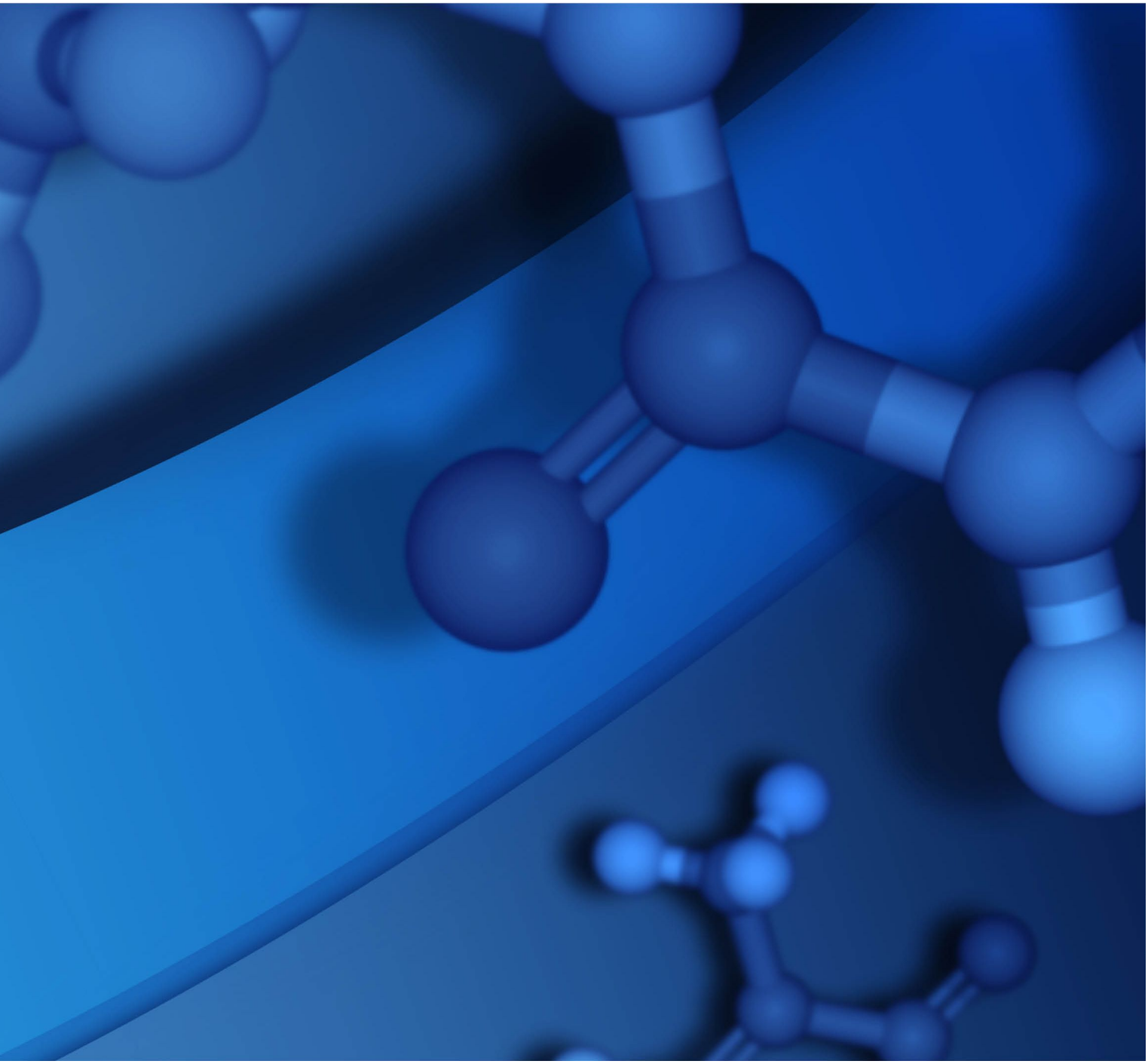
DASSAULT SYSTEMES

BIOVIA

RESTFUL WEB SERVICES GUIDE

INTEGRATION COLLECTION

PIPELINE PILOT 9.5



Copyright Notice

Acknowledgments and References

To print photographs or files of computational results (figures and/or data) obtained using BIOVIA software, acknowledge the source in an appropriate format. For example:

> "Computational results obtained using software programs from Dassault Systèmes BIOVIA. The *ab initio* calculations were performed with the DMol^3 program, and graphical displays generated with BIOVIA Pipeline Pilot Server."

BIOVIA may grant permission to republish or reprint its copyrighted materials. Requests should be submitted to BIOVIA Support, either through electronic mail to support@accelrys.com, or in writing to:

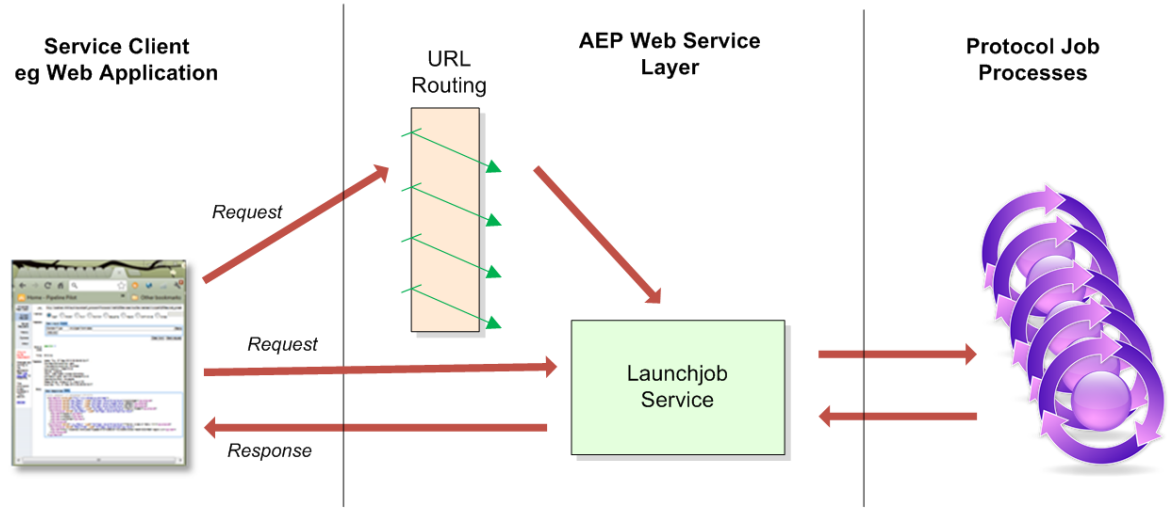
BIOVIA Support
5005 Wateridge Vista Drive, San Diego, CA 92121 USA

Contents

Chapter 1:
RESTful Web Services Guide

Overview

This information is intended to support application developers who need to expose protocols as RESTful web services.



A protocol can be invoked directly by name using the Launchjob service, or else the web service developer can define an arbitrary URL that maps to the protocol via the URL Routing layer. In either case, result data may be streamed back directly from the protocol job, or else the Launchjob service provided a number of options for defining and formatting the response data.

This document first describes the URL Routing system, and then follows with a section on the details of the Launchjob service, which will be useful for those intending to invoke the service directly or as an aid to writing more advanced URL routing configurations.

Additional Information:

- For more information about the Pipeline Pilot and other BIOVIA products, visit https://community.accelrys.com/index.jspa.

Chapter 2:
URL Routing

URL Routing is defined for a package in its `package.conf` file, and generally maps a specified URL to the execution of a protocol. For details about modifying the `package.conf` file, see the *Application Packaging Guide*.

To implement one or more RESTful web service URLs with protocols, you need to indicate which protocol should be invoked to handle the specific HTTP request – this is specified as a combination of a URL path and an HTTP method verb. To specify the mapping between a RESTful web service URL and a protocol in the `package.conf` file, use the <url> directive group and the REST modifier, as below.

```
<url REST AUTH>
…
…
</url>
```

The AUTH (default) modifier indicates that all requests will pass through the standard server authentication module, supporting various forms of single sign on.

> **Note:** The AUTH modifier may be replaced with ANON to indicate that the defined service supports anonymous access. For more information, see the section below on Anonymous service access.

```
<url REST ANON>
…
…
</url>
```

As with all `package.conf` directives, text values are case-insensitive.

Within the <url> directive group, you will define the HTTP method, the URL path of the web service request, the protocol to which it maps and other settings for the mapping. These are documented below in the Routing specification section of this document. Here is a simple example to give you a preview of what a full URL mapping might look like.

```
<url REST AUTH>
       method GET
       request /discussion/topics/{topicId}
       protocol Protocols/Web Services/AcclDev/board/messages/Show
       <meta>
               summary  Returns the content of a topic specified by id.
               publish True
       </meta>
</url>
```

Handling protocol parameters

When a protocol is exposed as a web service, the protocol's parameters are mapped as the exposed service parameters. The routing layer provides 3 ways to handle the provision of these service parameters:

1. A parameter whose value is specified by the web service client can be defined as an element in the service request. This is detailed in the **request** section below. In the example above, note the line:

```
request /discussion/topics/{topicId}
```

In this example, the value of the URL element matching the location of the `{topicId}` token in the request path is passed to the protocol as a parameter named "`topicId`". So let's assume the request is

```
https://host:port/discussion/topics/99
```

Then protocol parameter `topicId` is set to the value 99.

2. Another way to allow the user to pass a parameter to the protocol is as a query string parameter.

```
https://host:port/discussion/list?contrib=jwong
```

In this example, the `contrib` parameter value is passed to the protocol, unless query string passing is suppressed – see the **querystring** section below for details.

3. The routing layer can supply its own value for parameters, using the **param** property in the routing definition – again, see the corresponding section below. This may be a constant value or a value taken from the HTTP request itself. Here is an example of each:

```
param           NameFilter      L-P
param          UserAgent       %{HTTP_USER_AGENT}
```

Routing specification

Inside the <url> directive, the routing details are specified with property name-value settings following the standard package syntax. i.e. a setting is defined on a line of the `package.conf` file, where the value of the property follows the white space after the property name, up until the end of the line or a comment marker (#). These settings define the behavior of the routing. Valid property names include the following:

- method
- request
- protocol
- param
- constraint
- query
- string
- action

In addition, the service details can include metadata settings to supply information about service, such as a service description. This may be used by other subsystems such as a service registry or a client. Metadata is enclosed in a <meta> directive, described in a later section.

The following section provides details on the various routing properties that you can include in the service specification, within the <url> directive.

method

This property defines the http method under which the routing will take place. Valid values include:

HTTP Method	Value
GET	Retrieves a data resource or a resource collection
POST	Creates a new resource

HTTP Method	Value
PUT	Replaces a resource
PATCH	Updates resource content
DELETE	For removing a resource

Note: All methods except POST should always be idempotent. (This means that the effect is the same whether invoked 1 time or 100 times.)

request

This property defines the form of the URL path used in the mapping. How you specify the request path determines which HTTP requests are mapped. You may define the request as a simple fixed path or you can create more complex paths with variable elements which are matched against a request URL; when a matching request specification is found, the content of the variable elements are passed on to the protocol as protocol parameters.

The request specification format for the inclusion of variable tokens is based on some useful portions of the IETF RFC 5670 proposal on URL templates (http://tools.ietf.org/html/rfc6570). The details are listed below.

Note: There is an alternative approach for specifying a request that may be more natural for users of Pipeline Pilot and PilotScript and offers a couple of unique capabilities. See Appendix C: Alternative specification for a request property.

Request Specification Format Rules

1. **Basic format**

   The simplest specification is a URL path that matches a request with an identical path. The URL is a reference to a collection of items and has a trailing slash by convention.

   Example:

   ```
   request          /zoology/animals/
   ```

   The full request from the client would be something like this:

   ```
   http://localhost:9944/zoology/animals/
   https://localhost:9943/zoology/animals/
   ```

2. **Path element tokens**

   To add some flexibility, you can specify a named variable token in the path specification. During URL path matching, this tokenized section of the path is treated as variable. It can be different in each request. These path element tokens are enclosed in braces. The token can be specified as a whole or partial path element. When a match is found, a property is created with the name of the token and the content in the path that maps to the token is assigned as the property value. The property is passed on to the protocol that handles the request (this should be defined as a protocol parameter).

   Examples:

   ```
   request          /zoology/animals/{id}
   ```

   In the above example, a request path of

   ```
   /zoology/animals/gor32
   ```

results in the protocol parameter "id" being set to the value "gor32".

```
request          /zoology/animals/{name}/update
```

Here, a request path of

/zoology/animals/nargle/update

results in the protocol parameter "name" being set to the value "nargle".

Notes:

- A forward slash ('/') or end of the URL path is treated as the termination of a token. More specifically, the token will only match against unreserved URL characters, which excludes characters like '/', '&', '?' and '=' which have special meaning in a URL.
- To indicate a more complex regular expression for parameter matching, see the constraint property below.

Additional Examples:

```
request                 /zoology/animals/Bird{id}

request                /zoology/animals/{zoo_ref}

request                /zoology/animals/gen{genus}/{id}

request                /zoology/animals/{genus}G/{id}
```

3. **Comma-separated tokens**

   A variation on the item above is support for multiple tokens separated by commas, all within a single path element. This is not the only way to specify this tokenization, but is consistent with RFC 6570.

   Example:

   ```
   request          /zoology/animals/filters/{Name,NumLegs}
   ```

   In the above example, a request path of

   /zoology/animals/filters/M,4

   results in the protocol parameter "Name" being set to the value "M" and the parameter "NumLegs" being set to the value 4.

4. **Multiple path element tokens**

   This syntax also employs multiple tokens separated by commas, but prefixes a forward slash ('/') after the opening brace. This can be useful for describing a path element hierarchy, mapping each element to a separate protocol parameter. This is not the only way to specify this tokenization, but is consistent with RFC 6570.

   Example:

   ```
   request          /zoology/animals/filter{/Name,NumLegs}
   ```

   In the above example, a request path of

   /zoology/animals/filters/Pe/2

   results in the protocol parameter "Name" being set to the value "Pe" and the parameter "NumLegs" being set to the value 2.

5. **Reserved character tokens (& matching across multiple path elements)**

   To specify a token that matches any sequence of characters, including reserved URL characters

(such as '/' and '&'), use a token marker with a '+' character immediately following the opening brace. In particular, this is useful for matching across multiple path elements of a URL since the token can cross element boundaries denoted by the forward slash.

Example:

```
request          /zoology/animals/{id}/{+special}
```

In the above example, a request path of

`/zoology/animals/104/legs/eggs/gestation`

will result in the protocol parameter "`id`" being set to the value "104", and the protocol parameter "`special`" being set to the value "`legs/eggs/gestation`".

More Examples:

```
request                  /zoology/animals/{+gendetails}/@id
request                 /chem/img2/{+smiles}
```

6. **URL query string and fragments**

   URL query strings and fragments included in the URL specification are ignored for the purposes of matching URL requests. Any query string passed to the server with the request is passed on to the mapped service (unless the querystring setting - see below – is switched to "Off"). To specify a URL query string or fragment in the request specification (e.g. for documentation purposes), use the format defined by RFC 6570, listing the query string property or property list within braces, with a '?' immediately following the opening brace. URL fragments are indicated similarly, using a hash character ('#').

   Examples:

```
request                  /zoo2/animals/{?As}
request                 /zoo2/animals/{?As,Name,NumLegs}
request                 /zoo3/animals/{?Name}{#ClientTag}
request                 /zoo3/animals{/Name}{#ClientTag}
```

   In the above examples, the request matching is based on the section of the request before the first token with a '?' or '#'.

protocol

This is the protocol to which the request is mapped. The standard forms of protocol identification are supported (generally, full protocol path or the protocol GUID).

Examples:

```
protocol                 Protocols/Web Services/AcclDev/zoo/Actions/animal/Show

protocol                {291E3F68-073A-4E88-92B2-1A3F178138D8}
```

param

The param property facilitates the specification of parameter values specific to this request mapping. When a path mapping is made and a protocol is invoked, any param values are set as parameter values on the protocol. When a protocol is exposed as a web service, the protocol's parameters correspond to the exposed service parameters. A param setting provides a technique for a service definition to set a value for a specific named parameter based on a constant value or on an HTTP request setting.

There are three categories of param settings:

1. **Set a protocol parameter from a constant value**

   In this mode, the named parameter is defined with a constant value for this URL routing instance.

Example:

```
param           NameFilter P-T
```

Notes:

- By default the parameter name is space-delimited. For parameter names with spaces, you can surround the token name with braces, or the @'…' format (with single quotes) is also supported. Examples:

```
param           {Compress Response} false
param          @'Max Records' 1000
```

- Any text following the parameter name (and before a comment marker #) is taken as the parameter value.

2. **Set a protocol parameter from an HTTP request detail value**

   This mode of use allows you to set the parameter with a value taken from the HTTP request at run time. In this way, the protocol can be informed of any relevant information about the request and its behavior can be adjusted accordingly. For example, you could configure a single protocol to handle both POST and PUT requests, and include a parameter on the protocol to inform which request type to handle.

   HTTP request information is specified by defining the param value using the syntax %{request_variable}, where "request_variable" is one of a number of supported options. The request variable options are listed in Appendix A.

   Examples:

```
param           ReqM    %{REQUEST_METHOD}

param          @'accept encoding' %{HTTP:Accept-Encoding}
```

3. **Define a service behavior setting**

   There are numerous possible parameter values that are picked up by the service launcher. These begin with an underscore or a $, and are detailed in the section of this document that covers the launchjob service.

   Examples:

```
param           $timeout        20000

param          $expireJob      3600
```

constraint

The constraint property enables the specification of a regular expression to be used when matching a path element token. A constraint is declared as a parameter name followed by the regular expression. The parameter name supports the use of brace delimiters or the full @'…' syntax where the parameter name contains spaces. The regular expression can be written with or without regex slashes.

Examples:

```
request         /lab/sample/reset/@MaxNum

constraint              MaxNum          /\d+/

……
……
```

```
request                 /lab/sample/@'sample ID'

constraint              @'sample ID'    \w{3}-\d{2}
```

Notes:

- Be careful with the use of parentheses in the regular expression, especially if you have multiple tokens to match in the URL path. You can confuse the back reference substitution index count. A single outer set of parenthesis is okay, such as: (\w{3}|\d{2}).
- Any text following the parameter name (and before a comment #) is taken as the regex value.

querystring

By default, a query string appended to a request URL is broken up and passed as parameters to the target protocol. This behavior can be switched off by setting the querystring off. Default is on.

Example:

```
request          /lab/data/reset

querystring     Off
……
```

Note that even under default behavior, system parameters are suppressed from a query string that is passed through the routing layer. This means that any parameter that starts with an underscore ('_') or a $ is ignored by the protocol launching system and is *not* passed on to the target protocol. Such parameters are described in the section in this document detailing the launchjob service. If you decide that your service should permit the input of such parameters by a client, you can set the querystring setting to `Unrestricted`.

Example:

```
request          /lab/data/open

querystring     Unrestricted   # System parameters are supported
……
```

Note: As mentioned above, the default behavior suppresses any system parameter from a query string that is included in a request that passes through the routing layer. One exception to the rule above is the `_expireJob` parameter to support backward compatibility for web application clients that have set their job retention explicitly.

action

This is specialized setting that is not required for the common case of URL routing that maps protocol execution to a URL. The action setting value can be one of the following 3 options:

- The value `_protocol_`. This is the default and the **action** setting is therefore generally omitted in this case. In this mode, the request maps to a protocol and you must include a **protocol** setting as described above.

```
request /zoo/animals
action _protocol_   # This is not really needed in this case!
protocol        Protocols/Web Services/AcclDev/zoo/Actions/animal/Show
……
```

- The value _pass-through_. In this case, no URL mapping is created and the request URL is not affected by the presence of the entry. No **protocol** setting is expected. So why use this mode? To provide documentation and URL publication using a common methodology for those server URLs that are not implemented as protocols. See the later metadata section on how to document a defined service.

```
request /charts/cache/@cid
action _pass-through_
……
```

- Another server URL. In this mode, the request is mapped to that other server URL. This can be useful to provide flexibility with URL naming and backward compatibility. No **protocol** setting is expected.

```
request /jobs/list
action /job-info/listing/full
……
```

Service Metadata

Metadata properties of a service provide information about the service, but do not impact the actual routing behavior. Descriptive properties may be defined for a service by enclosing them in a <meta> directive. In principle, any named setting may be included, but there are some standard metadata property names that should be used to provide consistency. As with all other settings, the value of the property follows the white space after the property name, up until the end of the line or a comment marker (#).

summary	This entry should be a line of text that will be used in listings of available services. The value does not affect the behavior of the URL routing itself.
description	The value of this setting should be a server-relative url to an html link for the service description.
version	Version information for the service.
publish	This is a boolean setting, and so should have the value of True or False (default). The value does not affect the behavior of the URL routing, but is an indication of whether this URL is suitable for publication in a server URL directory to be discovered by any user. Some application URLs may be designed as an element in an application workflow and it may not be appropriate to publish the URL outside of that context.

Examples of full <url> specifications

```
<url REST AUTH>
       method GET
       request /zoology/animals/
protocol Protocols/Web Services/AcclDev/zoo/Actions/animal/List
<meta>
       summary  List all the animals in the zoo.
</meta>
</url>

<url REST AUTH>
```

```
       method GET
       request /zoology/animals/@id
       protocol Protocols/Web Services/AcclDev/zoo/Actions/animal/Show
<meta>
               summary  Returns information on an animal specified by id.
               description  /services/registry/lib/get-animal.htm
               publish  True
</meta>
</url>

<url REST AUTH>
       method POST
       request /zoology/animals/
       protocol Protocols/Web Services/AcclDev/zoo/Actions/animal/New
<meta>
               summary  Add a new animal to the zoo
</meta>
</url>
```

Nested <url> specifications

A single level of nesting is supported, to reduce duplication of common settings. A full or partial specification at the top level is accompanied by one or more subdirectives, each of which inherits the top level settings, and utilizes these, overrides them and augments with its own specific settings.

Example:

```
<url REST AUTH>
       method GET
       request /zoology/animals/
       protocol Protocols/Web Services/AcclDev/zoo/Actions/animal/List
       querystring     Off
</meta>
       summary  List all the animals in the zoo.
</meta>
<url>
               request /zoology/m-animals/
               param NameFilter M
       <meta>
                       summary Retrieve animals starting with the letter M
       </meta>
       </url>
       <url>
               request /zoology/c-animals/
               param NameFilter C
       <meta>
                       summary Retrieve animals starting with the letter C
       </meta>
       </url>
</url>
```

The example above specifies three different URL GET mappings, although the all utilize the same protocol. The nested <url> directives each set a specific parameter value to differentiate their behavior.

Job pools

Pooling of jobs is an important characteristic of the platform to achieve optimal turnaround performance, particularly for short-running (< 1 second) protocols. A pool is a named set of one or more persistent job processes. By targeting multiple job runs to a specific named pool, they can each leverage an existing process, avoiding any overhead of process startup, loading of libraries, etc. Pools works best when a pool is used to support related jobs that use the same libraries and components, since such resources will already be loaded into the pooled processes.

By default, all mapped URLs are assigned to a single pool named for the package that owns that URL. To manage this more directly, define a named pool for a specific URL. To do this, add a "__poolid" parameter to the <url> mapping section.

```
<url REST AUTH>
       method GET
       request /zoology/animals/
protocol Protocols/Web Services/AcclDev/zoo/Actions/animal/List
param  __poolid animal-magic
<meta>

       summary  List all the animals in the zoo.
</meta>
</url>
```

There may be cases where you want to disable all pooling (for example, under full impersonation on Windows with a large number of concurrent users), and you can achieve this with a NOPOOL flag on the <url> directive.

```
<url REST AUTH NOPOOL>
       method GET
       …
       …
</url>
```

Protocol authoring considerations

Bear the following mind when authoring a protocol to support a RESTful service:

- Request body content is passed to the first or only protocol parameter by default. The _bodyParam parameter of launchjob (see below) can be used to direct the body content to a specific parameter.
- Response body content is taken from the first or only declared result property of the protocol. The _streamData parameter of launchjob (see below) can be used to extract body content from a specific result property.
- The content type of a result property can be specified by annotating the global property with a metadata property named "content-type".

  Example:

  ```
  metadataproperty(@result, 'Content-Type') := 'text/xml; charset=utf-8';
  ```

- Any other response header can also be specified using this metadata approach. Use the header name as the metadata property name.

  ```
  metadataproperty(@result, 'Content-Language') := 'fr';
  ```

- A similar approach can be taken by a protocol author to define the http response code by annotation the global result property with a metadata property named "http-status-code".

```
metadataproperty(@result, 'Http-status-code') := '404';
```

- Use XML and JSON reader and writer components to convert from a textual input parameter to data records, and from data records to a textual result property.

Anonymous service access

There may be some services that you wish to make broadly available, bypassing the usual identification restrictions imposed by an authenticating server.

> **Note:** This facility is enabled only if a server administrator defines an anonymous user as a part of the Authentication configuration on the server.

If anonymous access is enabled in this way, when a user with no current valid session makes a request for a service denoted for anonymous access, the user is not prompted for credentials; instead, the request is permitted to execute under the credentials of the designated anonymous user.

To the end user, the effect is that they can make the request without providing credentials. Since they are running a server job as the anonymous user, the job is subject to whatever restrictions are imposed on that user.

> **Note:** Anonymous login events are logged in a way that allows you to track the number of different users who are running requests anonymously within any 24 hour period.

To designate a RESTful service that can be run anonymously, use the ANON modifier for the <url> directive.

```
<url REST ANON>
       method GET
       request /chem/img/@smiles
       protocol Protocols/Web Services/AcclDev/chem/Actions/compound/gen2D
       param __poolid chemimg
       <meta>
              summary  Stream a 2D chemical image derived from an input SMILES string.
       </meta>
</url>
```

See Appendix B: Configuration of Anonymous Access for information on the configuration of Anonymous access for an Pipeline Pilot administrator.

Modifying URL Routing with Parameters

It is possible to modify the URL Routing behavior with parameters (e.g., `_redirectFile`). However these cannot be added to the URL as with launchjob. Instead you must add these to `package.conf` configuration using the syntax from the following example:

```
<URL>
   method GET
   request /myprotocol/@smiles
   protocol <fullpathtoprotocol>
   param _redirectFile /\.pdf?$/
< /URL>
```

Chapter 3:
The Launchjob service

The Launchjob service is the principal entry point for REST web services that run protocols, and this section provides a reference to the service. (Some readers might be interested to know that it replaces the use of runjob.pl; in fact any invocation of this earlier service will be redirected to launchjob).

To make the best use of the URL routing described in the previous section, a good knowledge of the launchjob options is certainly useful, since this allows you to appropriately configure the defined services.

The URL to invoke the Launchjob service takes the following form (conventionally using the all lower case form "launchjob"):

```
http://host:port/auth/launchjob
```

The "/auth/" element of the URL indicates that all requests are processed by the authentication service that handles such things as session validation, browser login and single sign-on. When using this approach, once the authentication service completes its work, the session token is passed to the Launchjob service in the HTTP headers as a cookie.

Launchjob can also be invoked via URL rewriting, so you can design custom URIs to run a protocol and return results (see URL Routing).

Notes:

If anonymous access is enabled on the Pipeline Pilot server (see Appendix B for details), then launchjob requests may also be issued in a form that supports this facility:

`http://<host>:<port>/auth/anon/launchjob`

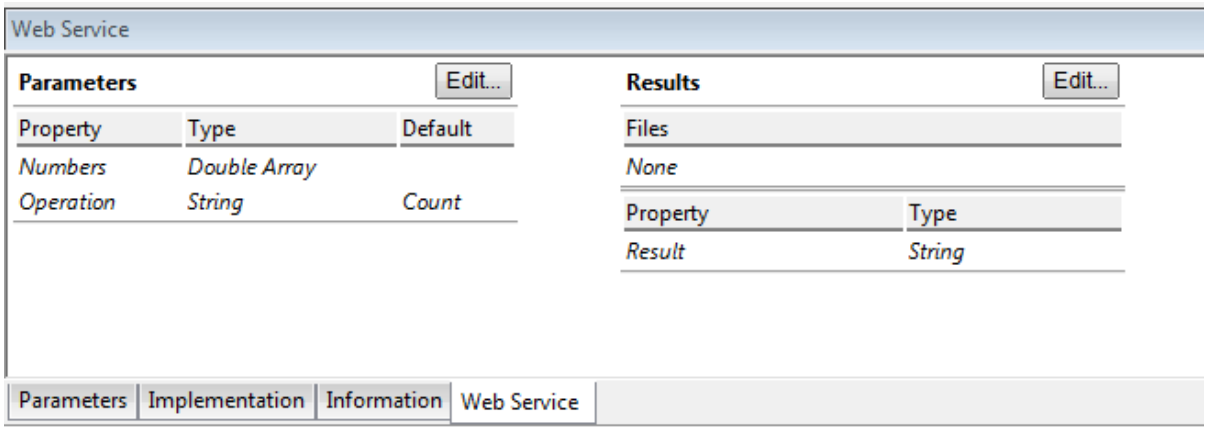
Anonymous access requires that Job Directory Access is set to Unrestricted in the Admin Portal setting Setup > Server Configuration > Job Directory Access

Launchjob parameters

The launchjob service executes Pipeline Pilot protocols. These protocols have parameters which expose the options for running the protocol. There are multiple ways to run a protocol and pass in parameter values, but when invoking a protocol with launchjob, simply pass the parameters as HTTP request parameters. In the case of GET method, this means in the query string of the request, following the question mark.

```
http://host:port/auth/launchjob?param1=val1&param2=val2&... etc.
```

When designing a protocol, the Web Service tab shows the parameters exposed on the protocol, and therefore the parameters of the web service. In the example below there are 2 parameters, named Numbers and Operation.



So the request to run the protocol as a REST service using an HTTP GET method, would look like this (should be a single line):

```
http://host:port/auth/launchjob
?
$protocol=Protocols/WebServices/Calc
&Operation=StdDev&Numbers=45.6&Numbers=53.5&Numbers=32.7&Numbers=50.1
```

The first parameter ($protocol) defines the protocol to run – this is an example of a service parameter, described fully below. The remaining query string parameter values (Operation and Numbers) are passed as parameters to the specified protocol. Note that the Numbers parameter is repeated, since it is an array value.

Note: For a GET method, the parameters are passed in the query string section of the URL, as shown above. For other HTTP methods (POST, PUT, etc.), the parameters are passed in the request body, usually in a form-encoded format. A protocol may also support a single protocol parameter passed as the body itself (e.g. a JSON text string) – see the $bodyParam service parameter in the following section.

To summarize:

IMPORTANT! Protocol parameters are represented as web service request parameters.

Launchjob service parameters

A Launchjob request supports a mix of protocol parameters and service parameters, for all methods, GET, PUT, etc.

Service parameters are identified by a single _ prefix or $ prefix, and are listed in the categorized sections below.

IMPORTANT! Protocol parameters should *not* have a _ or $ prefix to avoid conflicts with system parameters.

RUN MODE Parameters

The following section describes the parameters for running the protocol:

Parameter	Value	Details
`_blocking` `$blocking`	Boolean (0,1,y,n,t,f)	Defaults to true – blocking mode. In blocking mode, the request is serviced while the client waits on the request. The response and results (if any) are sent when the job completes. This behavior is also described as "synchronous". If this parameter is set to false, the job is launched in non-blocking (also known as polling or asynchronous) mode. In this case, the job is launched and left to run while the request returns immediately with the job id in the response (formatted according to the `_format` parameter, or the Accept header). The HTTP response code is 202 Accepted. The client may then poll at regular intervals to monitor the job status, retrieve the results when complete and finally to delete the job. See the separate section in this document on asynchronous job services.
`_progressMessage` `$progressMessage`	Text	If specified, a non-blocking protocol run is implied. Following the launch of the protocol, an HTML-based response (incorporating the specified message text) is returned to the browser, which includes a simple application to poll on the protocol job until it completes. The final job result or error information is then displayed. This is a useful option for long-running jobs linked from other documents.
`_log` `$log`	Boolean (0,1,y,n,t,f)	Defaults to true. If false, the job will not appear in the standard admin portal log. Designed for use by "utility" jobs.
`_expireJob` `$expireJob`	Integer (sec)	When a job requires retention after the job completes (e.g. it contains a result file to which is linked in the result data), the file is kept in existence for 10 minutes by default. This can be changed by including this parameter with a specified number of seconds.
`_keepJob` `$keepJob`	Boolean (0,1,y,n,t,f)	Defaults to false. If true, the job folder (if there is one) is retained upon job completion. Overrides any `_expireJob` setting.
`_timeout` `$timeout`	Integer (msec)	This setting is relevant to blocking jobs, and defines how long a request is serviced before a timeout error is triggered. By default, this value is defined at the server level by an administrator, but can be defined in milliseconds for a specific request by including this parameter.

Parameter	Value	Details
_onTimeout $onTimeout	One of {error,continue}	This parameter applies only to blocking jobs. The default timeout behavior for a blocking job is to terminate the job and return a 500 HTTP error code with a timeout error message in the response body. However, if the "continue" option is specified, then the job is defined as "trans-modal". It begins as a blocking job, but if the timeout duration is reached, the job is left to keep running. Meanwhile a 202 Accepted response code is returned to the HTTP client with a body containing information on the job id, identical to the information returned from the launch of a non-blocking job (see the _blocking parameter description above). The job can now be handled in the same way as a non-blocking, or asynchronous, job. See the separate section in this document on asynchronous job services, which can be used to poll the job for status, retrieve results and/or delete the job.
_compat $compat	Boolean (0,1,y,n,t,f)	Defaults to false. Compatibility mode for runjob.pl. See below for details.

REQUEST Parameters

The following section describes what to run:

Parameter	Value	Details
_protocol $protocol	Name, path or GUID	Indicates the protocol to be run. GUIDs should be URL-encoded to handle braces (web browsers will typically take care of this automatically). See the parameter notes below on alternative ways to specify the protocol identifier within the URL path itself.
_params $params	Pairs of parameter names and values in path format e.g. _params=/max/20/limit/False	Mostly useful for URL rewriting.
_bodyParam $bodyParam	The name of the parameter to hold body content.	The protocol parameter name to use for the body data (only relevant to POST, PUT, etc). Otherwise, maps to the first protocol parameter.
_passwordParams $passwordParams	Comma-separated list	Only relevant if there is a sensitive parameter being passed in and the output of the request is an HTML page. The default HTML output reports the input parameters. Any parameter in this list is excluded from the report.

RESPONSE Parameters

The following section describes how to construct the response:

Parameter	Value	Details
`_format` `$format`	One of {xml,json,text,html,debug} or a mime type like text/xml	When the output is a set of results, this defines the output format. When the output is a specific data value (using `_streamData` or `_streamFile`), the format can be used to define the response mime type. If not specified for the `_streamdata` scenario, the metadata of the result property can be set in the protocol and is then used to indicate the content-type. This is generally the preferable approach. When there is no explicit response format specification, auto-detection may also be deployed to figure out the appropriate mime type for streamed data content. The Accept HTTP header is also analyzed for guidance when there is no explicit formatting information available. If none of these approaches yields a usable type, then a simple text type is assumed. Note that under compatibility mode (see section below), the rules are slightly different (there is a bias toward using HTML).
`_streamData` `$streamData`	One of: 1. Protocol result property name. 2. If left blank, then the first or only property is used. 3. A value of * means that the result property list is sent back as the response. 4. A regular expression enclosed in forward slashes to match with a property name. Example: `/result\d\d/`	If defined using one of the specification schemes to the left, the identified result data value from the job is the response body. The metadata of the result property can be used to specify the content type, using a metadata property of 'content-type'. Alternatively, the `_format` parameter can be used to specify a mime type. The use of the `_format` parameter also applies to the * option to format the property list as XML, HTML, etc.

Parameter	Value	Details
`_streamFile` `$streamFile`	One of: 1. Protocol result file leaf name. 2. Incomplete matching names are OK too. 3. If left blank, then the first or only result file is used. 4. A regular expression enclosed in forward slashes to match with a file name. Example: `/\.pdf$/` 5. A series of regular expressions, which are attempted to match in the defined order. Example: `/\.pdf$/,/\.html?$/,/.+/`	If defined using one of the specification schemes to the left, the content of the identified result file is the response body. The `_format` parameter can be used to specify a mime type. Otherwise the file extension is used as a guide (for more common file types).
`_redirectFile` `$redirectFile`	One of: 1. Protocol result file leaf name. 2. Incomplete matching names are OK too. 3. If left blank, then the first or only result file is used. 4. A regular expression enclosed in forward slashes to match with a file name, example `/\.html?$/` 5. A series of regular expressions, which are attempted to match in the defined order. Example: `/\.pdf$/,/\.html?$/,/.+/`	If specified, the request response is a redirect instruction to the location of the result. This is most useful when the file contains relative links or images and no useful HTML BASE definition exists in a result HTML file.
`_allowCache` `$allowCache`	Boolean (0,1,y,n,t,f)	If set to true, then there is no attempt made to suppress browser-side caching. This can be useful when sending back static data such as stored images. Default value is false (i.e. no caching).

Note: Any launchjob service parameter that is not explicitly included in this parameter list is passed on to the job as a protocol parameter.

Job control parameters (using double __ prefix)

These are not unique to the Launchjob service, but are certainly relevant to its use. The following lists some more relevant job control parameters.

Parameter	Value	Details
__logJobName	Protocol name	The protocol name to be recorded in the usage log.
__poolID	Pool name	Runs job in the named scisvr pool.

Job parameters

For GET and DELETE methods (see below) all other parameters are passed to the invoked protocol as its parameter values. Since they are protocol parameters, such values are accessible to components within the protocol, at global scope.

Launchjob Service parameter notes

- For system parameters, the prefixes '$' and '_' are interchangeable.
- There is an alternative way to define the protocol to be run without using the _protocol parameter, by adding the protocol identifier to the URL path. All of the following examples will work (the name of the protocol is "popcount"):

```
http://host:port/auth/launchjob?_protocol=protocols/demographics/popcount
http://host:port/auth/launchjob/protocol/demographics/popcount
http://host:port/auth/launchjob/protocols/demographics/popcount
http://host:port/auth/launchjob?_protocol={1815D11A-B184-4229-8B8F-
65518C203F24}
http://host:port/protocol/demographics/popcount
http://host:port/protocols/demographics/popcount
http://host:port/protocol/{1815D11A-B184-4229-8B8F-65518C203F24}
```

- Debug mode (_format=debug) responds with a textual dump of the operation processing for diagnostic purposes.

Methods

The Launchjob service supports the following HTTP methods:

- GET
- POST
- PUT
- PATCH
- DELETE

GET, DELETE method

The simplest use of launchjob is to emulate the legacy runjob.pl GET method. Both service and protocol parameters are passed in the query section of the URL. The DELETE method can also extract query section protocol parameters.

Example that might be entered into a web browser:

```
http://host:port/auth/launchjob?_protocol=protocols/demographics/popcount&_
format=html
```

POST, PUT, PATCH, DELETE methods (with http body content)

Service and protocol parameters can be passed in the query section of the URL, as for GET methods. But the request body also contains the data to be passed to the protocol. The Content-Type header is critical when passing body content and is also supported for DELETE. See the `_bodyParam` parameter above as a way to identify the mapping of the body content to a protocol parameter.

Form encoded

In this case, the content-type header is:

`application/x-www-form-urlencoded`

The body is treated as a set of delimited tokens and the parameters are extracted and interpreted as described for GET method requests.

Notes:

- Multipart content is not currently supported.
- Protocol identification in the URI path is supported as described for the GET method.
- Parameters can also be extracted from the URL query string.

Formatted content type

For non-form-encoded body data, the content-type header may be one of:

- text/xml
- application/xml
- application/json
- text/plain

Character encoding types of UTF-8 and UTF-16 are supported.

Example of POSTing some JSON content to a protocol:

```
POST /auth/launchjob?_protocol=Protocols/zoo/animal/New HTTP/1.1
. . . (headers)
Content-Type: application/json; charset=utf-8
Content-Length: 207

[
{"animalname":"stickleback","type":"fish},
{"animalname":"jackdaw","type":"bird"}
]
```

In these cases:

- The body content is set to the first protocol parameter unless the `_bodyParam` parameter is used to indicate which protocol parameter should hold the body content.
- For the response, by default, the logic of `_streamData` applies and the first or only result property is streamed as the response content. As with other requests to the Launchjob service, the various response-related parameters can be used to modify this behavior.

HTTP headers

Accept header

As indicated in the `_format` parameter description above, the mime type value of the Accept header may be used to guide the format of the HTTP response, especially when returning a list of result values. Consistent with the `_format` parameter, supported formats are XML, JSON and HTML.

Compatibility mode

Compatibility mode is invoked with the `_compat` flag set to true. It provides behavior familiar to users of the legacy `runjob.pl` service. So what are the consequences of using this mode?

1. If neither `_blocking` nor `_progressMessage` are defined explicitly, the protocol is run under a "pseudo-blocking" mode in which the job is launched and polled asynchronously from within the Launchjob service.
2. If there are no settings for the response behavior (`_streamData`, `_streamFile`, `_redirectFile`), the compatibility mode behavior is to try each of the following in order:
   a. Redirect to the first .pdf result file, if there is one.
   b. Redirect to the first .htm or .html result file, if there is one.
   c. Redirect to the first result file of any other type, if there is one.
   d. Stream back information on all the result data.
3. If there is no `_format` setting defined, compatibility mode will use HTML for case 2d above.

Chapter 4:
Asynchronous mode for long-running jobs

For jobs that may run for an extended period of time, a client may employ a model of execution where the job is launched with one request, and then further requests are made to poll the status of the job (and potentially other information), until completion, when the final results can be retrieved and the job released. This approach avoids tying up a connection while the job runs, can allow a client to disconnect and reconnect to the job at a later time, and can also provide more information to the client for a running job.

The starting point for an asynchronous job run is the "`_blocking=false`" parameter to launchjob, described in the previous section. This parameter can also be incorporated into URL routing. When job blocking mode is disabled in this way, the job is started, but the job launch request then returns with a 202 Accepted HTTP response, without waiting for the job to complete. The job ID is always returned in the response body and this job ID is then used in subsequent service requests to get job information or to perform operations upon the job, using the job URL set listed below.

Note that a blocking job may also convert to an asynchronous mode of operation if the `onTimeout=continue` parameter value is passed to launchjob when requesting a blocking job. In this case, the job begins as synchronous; but if it does not complete before the timeout duration, a 202 Accepted HTTP response is returned to the client to indicate that the job is still running, in an asynchronous mode.

To enable a client to monitor and manage the running asynchronous job, the following job management URL set is published, where *<job-id>* should be replaced with the ID of the job returned from the Launchjob service. Note that in general the server will delete a job once any useful results have been retrieved, unless the "`_keepJob=true`" parameter is appended as a query string to the job results request. In this case, to remove the job, use the job deletion request detailed in the list below.

HTTP verb	Path	Description
GET	`/jobs/`	Responds with a list all jobs for the current user
GET	`/jobs/<job-id>`	Response contains details for the current job, including status, status code and progress message, if there is one. Replace *<job-id>* with the ID of the job returned from the Launchjob service.
GET	`/jobs/<job-id>/status`	Response is the status for the specified job. Possible values are: ■ **Initializing:** Startup phase ■ **Running:** Job is running ■ **Complete:** Job is successfully completed. Final results are available. ■ **Terminated:** Job ended prematurely by the owner or an administrator. ■ **Error:** Job ended with an error. ■ **Paused:** Job is waiting for client input. Unlikely to happen for a job not run from the Pro Client.

HTTP verb	Path	Description
GET	/jobs/<*job-id*>/files/	Responds with a list of links to files in the job directory.
GET	/jobs/<*job-id*>/result	When this request references a completed job, the response will contain the result of the job as defined when the job was initially launched. See the section on Launchjob Response parameters above for information about the options for defining the way in which a job result is defined. The job will then be marked for deletion unless the query string "?_keepJob=true" is appended to the request. For a job with an error status, the results will include error information.
GET	/jobs/<*job-id*>/result/all	Returns a listing of the job results, once the job is complete. This form of the result request will return a list of all job results, regardless of the result specification in the original Launchjob request.
DELETE	/jobs/<*job-id*>	Release the job, terminating it if necessary. This is only needed if you have not requested the job results or have used the _keepJob query string parameter.
DELETE	/jobs/<*job-id*>/stop	Stop the running job.

Notes:

- By default, responses are formatted as XML. Other formats are available using a query string such as ?_format=json. See the information above for the _format parameter for launchjob.
- After retrieving the job results, the job and its results are marked for deletion on the server. In practice this may be immediate or there may be a delay of several minutes if the job contains result files which may need to be downloaded. This is consistent with blocking jobs, which are by default marked for deletion once the job is complete and results returned. The underlying job management service has flexible behaviors in this area (e.g., job expiration time, described in the Launchjob parameters section). Although they are not currently exposed via the RESTful /jobs/ interface described above, but can be expressed as query string parameters.
- This job management service can be used against asynchronous jobs launched by means other than launchjob (e.g. via SOAP) provided the job identifier is known.

Appendices

Appendix A: HTTP Request Parameter Variables

HTTP headers:

HTTP_USER_AGENT
HTTP_REFERER
HTTP_COOKIE
HTTP_FORWARDED
HTTP_HOST
HTTP_PROXY_CONNECTION
HTTP_ACCEPT
%{HTTP:header}

Connection and Request:

REMOTE_ADDR
REMOTE_HOST
REMOTE_PORT
REMOTE_USER
REMOTE_IDENT
REQUEST_METHOD
SCRIPT_FILENAME
PATH_INFO
QUERY_STRING
AUTH_TYPE
%{SSL:variable}

Server Internals:

DOCUMENT_ROOT
SERVER_ADMIN
SERVER_NAME
SERVER_ADDR
SERVER_PORT
SERVER_PROTOCOL
SERVER_SOFTWARE
%{ENV:variable}

Date and Time:

TIME_YEAR
TIME_MON
TIME_DAY
TIME_HOUR
TIME_MIN
TIME_SEC
TIME_WDAY
TIME

Rewrite special variables:

API_VERSION
THE_REQUEST
REQUEST_URI
REQUEST_FILENAME
IS_SUBREQ
HTTPS

Appendix B: Configuration of Anonymous Access

Enabling Anonymous Execution

Anonymous execution is not enabled by default. To switch on this feature, an administrator must navigate to the Security/Authentication page in the Administration Portal. On this page is an "Anonymous Access" section to define the credentials of the anonymous user. Once defined, all jobs run anonymously will be making use of this user identity.

Notes:

If anonymous access is enabled, then launchjob requests may also be issued in a form that supports this facility:

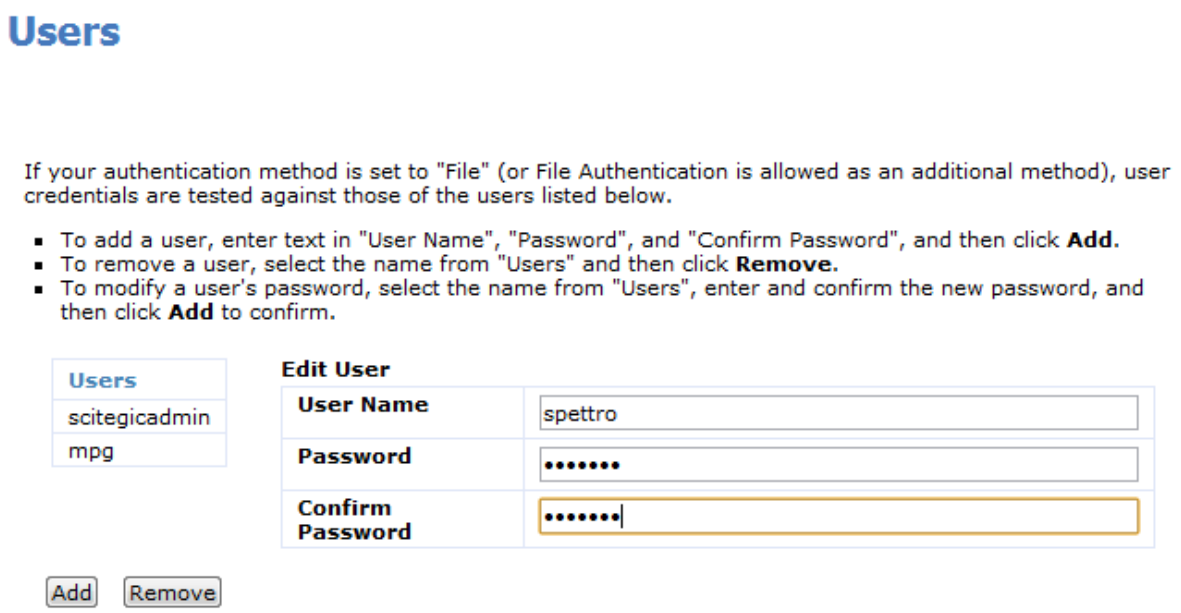
`http://<host>:<port>/auth/anon/launchjob`

Anonymous access requires that Job Directory Access is set to Unrestricted in the Admin Portal setting Setup > Server Configuration > Job Directory Access

The Anonymous User

The user designated for anonymous usage must be a user that is able to authenticate against the server, and so the set of possible users is defined by the method of authentication in force. If you change the authentication method, you might also need to update the anonymous user credentials.

If the authentication method is DOMAIN, then the anonymous user should be a domain user. However, note that you can also enable FILE authentication in addition to one of the other methods. Under this scenario, you can create a special-purpose user and password in the Security/Users page of the Administration Portal, and then designate this user as your anonymous user. This avoids the need to define an artificial user in the corporate directory for anonymous access. Instead you can define this special user name in the Users settings.



Appendix C: Alternative specification for a request property

As indicated in the URL routing specification section, there is an alternative approach to specifying variable tokens the request specification. This is described in this appendix.

1. **Basic format**

   The simplest specification is a URL path that matches a request with an identical path. The URL is a reference to a collection of items and has a trailing slash by convention.

   Example:

   ```
   request        /zoology/animals/
   ```

   The full request from the client would be something like this:

   `http://localhost:9944/zoology/animals/`
   `https://localhost:9943/zoology/animals/`

2. **Path element tokens**

   To add some flexibility, you can specify a named variable token in the path specification. During URL path matching, this tokenized section of the path is treated as variable. It can be different in each request. These path element tokens are specified using the '@' prefix and always match within a single path element (i.e. in between a pair of forward slashes, or following the final slash of the URL path). The token can be specified as a whole or partial path element. When a match is found, a property is created with the name of the token and the content in the path that maps to the token is assigned as the property value. The property is passed on to the protocol that handles the request (this should be defined as a protocol parameter).

   Examples:

   ```
   request        /zoology/animals/@id
   ```

   In the above example, a request path of

   `/zoology/animals/u634`

   results in the protocol parameter "`id`" being set to the value "`u634`".

```
request          /zoology/animals/@name/update
```

Here, a request path of

`/zoology/animals/nargle/update`

results in the protocol parameter "`name`" being set to the value "`nargle`".

Notes:

- A forward slash (/) or end of the URL path is treated as the termination of a token.
- Use the format `@'...'` (with single quotes) for cases where the simple variable form in inadequate (e.g. variable names with spaces, variable names followed by other non / characters.
- To indicate a more complex regular expression for parameter matching, see the constraint property below.

More Examples:

```
request                  /zoology/animals/Bird@id

request                 /zoology/animals/@'zoo handle'

request                 /zoology/animals/gen@genus/@id

request                 /zoology/animals/@'genus'G/@id
```

3. **Route globbing tokens**

   Route globbing is a way to specify that a particular token should be matched to multiple elements of a URL, and therefore these tokens can cross path element boundaries. These globbing tokens are specified using the * prefix.

   Example:

   ```
   request          /zoology/animals/@id/*special
   ```

   In the above example, a request path of

   `/zoology/animals/104/legs/eggs/gestation`

   results in the protocol parameter "`id`" being set to the value "`104`", and the protocol parameter "`special`" being set to the value "`legs/eggs/gestation`".

   Additional Examples:

   ```
   request                  /zoology/animals/*gendetails/@id

   request                 /zoology/animals/*'gen details'/@id
   ```

4. **Parameter extraction globbing**

   When the glob token has the value **, the matching path elements are treated as a series of name value pairs that are passed to the protocol. The parameter extracting ** token should only be used once in a request path specification.

   Example:

   ```
   request          /zoology/animals/@id/**
   ```

   In the above example, a request path of

   `/zoology/animals/unicorn/db/zoo3/fmt/png`

results in the protocol parameter "`id`" being set to the value "`unicorn`", and the protocol parameter "`db`" being set to the value "`zoo3`", and the protocol parameter "`fmt`" being set to the value "`png`".