={true*|false}]
  [&inclPublish={true*|false}]
  [&inclPrint={true*|false}]
  [&pageSize=10*]
  [&page=1*]
  [&order={chrono | reverse chrono*}]
  [&rawSize={true|false*}]
  [&verbosity=0101010]
```

Optional project, job option, and user identifiers are used to narrow the fetched jobs by project, job option, and/or user, respectively. These can be further narrowed by the `include` parameters:

- `inclComplete`: Whether or not to include complete (done) jobs.
- `inclIncomplete`: Whether or not to include incomplete jobs.
- `inclRoundtrip`: Roundtrip jobs are jobs that run during *Update Repository* operations from an existing job folder. These are typically hidden from the user and are not returned by default.

- `inclCompose`: Whether or not to include composition jobs. These are typically returned.
- `inclPublish`: Jobs can be run in the *publish* environment or the *print* environment. The publish environment refers to jobs run via the web Console.
- `inclPrint`: Print jobs refer to jobs run via one of the ***/runjob*** servlets, including jobs run from Typefi Writer.

`page` and `pageSize` parameters are used to limit the number of results returned, since there may be thousands of jobs in the system, which would be very slow to return. The throttling of results is based on the concept of pages. `pageSize` specifies the number of results to return per page and `page` specifies which page to return. For example, a `pageSize` of 10 and a `page` of 2, will return results 11-20. 0 is a special value for `page` which indicates that the last page of results should be returned. If `pageSize` is not supplied, it defaults to 10. Likewise, `page` defaults to 1. Beware that the larger the `pageSize`, the longer it will take for the query to return and the greater the load is on the Typefi Publish server. It is recommended to keep `pageSize` less than or equal to 50.

Job results are always ordered by the job's submission timestamp. The `order` parameter can be used to specify whether the ordering should be chronological or reverse chronological (the default).

The `rawSize` and `verbosity` parameters are identical to that used with the ***/listjob*** servlet. Please see the description for that servlet.

Examples

LIST ALL JOBS, EXCLUDING ROUNDTRIP; SHOW RESULTS 201-250:

```
/listjobs?pageSize=50&page=5
```

LIST ALL JOBS, INCLUDING ROUNDTRIP. SHOW RESULTS 11-20:

```
/listjobs?inclRoundtrip=true&pageSize=10&page=2
```

LIST ALL JOBS FOR A PROJECT. SHOW RESULTS 17-24:

```
/listjobs?project=My Project&pageSize=8&page=3
```

LIST ALL JOBS FOR A JOB OPTION. SHOW FIRST 20 RESULTS IN CHRONOLOGICAL ORDER.:

```
/listjobs?jobOption=My Job Option&project=My Project&pageSize=20&order=chrono
```

LIST ALL JOBS OWNED BY A USER. SHOW LAST 20 RESULTS:

```
/listjob?userId=123&pageSize=20&page=0
```

LIST ALL NON-PRINT JOBS FOR A JOB OPTION, OWNED BY A USER. SHOW FIRST 10 RESULTS:

```
/listjob?jobOptionId=123&user=admin&inclPrint=false
```

Response

SUCCESS

```
<response success="true">
  <jobs>
    <!-- Stucture of <job> elements is identical to that for /listjob. Please see
the /listjob servlet. -->
    <job id="123">
      ...
    </job>
    ...
  </jobs>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listproject

Description

Lists information about a project.

Syntax

```
/listproject
  ?{projectId=123 | project=My Project}
  [&inclJobOptions={true | false*}]
  [&inclDocs={true | false*}]
  [&verbosity=0101010]
```

If `inclJobOptions` is provided and is true, all job options in the project are included in the response. Likewise, if `inclDocs` is provided and is true, all documents in the project are included in the response.

The `verbosity` parameter can be used to control how much information is returned. It is provided as a binary string where each bit represents an on/off switch. Bits are numbered from right to left and have the following meanings:

- bit 0: Show id and name attributes.
- bit 1: Show child data elements.
- bit 2: Expand owner.
- bit 3: Expand engine spec.
- bit 4: Expand users (members).
- bit 5: If including job options, expand them.
- bit 6: If including docs, expand them.
- bit 7: Expand XSL information.

The default verbosity is `0000011`.

Response (inclJobOptions = false, inclDocs = false)

SUCCESS

```
<response success="true">
  <project id="123" name="My Project">
    <descr>My Project's Description</descr>
    <owner id="123" username="admin"/>
    <creationDate>2006-05-17 03:08:44.229 GMT</creationDate>
    <engineSpec>
      <engineType id="123" name="My Engine Type"/>
      <engine id="123" name="My Engine"/> <!-- only exists if a specific engine is
identified -->
    </engineSpec>
    <fields>
      <field id="123" name="BookTitle">Alice's Adventures in Wonderland</field>
      ...
    </fields>
    <conditions>
      <condition name="US English" ID="123">
        <descr>US English Description</descr>
      </condition>
      ...
    </conditions>

    <!-- membership list -->
    <users>
      <user id="124" username="joe"/>
      ...
    </users>

    <!-- XML transformation information used by this project. -->
    <xsl>
      <catalog name="My XML Catalog.xml"/>
      <transforms>
        <transform name="First Transform.xsl"/>
        <transform name="Second transform.xsl"/>
      </transforms>
    </xsl>
  </project>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
```

```
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Response (inclJobOptions = true, inclDocs = true)

SUCCESS

```
<response success="true">
  <project id="123" name="My Project">
    <descr>My Project's Description</descr>
    <owner id="123" username="admin"/>
    <creationDate>2006-05-17 03:08:44.229 GMT</creationDate>
    <engineSpec>
      <engineType id="123" name="My Engine Type"/>
      <engine id="123" name="My Engine"/>
    </engineSpec>
    <fields>
      <field id="123" name="BookTitle">Alice's Adventures in Wonderland</field>
      ...
    </fields>
    <conditions>
      <condition name="US English" ID="123">
        <descr>US English Description</descr>
      </condition>
      ...
    </conditions>

    <!-- membership list -->
    <users>
      <user id="124" username="joe"/>
      ...
    </users>

    <!-- job options -->
    <jobOptions>
      <jobOption id="123" name="My Job Option" projectId="123" projectName="My
Project"/>
      <jobOption id="124" name="My 2nd Job Option" projectId="123"
projectName="My Project"/>
    </jobOptions>

    <!-- docs -->
    <docs>
      <doc id="123" path="/My Project/Content/My Doc.doc"/>
    </docs>
```

```
  </project>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listprojects

Description

Lists information about projects that the authenticated user is a member of.

Syntax

```
/listprojects
  [?inclJobOptions={true | false*}]
  [&inclDocs={true | false*}]
  [&verbosity=0101010]
```

If `inclJobOptions` is provided and is true, all job options in each project are included in the response. Likewise, if `inclDocs` is provided and is true, all docs in each project are included in the response.

The `verbosity` parameter is identical to that used with the ***/listproject*** servlet. Please see the description for that servlet.

Response

SUCCESS

```
<response success="true">
  <projects>
    <!-- Stucture of <project> elements is identical to that for /listproject.
Please see the /listproject servlet. -->
    <project id="123" name="My Project">
      ...
    </project>
    ...
  </projects>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listsections

Description

Lists the sections in the Content/Sections folder for a given project.

Syntax

```
/listsections
  ?{projectId=123 | project=My Project}
  [&rawSize={true|false*}]
  [&verbosity=1]
```

If `rawSize` is false or is omitted, the size reported for each .sxml asset (<size>) will be the size of the asset after cross-references are resolved. This is the default behavior. If `rawSize` is true, the size of the file before cross-references are resolved will be used.

The `verbosity` parameter is identical to that used with the ***/listassets*** servlet. Please see the description for that servlet.

Response

Servlet output is identical to that for ***/listassets***. Please see that servlet.

/listserver

Description

Lists information about the Typefi Publish server version and version of the web service interface (WSI).

Syntax

```
/listserver
  [?extendedInfo=[true|false*]]
```

The optional parameter `extendedInfo` determines if the <systemInfo> element will be returned. The default is false.

Response

SUCCESS

```
<?xml version="1.0" encoding="UTF-8"?>
<response success="true">
  <server version="5.1" wsiVersion="5">
    <license licensedTo="host.example.com" expirationDate="None"/>
  </server>
  <terms>
    <term name="Projects">
      <language type="en-US">Projects</language>
    </term>
    <term name="projects">
      <language type="en-US">projects</language>
    </term>
    <term name="Project">
      <language type="en-US">Project</language>
    </term>
    <term name="project">
      <language type="en-US">project</language>
```

The <terms> section is deprecated and will be removed in a future release. These values are no longer customizable.

This servlet does not require authentication.

```
    </term>
    <term name="Editions">
      <language type="en-US">Job Options</language>
    </term>
    <term name="editions">
      <language type="en-US">job options</language>
    </term>
    <term name="Edition">
      <language type="en-US">Job Option</language>
    </term>
    <term name="edition">
      <language type="en-US">job option</language>
    </term>
  </terms>

  <!-- systemInfo is only returned if 'extendedInfo=true' -->
  <systemInfo timestamp="2011-09-09 13:00:05 +0000">
    <memory>
      <committed>534904832</committed>
      <free>940125488</free>
      <totalUsedPercent>9</totalUsedPercent>
      <totalFreePercent>91</totalFreePercent>
      <committedUsedPercent>18</committedUsedPercent>
      <committedFreePercent>82</committedFreePercent>
      <heap init="536870912" used="93901520" max="1034027008"
committed="534904832"/>
      <nonHeap init="19136512" used="38587832" max="117440512"
committed="38993920"/>
    </memory>
    <serverUsage>
      <projects count="18"/>
      <assets count="2536"/>
      <jobs count="69"/>
      <users count="20"/>
    </serverUsage>
    <licenseReport timestamp="Sep 09, 2011 14:00:05 BST">
      <server name="clamhost" ip="1.1.1.1" mac="FF:FF:FF:FF:FF:FF">
        <licensedTo>ACME Corporation</licensedTo>
        <expirationDate>None</expirationDate>
        <issueDate>2011-01-24</issueDate>
      </server>
      <summary>
        <writerLicenses>11</writerLicenses>
        <designerLicenses>17</designerLicenses>
        <engineLicenses>1</engineLicenses>
```

```
        <pagesTypeset>18285</pagesTypeset>
      </summary>

      <!-- List of client machines -->
      <clients>
        <client id="cde9e97f-b14a-11e0-b875-5b32c718d643" name="Writer-Clam"
ip="1.1.1.2" mac="FF:FF:FF:FF:40:D8">
          <licenses>
            <license type="writer" version="5.0.1.34"/>
          </licenses>
          <status>
            <lastSuccess>Jul 21, 2011 13:22:54 BST</lastSuccess>
          </status>
        </client>
        <client id="b53f232d-a48a-11df-ac3c-cf5dc921af8f" name="Designer-Clam"
ip="10.0.1.11" mac="FF:FF:FF:09:99:0C">
          <licenses>
            <license type="designer" version="5.1.0.373"/>
          </licenses>
          <status>
            <lastSuccess>Aug 19, 2011 18:57:45 BST</lastSuccess>
          </status>
        </client>
        . . .
      </clients>
      <engines>
        <engine name="Enging-Clam" host="127.0.0.1" port="8470"
pagesTypeset="18285"/>
      </engines>
    </licenseReport>
    <volumes>
      <filestore total="214745214976" free="183749390336" freePercent="86"/>
      <database total="214745214976" free="183749390336" freePercent="86"/>
      <server total="37474004992" free="15618314240" freePercent="42"/>
    </volumes>

    <!-- Java system variables depend on operating system. -->
    <java>
      <property name="awt.toolkit">sun.awt.windows.WToolkit</property>
      <property name="java.version">1.6.0_11</property>
    </java>

    <!-- Environment variables defined on the server operating system. -->
    <environment>
      <property name="=C:">C:\Program Files (x86)\Typefi\Publish\Server</property>
      <property name="ALLUSERSPROFILE">C:\ProgramData</property>
```

```
      <property name="CommonProgramFiles(x86)">C:\Program Files (x86)\Common
Files</property>
    </environment>

    <!-- Database variables depend on the database type and configuration. -->
    <database>
      <property name="database_host">localhost</property>
      <property name="database_jdbc_url">jdbc:hsqldb:hsql://localhost:9001/tpss</
property>
      <property name="database_name">tpss</property>
      <property name="database_port">9001</property>
      <property name="database_type">HSQLDB</property>
      <property name="hsqldb_path">D:/Typefi/Publish/Database</property>
    </database>
  </systemInfo>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/listtemplates

Description

Lists the templates in the Templates folder of a given project.

Syntax

```
/listtemplates
  ?{projectId=123 | project=My Project}
  [&pattern=<file pattern>]
  [&verbosity=1]
```

`pattern`, if provided, can be used to match the filenames to return. The `*` wildcard may be used to indicate a sequence of zero or more unspecified characters. For example, `*.txml` would return only files with a .txml extension.

The `verbosity` parameter is identical to that used with the ***/listassets*** servlet. Please see the description for that servlet.

Response

Servlet output is identical to that for ***/listassets***. Please see that servlet.

Examples

FIND ALL INDESIGN TEMPLATES IN 'MY PROJECT'

```
/listtemplates?project=My Project
```

FIND ALL TEMPLATE XML (.TXML) FILES IN 'MY PROJECT'

```
/listtemplates?project=My Project&pattern=*.txml
```

/markmessage

Description

Marks a message as being sent or refreshed.

This servlet is used internally by the Console and is only applicable to applications that make use of messages. It is furnished as part of the WSI in the event that it may be useful to an integration but is not expected to be heavily used.

Syntax

```
/markmessage
  ?messageId=123
  [&sent={true|false}]
  [&refreshed={true|false}]
```

The `messageId` identifies the message.

The `sent` parameter is used to indicate whether or not the message has been sent to the user.

The `refreshed` parameter is used to set the message's refreshed flag. This is useful for messages that request that the Console attempt a browser refresh. In the case of a browser refresh being done before showing the user the message (see ***/sendmessage's*** `refresh` ***and*** `refreshBefore` ***parameters***), the Console will use this parameter to indicate that it has already done the refresh. That way, the next time the Console encounters the message it will know not to do the refresh again and can just display the message.

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/put

Description

Puts an asset into the repository.

Syntax

```
/put
  ?checkin={true|false*}
  &quiet={true|false*}
 <File MIME Part: File>  [1 or more File parts]
 <File MIME Part: File>
```

The `checkin` parameter indicates whether or not the asset should be checked in. This only makes sense if the asset already exists in the repository. The `quiet` parameter controls whether or not a message is sent to the Console user (the user used to authenticate this servlet call) when the file upload is complete. Normally, this parameter will be set to false (or left off completely).

`<File MIME Part: File>`

There can be any number of file MIME parts. Each part contains a file to be uploaded to the repository. The file will be uploaded to the path specified by the `filename` parameter of the "Content-Disposition" header field (see *RFC 1806*).

To allow for non-ascii characters in file paths, the value supplied for the `filename` parameter may be encoded as per the "Q" encoding scheme described in *RFC 1522*. The Apache Java class org.apache.commons.codec.net.QCodec may be helpful for this purpose.

If your file paths contain only ascii characters, there is no need to encode them.

Note that the Content-Type must be set to form-data and a filename property is required for each part.

To maintain backward compatibility, Typefi Publish 5 and later will check if the part name contains a `/` (a directory separator). If so, it will use the part name for the filename. If not, the `filename` parameter will be used.

It is highly recommended that you use the filename parameter to specify the file name.

SAMPLE MIME CONTENT

```
POST /put HTTP/1.1
Pragma: no-cache
Content-Type: multipart/form-data; boundary=XYTxYDxxYYYApEf
Host: 192.168.1.147:8080
Content-Length: 10268
Connection: Close

--XYTxYDxxYYYApEf
Content-Disposition: form-data; name="File"; filename="=?ISO-8859-1?Q?/Test/
Content/ParallelsTest1.doc?="
Content-Type: application/octet-stream

{\rtf1 .... File contents .... }
--XYTxYDxxYYYApEf--
```

Files are only uploaded to projects that the authenticated user is a member of.

Response

SUCCESS

On success, descriptions of the uploaded assets are returned.

```
<response success="true">
  <assets>
    <asset id="123" path="/My Project/Content/Images/My Image.jpg">
      <owner id="123" username="admin"/>
      <creationDate>2006-05-17 01:07:42.155 GMT</creationDate>
      <modifiedDate>2006-05-17 02:17:14.301 GMT</modifiedDate>
      <checkedOut val="{true | false}">
        <user id="123" username="admin"/> <!-- only present if val = true -->
      </checkedOut>
      <readOnly val="{true | false}">
        <user id="123" username="admin"/> <!-- only present if val = true -->
      </readOnly>
      <size>23100</size>
    </asset>
    ...
  </assets>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
```

```
  </error>
</response>
```

/releasejob

Description

Notifies Typefi Publish that a job's output is no longer needed. This allows Typefi Publish to cleanup the job and its assets from the database and filestore, thus reducing clutter. As an example, an application may call ***/releasejob*** after it has retrieved the job's output PDF from the server.

This only applies to jobs that are not being permanently stored on Typefi Publish as indicated by the Store Typefi Print Jobs job option setting (also see <storeJob> in ***/runjob***'s job params).

Syntax

```
/releasejob?jobId=123
```

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/roundtripjob

Description

/roundtripjob takes an output .indd file (as generated by */runjob*, e.g.), extracts its content as .cxml, and optionally uses that .cxml to update the sections in the repository. This process is called round tripping.

Syntax

```
/roundtripjob
  [{/My Project/Editions/My Job Option/09 Nov 2007 20_05_24/My Job.indd |
?assetId=123}]
  [&asynch={true | false*}]
  [&return={none | status* | cxml}]
  [&updateSections={true* | false}]
  [&errorIfNoEngines={true | false*}]
 <File MIME Part: indd>
 <File MIME Part: lastcxml>
 <XML MIME Part: jobParams>
```

The InDesign document to be round tripped can either be an existing file in the repository or it can be external to the system entirely. If already existing in the repository, it can be identified by its path in the repository or by its asset ID. If an external file, the contents of the file itself is supplied in the `indd` file MIME part.

If the InDesign document already exists in a job folder in the repository, the roundtrip is done in that folder, which is consistent with operation via the Typefi Publish web UI. If, however, the InDesign document resides elsewhere in the repository or is supplied as a file MIME part, a new job folder will be created in which to perform the roundtrip. This is a minor point but is important in that the content.cxml in an existing folder will be overwritten with the round tripped .cxml.

The `asynch` parameter specifies whether the roundtrip job should run asynchronously or synchronously. If asynchronous, the servlet will queue the job and return immediately. It is the responsibility of the caller to determine when the job is complete by using the ***/listjob*** servlet. If synchronous, the servlet will not return until the job is complete. Since roundtrip jobs are usually fairly quick, running them asynchronously is usually not necessary. It can be valuable, however, if all Engines are busy running other jobs. In that case, the roundtrip job will have to wait in the queue until an available Engine frees up.

When `asynch` is false, the `return` parameter can be used to specify what is returned when the roundtrip job completes. This can be the .cxml file or the status of roundtrip the job as defined below:

- `none`: Return a simple response with the job id.
- `status`: Return the job status as XML (see ***/listjob***).
- `cxml`: Return the resulting .cxml file.

The `updateSections` parameter indicates whether or not the resulting .cxml should be used to update the sections in the repository.

If `errorIfNoEngines` is provided and true, Typefi Publish will abort the job and return an error when there are no engines available to run the job. This applies only to instances where there are no enabled engines capable of running the job. If, on the other hand, there are capable engines but they are busy running other jobs, no error will be returned and the current job will be queued. Likewise if `errorIfNoEngines` is omitted or false, the job will be queued and will remain in the queue until an appropriate engine becomes available.

Lastly, it is important for the caller to release jobs (***/releasejob***) that are not stored permanently (as dictated by the job option's `storeJob` parameter) when they are finished with them. Unreleased jobs will linger in the database until they are explicitly released, causing unnecessary clutter and disk usage. As a convenience, jobs that are not stored are automatically released when output file(s) are returned, as dictated by the `return` parameter.

`<File MIME Part: indd>`

The raw bytes of the .indd file are optionally supplied as a file MIME part.

`<File MIME Part: lastcxml>`

Conditionalized content that does not appear in an output .indd document is not stored in the document. Instead, the InDesign document keeps a reference to that content with a condition identifier that is unique to the section. During a roundtrip, the .cxml exported by Designer is created with those unique references. Typefi Publish then attempts to

update the .cxml with the proper content for all referenced conditions. If the section that a reference resides in already exists in the repository, the missing content is fetched from that section. If, however, the section does not already exist in the repository, Typefi Publish will search for the content in the last known .cxml. If the .indd being roundtripped already exists in a job folder in the repository, the content.cxml in that folder will be used. Otherwise, the .cxml provided in this MIME part will be used.

This MIME part is optional and only needs to be provided if there is a concern with losing unused condition's content during a roundtrip. If so, it is simplest to provide the last known content.cxml that was used to generate the .indd (e.g., the content.cxml used with ***/runjob-cxml***).

`<XML MIME Part: jobParams>`

The `jobParams` MIME part is required and represents a subset of the jobParams used by the ***/runjob*** servlet. Its primary purpose is to define the job option to be used for the roundtrip job. This allows the specification of such things as a particular Engine to use or a set of Designer scripts to run on roundtrip.start and roundtrip.end events. The job option can reference an existing job option, possibly with overrides, or it can define a job option in its entirety.

The jobParams are structured as follows:

```
<jobParams>
  <jobOption {id="123" | name="My Job Option" {projectId="123" | projectName="My
Project"}}>
  ...
  </jobOption>
</jobParams>
```

Like ***/runjob***, <jobOption> can be used to identify an existing job option by id or by its name plus the project identifier. The project identifier, in turn, can be supplied as an id or a name. The other use of <jobOption> is to create a brand new job option, on the fly. In this case, only the project needs to be identified (by its id or name). See the examples below for some of the allowed variations.

Child elements of <jobOption> have different meanings depending on whether or not an existing job option has been identified. If an existing job option has been identified, any child elements are used to override the settings of that job option. If no job option has been specified, the child elements are used to construct a new, on-the-fly job option to be used just for the current job.

Here is how <jobOption>'s child elements are structured:

```
<!-- Engine Selection -->
<!-- Can be used to force the roundtrip to use a particular engine. -->
<engineSelection>
  <overrideProject>{true | false}</overrideProject>
  <engineSpec>
    <engineType id="123" name="My Engine Type"/>
    <engine id="123" name="My Engine"/>
  </engineSpec>
</engineSelection>

<!-- Whether or not to use absolute image paths. Typically this is omitted
(false). -->
<useAbsoluteImagePaths>{true | false*}</useAbsoluteImagePaths>

<!-- Output path -->
<!-- Path is relative to the filestore. content.cxml will be copied here in
addition to its existence in the roundtrip job folder. -->
<outputPath>
  ../My Output
</outputPath>

<!-- Store the job in the repository? -->
<storeJob>
  {*true | false}
</storeJob>

<!-- Scripting options -->
<scriptOptions>
  <!-- Whether or not these options should be used to override the System
defaults. -->
  <overrideSystem>{true | false}</overrideSystem>
  <!-- Script set to use if overriding. -->
  <scriptSet>
    <params>My Params</params>
    <script eventId="roundtrip.start">MyRoundtripStartScript</script>
    <script eventId="roundtrip.end">MyRoundtripEndScript</script>
  </scriptSet>
</scriptOptions>
```

Response (`asynch=true`)

Anytime `asynch` is true, XML indicating the success of the roundtrip job queueing is returned.

SUCCESS

```
<response success="true">
```

```
  <job id="123"/>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Response (asynch=false)

When `asynch` is false, the `return` parameter dictates what is returned.

Response (return=none)

When a return of none is specified, XML indicating the success of the job run is returned.

SUCCESS

```
<response success="true">
  <job id="123"/>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Response (return=status)

If successful, XML describing the job is returned. Otherwise, XML indicating failure is returned. On success, the XML is structured the same as returned by the ***/listjob*** servlet with its default verbosity. Please see that servlet for sample output.

Response (return=cxml)

If successful, the .cxml file is returned. If unsuccessful, the XML is of the same structure as for the failure case above when return=none.

Examples

These examples show various uses of <jobParams>.

USE AN EXISTING JOB OPTION; IDENTIFIED BY ID

```
<jobParams>
  <jobOption id="123"/>
</jobParams>
```

USE AN EXISTING JOB OPTION; IDENTIFIED BY NAME

Since job option is specified by name, a project must also be specified.

```
<jobParams>
  <jobOption name="My Job Option" projectId="123"/>
</jobParams>
```

OVERRIDE AN EXISTING JOB OPTION

```
<jobParams>
  <jobOption id="123">
    <scriptOptions>
      <overrideSystem>true</overrideSystem>
      <scriptSet>
        <params>My Params</params>
        <script eventId="roundtrip.start">MyRoundtripStartScript</script>
        <script eventId="roundtrip.end">MyRoundtripEndScript</script>
      </scriptSet>
    </scriptOptions>
    <storeJob>false</storeJob>
  </jobOption>
</jobParams>
```

CREATE AN ON-THE-FLY JOB OPTION

```
<jobParams>
  <jobOption projectName="My Project">
    <useAbsoluteImagePaths>true</useAbsoluteImagePaths>
    <outputPath>
      ../My Output
    </outputPath>
    <storeJob>false</storeJob>
  </jobOption>
</jobParams>
```

/runjob

Description

/runjob is used to run jobs based on a job option. The job option may already exist or it can be constructed on the fly. The content must already exist in the repository and is referenced by the job option. If running a job for an existing job option, the job option's settings can also be overridden.

Syntax

```
/runjob
  [?asynch={true | false*}]
  [&return={none | pdf | indd | cxml | status | any*}]
  [&inline={true | false*}]
  [&errorIfNoEngines={true | false*}]
 <XML MIME Part: jobParams>
```

The `asynch` parameter specifies whether the job should run asynchronously or synchronously. If asynchronous, the servlet will queue the job and return immediately. It is the responsibility of the caller to determine when the job is complete by using the */listjob* servlet. If synchronous, the servlet will not return until the job is complete.

When `asynch` is false, the `return` parameter can be used to specify what is returned when the job completes. This can be the contents of an output file or the status of the job as defined below:

- `none`: Return a simple response with the job id.
- `pdf`: Return the resulting .pdf.
- `indd`: Return the resulting .indd if one and only one was created.
- `cxml`: Return the resulting .cxml.
- `any`: If a .pdf was created, return it. Otherwise, return the .indd if one and only one was created.
- `status`: Return the same XML structure that is returned by the */listjob* servlet. This includes the status of the job, any messages from the Engine, and a list of all output file paths. Output files can subsequently be fetched with a call to the */get* servlet.

The `inline` parameter is only useful when output files are returned (`asynch = false & return = pdf`). It is used to set the value of the Content-Disposition MIME header (see *RFC 1806*) for the returned files. The header is set to `inline` if the `inline` parameter is true and `attachment` if it is false. As an example of its use, consider the case where the request originates from a web browser. If a .pdf is returned and `inline` is true, the browser will attempt to display the .pdf in the current browser window. If `inline` is false, the browser will attempt to save the .pdf to the file system.

If `errorIfNoEngines` is provided and true, Typefi Publish will abort the job and return an error when there are no engines available to run the job. This applies only to instances where there are no enabled engines capable of running the job. If, on the other hand, there are capable engines but they are busy running other jobs, no error will be returned and the current job will be queued. Likewise if `errorIfNoEngines` is omitted or false, the job will be queued and will remain in the queue until an appropriate engine becomes available.

Lastly, it is important for the caller to release jobs (*/releasejob*) that are not stored permanently (as dictated by the job option's `storeJob` parameter) when they are finished with them. Unreleased jobs will linger in the database until they are explicitly released, causing unnecessary clutter and disk usage. As a convenience, jobs that are not stored are automatically released when output file(s) are returned, as dictated by the `return` parameter.

`<XML MIME Part: jobParams>`

The jobParams MIME part is used to specify the job option to run the job for. Its high level elements look like:

```
<jobParams>
  [<project/>]
  [<jobOption/>]
</jobParams>
```

<project>

<project> is used to override project field values for use with the job. It is structured as:

```
<project {id="123" | name="My Project"}>
  <fields>
    <field {id="123" | name="DocumentTitle"}>
      Alice's Adventures in WonderLand
    </field>
    ...
  </fields>
```

```
</project>
```

<jobOption>

<jobOption> is used to specify an existing job option, possibly with overrides, or to define a job option in its entirety. Here's its syntax:

```
<jobOption {id="123" | name="My Job Option" {projectId="123" | projectName="My
Project"}}>
  ...
</jobOption>
```

In practical terms, <jobOption> can be used to identify an existing job option by id or by its name plus the project identifier. The project identifier, in turn, can be supplied as an id or a name. The other use of <jobOption> is to create a brand new job option, on the fly. In this case, only the project needs to be identified (by its id or name). See the examples below for some of the allowed variations.

Child elements of <jobOption> have different meanings depending on whether or not an existing job option has been identified. If an existing job option has been identified, any child elements are used to override the settings of that job option. If no job option has been specified, the child elements are used to construct a new, on-the-fly job option to be used just for the current job.

Here is how <jobOption>'s child elements are structured:

```
<!-- Job Option Name -->
<!-- Required when building job options on-the-fly. Has no effect otherwise. -->
<name>My Job Option</name>
<!-- Engine Selection -->
<engineSelection>
  <overrideProject>{true | false}</overrideProject>
  <engineSpec>
    <engineType id="123" name="My Engine Type"/>
    <engine id="123" name="My Engine"/>
  </engineSpec>
</engineSelection>

<!-- Engine Configuration -->
<engineCfg>
  <maxTimePerPage>30</maxTimePerPage>
  <maxLayoutsPerPage>100</maxLayoutsPerPage>
</engineCfg>

<!-- Options -->
```

```
<!-- If preset is not provided, will use the default preset set for the engine
type. If presetName is provided, an engineType must also be provided as part of
the engineSelection. -->
<createPdf [{presetId="123" | presetName="[High Quality Print]"}]>
  {*true | false}
</createPdf>
<createIndd>
  {*true | false}
</createIndd>
<createSingleDoc>
  {*true | false}
</createSingleDoc>
<firstPageSide>{left | right*}</firstPageSide>
<firstPageNum>1</firstPageNum>
<xrefs>
  <!-- Whether or not to link xrefs in output. -->
  <createHyperlinks>{*true | false}</createHyperlinks>

  <!-- How to handle unresolvable xrefs. -->
  <handleUnresolvable>
    <!-- If present, replace the xref text with that provided. Otherwise, keep
the last known text. -->
    <replaceText>My Replacement Text</replaceText>
  </handleUnresolvable>
</xrefs>

<!-- Whether or not to use absolute image paths. Typically this is omitted
(false). -->
<useAbsoluteImagePaths>{true | *false}</useAbsoluteImagePaths>

<!-- A field reference only needs to be provided when child is "true" -->
<labelJob [fieldId="123" | fieldName="DocumentTitle"]>
  {*true | false}
</labelJob>
<!-- Conditions -->
<conditions>
  <condition name="US English">
    <value>{*true | false}</value>
  </condition>
  ...
</conditions>

<!-- Template -->
<!-- Required when building job options on-the-fly. -->
<template {id="123" | name="My Template"}/>

<!-- Content -->
```

```
<content>
  <section>
    ...
  </section>
  ...
</content>

<!-- Output Path -->
<!-- Path is relative to the filestore. Output will be written here instead of to
the job folder. Output files are not accessible via the WSI since they are outside
the control of Typefi Publish. -->
<outputPath>
  ../My Output
</outputPath>

<!-- Store the job in the repository? -->
<storeJob>
  {*true | false}
</storeJob>

<!-- Scripting Options -->
<scriptOptions>
  <!-- Whether or not these options should be used to override the System
defaults. -->
  <overrideSystem>{true | false}</overrideSystem>
  <!-- Script set to use if overriding. -->
  <scriptSet>
    <params>My Params</params>
    <script eventId="job.start">MyJobStartScript</script>
    <script eventId="job.end">MyJobEndScript</script>
    <script eventId="book.start">MyBookStartScript</script>
    <script eventId="book.end">MyBookEndScript</script>
    <script eventId="document.start">MyDocumentStartScript</script>
    <script eventId="document.end">MyDocumentEndScript</script>
    <script eventId="section.start">MySectionStartScript</script>
    <script eventId="section.end">MySectionEndScript</script>
    <script eventId="spread.start">MySpreadStartScript</script>
    <script eventId="spread.end">MySpreadEndScript</script>
    <script eventId="page.start">MyPageStartScript</script>
    <script eventId="page.end">MyPageEndScript</script>
        <script eventId="roundtrip.start">MyRoundtripStartScript</script>
        <script eventId="roundtrip.end">MyRoundtripEndScript</script>
    <!-- There is no spill.start event. -->
    <script eventId="spill.end">MySpillEndScript</script>
  </scriptSet>
</scriptOptions>
```

When <outputPath> in the jobParams is used to override the output location, `pdf` and `indd` cannot be used as return types. This is because those files are not under Typefi Publish's control, and therefore, cannot be returned with guaranteed accuracy. Additionally, these files and the job .log will not be listed in the job's status.

Response (asynch=true)

Anytime asynch is true, XML indicating the success of the job queuing is returned.

SUCCESS

```
<response success="true">
  <job id="123"/>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Response (asynch=false)

When asynch is false, the return parameter dictates what is returned.

Response (return=none)

When a return of none is specified, XML indicating the success of the job run is returned.

SUCCESS

```
<response success="true">
  <job id="123"/>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Response (return=pdf)

If successful, the .pdf file itself is returned. If unsuccessful or if no .pdf was produced, XML indicating failure is returned. The XML is of the same structure as for the failure case above when return=none.

Response (`return=indd`)

If successful, all .indd files are returned. As for pdf, XML indicating failure is returned when the job is unsuccessful or when no InDesign files were produced.

Response (`return=cxml`)

If successful, the .cxml file is returned. As for pdf, XML indicating failure is returned when the job is unsuccessful or when no .cxml file is produced.

Response (`return=status`)

If successful, XML describing the job is returned. Otherwise, XML indicating failure is returned. On success, the XML is structured the same as returned by the ***/listjob*** servlet with its default verbosity. Please see that servlet for sample output.

Response (`return=any`)

`any` indicates that the servlet should return whatever it can. If a .pdf was produced, it will return that. If a .pdf was not produced but .indd documents were, it will return those. If no .indd documents were produced but a .cxml was, it will return that. If none of these cases holds, the servlet will return XML showing the status of the job as described above for return=status. On failure, XML indicating failure is returned as for the other return options.

Examples

These examples show various uses of <jobParams>.

USE AN EXISTING JOB OPTION; IDENTIFIED BY ID

```
<jobParams>
  <jobOption id="123"/>
</jobParams>
```

USE AN EXISTING JOB OPTION; IDENTIFIED BY NAME

Since job option is specified by name, a project must also be specified.

```
<jobParams>
  <jobOption name="My Job Option" projectId="123"/>
</jobParams>
```

USE AN EXISTING JOB OPTION; OVERRIDING PROJECT FIELD VALUES

If the specified project does not match the project of the specified job option, the fields will be ignored.

```
<jobParams>
  <project name="My Project">
    <fields>
      <field name="DocumentTitle">Alice's Adventures in WonderLand</field>
      <field id="123">Lewis Carroll</field>
    </fields>
  </project>
  <jobOption name="My Job Option" projectName="My Project"/>
</jobParams>
```

USE AN EXISTING JOB OPTION; OVERRIDING SOME SETTING

```
<jobParams>
  <jobOption id="123">
    <createPdf presetName="[High Quality]">true</createPdf>
    <storeJob>true</storeJob>
  </jobOption>
</jobParams>
```

CREATE AN ON-THE-FLY JOB OPTION

```
<jobParams>
  <jobOption projectName="My Project">
    <name>My Job Option</name>
    <createPdf presetName="[High Quality]">
      true
    </createPdf>
    <template name="My Template"/>
    <content>
      <section id="123"/>
      <section id="456"/>
      <section id="789"/>
    </content>
  </jobOption>
</jobParams>
```

/runjob-cxml

Description

/runjob-cxml is the same as */runjob* except that it also takes a `cxml` MIME part containing the content for the job. This content is used instead of the sections configured for the job option. Like */runjob*, the job can be run for an existing job option or for a job option created in the `jobParams` part.

Syntax

```
/runjob-cxml
  [?asynch={true | false*}]
  [&return={none | pdf | indd | status | any*}]
  [&inline={true | false*}]
  [&errorIfNoEngines={true | false*}]
 <XML MIME Part: jobParams>
 <XML MIME Part: cxml>
```

<XML MIME Part: jobParams>

The jobParams MIME part is identical to that for */runjob*.

<XML MIME Part: cxml>

The cxml MIME part contains the content XML to be used to run the job.

Response

The response is identical to that for */runjob*.

/runjob-xml

Description

/runjob-xml is the same as */runjob-cxml* except that it takes an `xml` MIME part and supports an `xsl` element within the jobOption XML. The `xml` part holds XML to be used for the content of the job. XSL templates are stored in the Transforms folder of the Typefi Publish XML folder (ex: /Typefi/Publish/XML/Transforms)

Syntax

```
/runjob-xml
  [&asynch={true | false*}]
  [&return={none | pdf | indd | status | any*}]
  [&inline={true | false*}]
  [&errorIfNoEngines={true | false*}]
 <XML MIME Part: jobParams>
 <XML MIME Part: xml>
```

<XML MIME Part: jobParams>

The `jobParams` MIME part is identical to that for */runjob* except for overriding the XSL specification. For example,

```
<jobParams>
  <jobOption name="My Job Option">
    <xsl>
      <transform name="OurXMLtoCXML.xsl"/>
      <catalog name="OurCatalog.xml"/>
    </xsl>
  </jobOption>
</jobParams>
```

Of course, other `jobOption` details can be specified as well. See */runjob*for more details.

Note that the label for the job will always use the XML file name.

`<XML MIME Part: xml>`

This xml MIME part contains the XML that will be transformed and then used as content for the job.

Response

The response is identical to that for ***/runjob***.

/runjob-raw

Description

/runjob-raw is used to run jobs using files that are not part of Typefi Publish's repository. It shares a similar API as */runjob-cxml* with a few additions and restrictions.

Syntax

```
/runjob-raw
  [?asynch={true | false*}]
  [&errorIfNoEngines={true | false*}]
 <File MIME Part: cxml>
 <XML MIME Part: jobParams>
```

The `asynch` parameter specifies whether the job should run asynchronously or synchronously. If asynchronous, the servlet will queue the job and return immediately. It is the responsibility of the caller to determine when the job is complete by using the */listjob* servlet. If synchronous, the servlet will not return until the job is complete.

`<File MIME Part: cxml>`

Optional: If a cxml file is not specified with the 'cxml' element (see *below*) then a CXML MIME part must be included with the content to use for the job.

Files produced by the */runjob-raw* servlet cannot be downloaded via the Typefi Publish web UI. This is because those files are not under Typefi Publish's control, and therefore, cannot be returned with guaranteed accuracy. Additionally, these files and the .log will not be listed in the job's status.

Unlike the */runjob* and */runjob-cxml* calls, */runjob-raw* does not support the `return` and `inline` parameters. It always returns the status of the job using the same XML structure that is returned by the */listjob* servlet. This includes the status of the job, any messages from the Engine, and a list of all output file paths.

If `errorIfNoEngines` is provided and true, Typefi Publish will abort the job and return an error when there are no engines available to run the job. This applies only to instances where there are no enabled engines capable of running the job. If, on the other hand, there are capable engines but they are busy running other jobs, no error will be returned and the current job will be queued. Likewise if `errorIfNoEngines` is omitted or false, the job will be queued and will remain in the queue until an appropriate engine becomes available.

Syntax

As a convenience, ***/runjob-raw*** will automatically release jobs that have `storeJob` set to false. This avoids cluttering the database with job entries for completed jobs.

```
<XML MIME Part: jobParams>
```

The `jobParams` MIME part is used to specify the job option to run the job for. Its high level elements look like:

```
<jobParams>
  [<project/>]
  [<jobOption/>]
</jobParams>
```

<PROJECT>

<project> is optional and is only used to define project level field values. Since the job is not being run within the context of a Server project, no project name or id is required. It is structured as:

```
<project>
  <fields>
    <field {name="DocumentTitle"}>
      Alice's Adventures in WonderLand
    </field>
    ...
  </fields>
</project>
```

<JOBOPTION>

<jobOption> is used to define a job option in its entirety. Unlike, ***/runjob*** & ***/runjob-cxml***, an existing `jobOption` cannot be used. The only attribute to `jobOption` supported by ***/runjob-raw*** is `basePath` which indentifies the folder where the job files are located.

Note that the folder needs to be directly accessible to the Engine & Designer, but not to the Typefi Publish server itself. Here is its syntax:

```
<jobOption basePath="/JobFiles/Job">
  ...
</jobOption>
```

Here is how <jobOption> child elements are structured:

```
<!-- Engine Selection - optional:  if not specified the first available 'Generic'
engine will be used -->
<engineSelection>
  <engineSpec>
    <engineType id="123" name="My Engine Type"/>
    <engine id="123" name="My Engine"/>
  </engineSpec>
</engineSelection>

<!-- Engine Configuration - optional -->
<engineCfg>
  <maxTimePerPage>30</maxTimePerPage>
  <maxLayoutsPerPage>100</maxLayoutsPerPage>
</engineCfg>

<!-- Options -->
<!-- If preset is not provided, will use the default preset set for the engine
type. -->
<createPdf [{presetId="123" | presetName="[High Quality Print]"}]>
  {*true | false}
</createPdf>
<createIndd>
  {*true | false}
</createIndd>
<!-- true to create a single InDesign document, false to create a book -->
<createSingleDoc>
  {*true | false}
</createSingleDoc>
<firstPageSide>{left | right*}</firstPageSide>
<firstPageNum>1</firstPageNum>

<!-- Whether or not to use absolute image paths.
Typically this is omitted (false). -->
<useAbsoluteImagePaths>{true | *false}</useAbsoluteImagePaths>

<!—How to label the job. A literal value can be specified or the value
of a project field (specified in the project element above) can be used. -->
<labelJob [value="Job Label" | fieldName="DocumentTitle"]>
```

```
  {*true | false}
</labelJob>

<!-- Conditions -->
<conditions>
  <condition name="US English">
    <value>{*true | false}</value>
  </condition>
  ...
</conditions>

<!-- Template -->
<!-- Required. Name is a path relative to basePath of the InDesign template to
use. You can specify the indd or txml file. -->
<template name="My Template.indd"/>

<!-- Content -->
<!-- Optional. Name is a path relative to basePath of the CXML file to use. If not
specified, the CXML can be attached using the CXML MIME part in the same manner
as /runjob-cxml. This file will always be used even when a cxml MIME part is
specified.-->
<cxml name="My Content File.cxml"/>

<!-- Output Path -->
<!—Optional. Path is relative to basePath. If not specified, files will be written
to 'basePath'. -->
<outputPath>
  ../My Output
</outputPath>

<!-- Store the job entry in the database? -->
<storeJob>
  {*true | false}
</storeJob>
<!-- Scripting Options -->
<scriptOptions>
  <!-- Whether or not these options should be used to override the System
defaults. -->
  <overrideSystem>{true | false}</overrideSystem>

  <!-- Script set to use if overriding. -->
  <scriptSet>
    <params>My Params</params>
    <script eventId="job.start">MyJobStartScript</script>
    <script eventId="job.end">MyJobEndScript</script>
    <script eventId="book.start">MyBookStartScript</script>
    <script eventId="book.end">MyBookEndScript</script>
```

```
    <script eventId="document.start">MyDocumentStartScript</script>
    <script eventId="document.end">MyDocumentEndScript</script>
    <script eventId="section.start">MySectionStartScript</script>
    <script eventId="section.end">MySectionEndScript</script>
    <script eventId="spread.start">MySpreadStartScript</script>
    <script eventId="spread.end">MySpreadEndScript</script>
    <script eventId="page.start">MyPageStartScript</script>
    <script eventId="page.end">MyPageEndScript</script>
        <script eventId="roundtrip.start">MyRoundtripStartScript</script>
        <script eventId="roundtrip.end">MyRoundtripEndScript</script>
    <!-- There is no spill.start event. -->
    <script eventId="spill.end">MySpillEndScript</script>
  </scriptSet>
</scriptOptions>
```

Response (`asynch=true`)

Anytime `asynch` is true, XML indicating the success of the job queueing is returned.

SUCCESS

```
<response success="true">
  <job id="123"/>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

Response (`asynch=false`)

When `asynch` is false, and the job is successful, status XML describing the job is returned. Otherwise, XML indicating failure is returned. On success, the XML is structured the same as returned by the ***/listjob*** servlet with its default verbosity. Please see that servlet for sample output.

Example

This example shows various uses of <jobParams>.

Runs a job using files stored in the folder /Typefi/Tests/runjob-raw. Output is stored in a subfolder named 'Output'. Content is provided in the file named 'ThingsToDo.cxml' located in /Typefi/Tests/runjob-raw.

```
<jobParams>
<!-- Unlike /runjob & /runjob-cxml, the project element does not refer to an
existing project. Instead it is used to list project field values. -->
  <project>
    <fields>
      <field name="Book Title">Super Duper</field>
      <field name="Job Label">runjob-raw test</field>
      <field name="Copyright Year">2010</field>
    </fields>
  </project>

  <!-- jobOption is slightly modified from jobParams used by /runjob & /runjob-
cxml. id & project information is not used. Instead a basePath is specified. This
is the folder where the job files reside. The 'label' attribute is used to specify
-->
  <jobOption basePath="/Typefi/Tests/runjob-raw">
    <template name="Template_November.indd" />
    <cxml name="ThingsToDo.cxml"/>

    <!-- Generated documents will be placed in an Output folder within the
basePath -->
    <outputPath>Output</outputPath>
    <engineSelection>
       <engineSpec>
          <engineType name="Generic"/>
       </engineSpec>
    </engineSelection>
    <createPdf presetName="[Low Quality]">true</createPdf>
    <createSingleDoc>true</createSingleDoc>
    <conditions>
      <condition name="Int English">false</condition>
      <condition name="US English">true</condition>
    </conditions>
    <!-- Use the value of 'fieldName' for the job label -->
    <labelJob fieldName="Job Label">true</labelJob>
    <storeJob>true</storeJob>
  </jobOption>
</jobParams>
```

/sendmessage

Description

Sends a message to the authenticated user via the Console.

Syntax

```
/sendmessage
  [?{userId=123 | user=admin}]
  [&body=<message text>]
  [&subj=<message subject>]
  [&type={info*|warn|error}]
  [&dest={console*|other}]
  [&refresh={true|false*}]
  [&refreshBefore={true|false*}]
```

The user identifier specifies what user to send the message to. If it is omitted, the message will be sent to the authenticated user.

`body` contains the message, while `subj` contains a subject. It is up to the application displaying the message as to whether or not the subject is used (the Console does not display subject).

The `type` parameter indicates whether the message is an informational message, a warning, or an error. The application displaying the message may choose to show the message differently depending on the type. The Console displays `info` messages in a sliding tab that does not require user input and may, in fact, never be seen by the user (if they're away from their computer, e.g.). `warn` and `error` messages, on the other hand, are displayed in dialogs that must be actively dismissed by the user.

The `dest` parameter indicates where this message should be delivered to. A value of `console` will deliver the message to the user at the Console. A value of `other` is meant for third party applications. Those applications can use ***/getmessages*** to fetch these messages.

The `refresh` parameter specifies whether or not the Console should attempt a browser refresh before or after the user sees the message. This can be useful if a page refresh would reveal something important to the user. For example, when a new file is uploaded to Typefi Publish, a page refresh forces that file into the user's view.

`refreshBefore` only has meaning when `refresh` is true. If `refreshBefore` is supplied and is true, the Console will refresh before displaying the message to the user. Otherwise, the Console refreshes after displaying the message to the user.

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/synchfs

Description

When files are placed into a filestore folder via the filesystem, they are only added to the database once a user browses the folder in the Console. The */synchfs* servlet forces this synchronization to occur immediately. This allows, for example, an external application to place a file in the filestore and then call */synchfs* to register it with the database. Likewise, */synchfs* can be used to update the database when a file is deleted from the filestore.'

The folder names have changed since the servlet was originally created. The labels here are for backwards compatibility. Use `sandbox` to synch the Content folder and `repository` to synch the Sections folder.

Syntax

```
/synchfs
  {/My Project/Content |
  {?{projectId=123 | project=My Project}
  [&folder={templates|sandbox|repository|images|all*|jobs}]}}
```

The caller can provide a path or a project identifier (project id or name) and a folder. The `all` value for folder is used to synchronize all of the folders with the exception of job folders. The `jobs` value for folder is used to synchronize the job folders and should be use cautiously since there can be potentially thousands of jobs to synchronize.

Examples

SYNCH BASED ON PATH:

```
/synchfs/My Project/Content
```

SYNCH BASED ON PROJECT AND FOLDER:

```
/synchfs?project=My Project&folder=sandbox
```

SYNCH A JOB FOLDER:

```
/synchfs/My Project/Editions/My Job Option/18 Dec 2006 21_31_33
```

Response

SUCCESS

```
<response success="true"/>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```

/updateasset

Description

Updates the description and/or contents of an asset.

Syntax

```
/updateasset
 <XML MIME Part: assetDescr>
 <XML MIME Part: assetContent>
```

<XML MIME Part: assetDescr>

The required `assetDescr` part is XML used to modify information about the asset like owner, creation date, whether or not it's read only, etc. It is required because it also identifies the asset.

```
<asset {id="123" | path="/My Project/Content/alice.doc"}>
  [<position>2</position>]  <!-- applies to sections only -->
  [<owner id="123" username="admin"/>]
  [<creationDate>2006-05-17 01:07:42.155 GMT</creationDate>]
  [<modifiedDate>2006-05-17 02:17:14.301 GMT</modifiedDate>]
  [<checkedOut val="{true | false}">
    <user {id="123" | username="admin"}/>  <!-- only present if val = true -->
  </checkedOut>]
  [<readOnly val="{true | false}">
    <user {id="123" | username="admin"}/>  <!-- only present if val = true -->
  </readOnly>]
</asset>
```

<XML MIME Part: assetContent>

The optional `assetContent` part holds actual asset binary content. If supplied, it will be used to update the actual content of the asset in the filestore.

Response

SUCCESS

```
<response success="true">
  <asset id="123"/>
</response>
```

FAILURE

```
<response success="false">
  <error code="<error code>">
    <language type="en-US">Error Text</language>
  </error>
</response>
```