Taverna 2.3 Server: Usage and API Guide

This document relates to the release of Taverna Server that is based on the Taverna 2.3 Platform, from the myGrid team at the University of Manchester.

About

This release is a feature-complete version of the Taverna 2 Server that is provided as a basis for deployments of server-ized Taverna in a multi-user environment.

In addition to its improved performance, this release supports a number of new key features:

- Based on **Taverna 2.3**
- **Multiple users**, with strong separation between them.
- Limited **persistence over service restarts**, depending on exact deployment.
- **Workflow run introspection** capabilities; clients can ask the server what inputs they should supply and what outputs were provided.
- **Workflow run termination notifications** through multiple mechanisms (RSS feed, email, SMS, twitter, etc. depending on deployment).
- **Security**, both of access to the service and access by the workflow runs to other services.
- **Administrative REST interface** including resource accounting
- Streaming of **large files** both for download and upload.

This is in addition to the features supported by the previous version of Taverna 2 Server:

- **Upload and Execution of arbitrary Taverna 2 workflows**
- **Access to Workflow's Interim Output Files**; no need to wait for the workflow to finish if the results are available sooner
    - **Safe File Management** for handling results; workflows cannot interfere with each others files
- Simple mechanism for **Removal of Expired Workflows**
- Support for both **RESTful and SOAP APIs**, for easier tooling
- **JMX-based Management API**

Conceptual interface

Conceptually, an instance of Taverna Server exists to manage a collection of workflow runs, as well as some global information that is provided to all on the server's general capabilities. The server also supports an overall Atom feed per user that allows you to find out when your workflows terminate without having to poll each one separately. This feed is at *https://«SERVER:PORT»/taverna-server/feed* (with the default web-application name). The feed is not available to anonymous users, and will only accept updates from the internal notification mechanism.

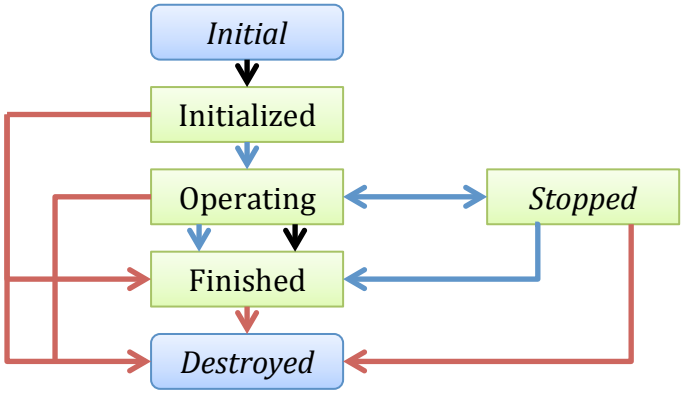
Each workflow run is associated with a working directory that is specific to that run; the name of the working directory is a value that is not repeated for any other run. Within the working directory, these[1] subdirectories will be created:

conf	Contains optional additional configuration files for the Taverna execution engine; empty by default.
externaltool	Contains optional additional configuration files for the external tool plugin; empty by default.
lib	Contains additional libraries that will be made available to beanshell scripts; empty by default.
logs	Location that logs will be written to. In particular, will eventually contain the file *detail.log*, which can be very useful when debugging a workflow.
out	Location that output files will be written to if they are not collected into a Baclava file. This directory is only created during the workflow run; it should not be made beforehand.
plugins	Contains the additional plug-in code that is to be supported for the specific workflow run.
t2-database	Contains the database working files used by the Taverna execution engine.

All file access operations are performed on files and directories beneath the working directory. The server prevents all access to directories outside of that, so as to promote proper separation of the workflow runs. (Note in particular that the credential manager configuration directory will not be accessible; it is managed directly by the server.)

Associated with each workflow run is a state. The state transition diagram is this:

[1] Each run also has *repository* and *var* directories created for it; their purpose is not documented and they are initially empty.



The blue states are the initial and final states, and all states in *italic* cannot be observed in practice. The black arrows represent automatic state changes, the blue arrows are for manually-triggered transition, and the red arrows are destructions, which can be done from any state (other than the initial unobservable one) and which may be either manually or automatically triggered; automatic destruction happens when the run reaches its expiry time (which you can set but cannot remove). Note that there are two transitions from *Operating* to *Finished*; they are not equivalent. The automatic transition represents the termination of the workflow execution with such outputs produced as are going to be generated, whereas the manual transition is where the execution is killed and outputs may be not generated even if they conceptually existed at that point. Also note that it is only the transition from *Initialized* to *Operating* that represents the start of the workflow execution engine.

Each workflow run is associated with a unique identifier, which is constant for the life of the run. This identifier is used directly by the SOAP interface and forms part of the URI for a run in the REST interface, but it is the *same* between the two. Any run may be accessed and manipulated via either interface, so long as the right identifier is used and you have permission to do the action concerned. The permissions associated with a run are the ability to *read* features of the run and files associated with it, the ability to *update* features (including creating files), and the ability to control the lifespan of a run and *destroy* it, each of which implies the ones before it as well. The owner of a run (i.e., the user who created it) always has all those permissions, and can also manipulate the security configuration of the run — these permissions and any credentials granted to the run such as passwords and X.509 key-pairs — which are otherwise totally shrouded in the execution interface. The permissions of a user to access a particular run can also be set to *none*, which removes all granted permissions and restores the default (no access granted at all).

Associated with each run are a number of listeners. This release of the server only supports a single listener, "*io*", which is applied automatically. This listener is responsible for detecting a number of technical features of the workflow run and exposing them. In particular, it reports any output produced by the workflow engine on either *stdout* or *stderr*, what the result ("*exitcode*") would be, where to send termination notifications to ("*notificationAddress*") and what resources were used during the workflow run ("*usageRecord*").

The (RESTful) Usage Pattern

The Taverna 2 Server supports both REST and SOAP APIs; you may use either API to access the service and any of the workflow runs hosted by the service. The full service descriptions are available at *http://«SERVER:PORT»/taverna-server/services* but to illustrate their use, here's a sample execution using the REST API.

1. The client starts by creating a workflow run. This is done by POSTing a T2flow document to the service at the address *http://«SERVER:PORT»/taverna-server/rest/runs*; may be either wrapped with XML or an unwrapped T2flow document (provided the right HTTP content-type is used).

   When using the wrapped form, the wrapping of the submitted document is a single XML element, *workflow* in the namespace *http://ns.taverna.org.uk/2010/xml/server/*, and the workflow (as saved by the Taverna Workbench) is the child element of that.

   The result of the POST is an *HTTP 201 Created* that gives the location of the created run (in a *Location* header), hereby denoted the *«RUN_URI»* (it includes a UUID which you will need to save in order to access the run again, though the list of known UUIDs can be found above). Note that the run is not yet actually doing anything.

2. Next, you need to set up the inputs to the workflow ports. This is done by either uploading a file that is to be read from, or by directly setting the value.

   Directly Setting the Value of an Input

   To set the input port, *FOO*, to have the value *BAR*, you would PUT a message like this to the URI *«RUN_URI»/input/input/FOO*

   ```
   <t2sr:runInput xmlns:t2sr=
           "http://ns.taverna.org.uk/2010/xml/server/rest/">
       <t2sr:value>BAR</t2sr:value>
   </t2sr:runInput>
   ```

   Uploading a File for One Input

   The values for an input port can also be set by means of creating a file on the server. Thus, if you were staging the value BAR to input port FOO by means of a file BOO.TXT then you would first POST this message to *«RUN_URI»/wd*

   ```
   <t2sr:upload t2sr:name="BOO.TXT" xmlns:t2sr=
           "http://ns.taverna.org.uk/2010/xml/server/rest/">
       QkFS
   </t2sr:upload>
   ```

   Note that "*QkFS*" is the base64-encoded form of "*BAR*", and that each workflow run has its own working directory into which uploaded files are placed; you are never told the name of this working directory.

You can also PUT the contents of the file (as application/octet-stream) directly to the virtual resource name that you want to create the file as; for the contents "BAR" that would be three bytes 66, 65, 82 (with appropriate HTTP headers). This particular method supports upload of very large files if necessary.

Once you've created the file, you can then set it to be the input for the port by PUTting this message to *«RUN_URI»/input/input/FOO*

```
<t2sr:runInput xmlns:t2sr=
        "http://ns.taverna.org.uk/2010/xml/server/rest/">
    <t2sr:file>BOO.TXT</t2sr:file>
</t2sr:runInput>
```

Note the similarity of the final part of this process to the previous method for setting an input.

You can also create a directory, e.g., *IN*, to hold the input files. This is done by POSTing a different message to *«RUN_URI»/wd*

```
<t2sr:mkdir t2sr:name="IN" xmlns:t2sr=
        "http://ns.taverna.org.uk/2010/xml/server/rest/"
    />
```

With that, you can then create files in the *IN* subdirectory by sending the upload message to *«RUN_URI»/wd/IN* and you can use the file as an input by using a name such as *IN/BOO.TXT*. You can also create sub-subdirectories if required by sending the *mkdir* message to the natural URI of the parent directory, just as sending an upload message to that URI creates a file in that directory.

Using a File Already on the Taverna Server Installation

You can use an existing file attached to a workflow run *on the same server*, provided you have permission to access that run. You do this by using a PUT to set the input to a reference (the actual URL below is just an example, but it must be the full URL to the file):

```
<t2sr:runInput xmlns:t2sr=
        "http://ns.taverna.org.uk/2010/xml/server/rest/">
    <t2sr:reference>
        «OTHER_RUN_URI»/wd/file.name
    </t2sr:reference>
</t2sr:runInput>
```

The data will be copied across efficiently into a run-local file. This version of Taverna Server does not support accessing files stored on any other server or on the general web via this mechanism.

Uploading a Baclava File

The final way of setting up the inputs to a workflow is to upload (using the same method as above) a Baclava file (e.g., *FOO-BAR.BACLAVA*) that describes the inputs. This is then set as the provider for all inputs by PUTting the name of the Baclava file (as plain text) to *«RUN_URI»/input/baclava*

3. If your workflow depends on external libraries (e.g., for a beanshell or API consumer service), these should be uploaded to *«RUN_URI»/wd/lib*; the name of the file that you create there should match that which you would use in a local run of the service.

4. If the workflow refers to a secured external service, it is necessary to supply some additional credentials. For a SOAP web-service, these credentials are associated in Taverna with the WSDL description of the web service. The credentials *must* be supplied before the workflow run starts.

   To set a username and password for a service, you would POST to *«RUN_URI»/security/credentials* a message like this (assuming that the WSDL address is "*https://host/serv.wsdl*", that the username to use is "*fred123*", and that the password is "*ThePassWord*"):

```
<t2sr:credential xmlns:t2sr=
            "http://ns.taverna.org.uk/2010/xml/server/rest/"
        xmlns:t2s="http://ns.taverna.org.uk/2010/xml/server/">
    <t2s:userpass>
        <t2s:serviceURI>https://host/serv.wsdl</t2s:serviceURI>
        <t2s:username>fred123</t2s:username>
        <t2s:password>ThePassWord</t2s:password>
    </t2s:userpass>
</t2sr:credential>
```

   For REST services, the simplest way to find the correct security URI to use with the service is to run a short workflow against the service in the Taverna Workbench and to then look up the URI in the credential manager.

5. Now you can start the workflow running. This is done by using a PUT to set *«RUN_URI»/status* to the plain text value *Operating*.

6. Now you need to poll, waiting for the workflow to finish. To discover the state of a run, you can (at any time) do a GET on *«RUN_URI»/status*; when the workflow has finished executing, this will return *Finished* instead of *Operating* (or *Initialized*, the starting state).

   There is a fourth state, *Stopped*, but it is not supported in this release.

7. Every workflow run has an expiry time, after which it will be destroyed and all resources (i.e., local files) associated with it cleaned up. By default in this release, this is 20 minutes after initial creation. To see when a particular run is scheduled to be disposed of, do a GET on *«RUN_URI»/expiry*; you may set the time when the run is disposed of by PUTting a new time to that same URI. Note that this includes not just the time when the workflow is executing, but also when the input files are being created beforehand and when the results are being downloaded afterwards; you are advised to make your clients regularly advance the expiry time while the run is in use.

8. The outputs from the workflow are files created in the out subdirectory of the run's working directory. The contents of the subdirectory can be read by doing a GET on *«RUN_URI»/wd/out* which will return an XML document describing the contents of the directory, with links to each of the files within it. Doing a GET on those links will retrieve the actual created files (as uninterpreted binary data).

Thus, if a single output *FOO.OUT* was produced from the workflow, it would be written to the file that can be retrieved from *«RUN_URI»/wd/out/FOO.OUT* and the result of the GET on *«RUN_URI»/wd/out* would look something like this:

```
<t2sr:directoryContents
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:t2s="http://ns.taverna.org.uk/2010/xml/server/"
        xmlns:t2sr=
            "http://ns.taverna.org.uk/2010/xml/server/rest/">
    <t2s:file xlink:href="«RUN_URI»/wd/out/FOO.OUT"
            t2sr:name="FOO.OUT">out/FOO.OUT</t2s:file>
</t2sr:directoryContents>
```

9. The standard output and standard error from the T2 Command Line Executor subprocess can be read via properties of the special I/O listener. To do that, do a GET on *«RUN_URI»/listeners/io/properties/stdout* (or *.../stderr*). Once the subprocess has finished executing, the I/O listener will provide a third property containing the exit code of the subprocess, called *exitcode*.

   Note that the supported set of listeners and properties will be subject to change in future versions of the server, and should not be relied upon.

10. Once you have finished, destroy the run by doing a DELETE on *«RUN_URI»*. Once you have done that, none of the resources associated with the run (including both input and output files) will exist any more. If the run is still executing, this will also cause it to be stopped.

All operations described above have equivalents in the SOAP service interface.

API of the REST Interface

Note that schemas in this document are actually pseudo-schemas. For example, this shows how the various marks on attributes and elements indicate their cardinality and type:

```
<element requiredAttr="xsd:someType">
    <requiredChildElement />
    <zeroOrMoreChildren /> *
    <optionalChild /> ?
    <alternative1 /> | <alternative2 />

    <childWithSimpleStringContent>
        xsd:string
    </childWithSimpleStringContent>
    <childWithUndescribedContent ... />
</element>
```

To be exact, a suffix of "*" marks an element that can be repeated arbitrarily often, a suffix of "?" marks an element or attribute that can be either present or absent, and otherwise exactly one of the element is required (or, for attributes, the attribute must be present). We never use cardinalities other than these, and order is always respected. Where there is complex content, it will either be described inline or separately. Where there is a choice between two elements, they are separated by a "|" character.

Namespaces are always defined as follows; their definitions are omitted from the pseudoschemas:

Prefix	Namespace URI
t2flow	http://taverna.sf.net/2008/xml/t2flow
t2s	http://ns.taverna.org.uk/2010/xml/server/
t2sr	http://ns.taverna.org.uk/2010/xml/server/rest/
port	http://ns.taverna.org.uk/2010/port/
xlink	http://www.w3.org/1999/xlink
xsd	http://www.w3.org/2001/XMLSchema

Main Server Resource

Resource: /

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK
Notes: not secured

Retrieves a description of the server as either XML or JSON (determined by content negotiation) that indicates other locations to find sub-resources.

```
<t2sr:serverDescription
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string" >
    <t2sr:runs xlink:href="xsd:anyURI" />
    <t2sr:policy xlink:href="xsd:anyURI" />
    <t2sr:feed xlink:href="xsd:anyURI" />
</t2sr:serverDescription>
```

The *feed* element is a pointer to the location of the Atom feed for events issued to the user about things like the termination of their workflows.

Resource: /runs

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Retrieve a list of all runs that the current user can see, as XML or JSON. Note that the user will not be told about runs that they are not permitted to see (i.e., that they didn't create and haven't been given permission to see). Deleted runs will also not be present.

```
<t2sr:runList>
    <t2sr:run xlink:href="xsd:anyURI" /> *
</t2sr:runList>
```

Method: POST
Consumes: application/xml, application/vnd.taverna.t2flow+xml

Produces: N/A
Response codes: 201 Created

Accepts (or not) a request to create a new run executing the given workflow. When the content type is XML, the workflow must be wrapped inside an `{http://ns.taverna.org.uk/2010/xml/server/}workflow` element.

```
<t2s:workflow>
    <t2flow:workflow ... />
</t2s:workflow>
```

When the content type is not simple XML, it must be the literal document in the format as defined by Taverna.

The result will be a redirect (via *Location:* HTML header) to the resource created for this particular run. The run will be in the *Initialized* state, with a default lifetime.

Resource: /policy

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK
Notes: not secured

Describe the (public) parts of the policies of this server, as XML or JSON.

```
<t2sr:policyDescription
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string" >
    <t2sr:runLimit xlink:href="xsd:anyURI" />
    <t2sr:permittedWorkflows xlink:href="xsd:anyURI" />
    <t2sr:permittedListeners xlink:href="xsd:anyURI" />
    <t2sr:enabledNotificationFabrics xlink:href="xsd:anyURI" />
</t2sr:policyDescription>
```

Resource: /policy/enabledNotificationFabrics

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK
Notes: not secured

Gets the list of supported, enabled notification fabrics. Each corresponds (approximately) to a protocol, e.g., email.

Resource: /policy/permittedListenerTypes

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK
Notes: not secured

Gets the list of permitted event listener types.

Resource: /policy/permittedWorkflows

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK
Notes: not secured

Gets the list of permitted workflows. An empty list indicates that *all* workflows are permitted.

Resource: /policy/runLimit

Method: GET
Consumes: N/A
Produces: text/plain (xsd:int)
Response codes: 200 OK
Notes: not secured

Gets the maximum number of simultaneous runs that the user may create. Note that this is an upper bound; other resource contention may cause the actual number to be lower.

Per-Workflow Run Resource

Note that all of these resources require that the user be authenticated and permitted to (at least) see the run.

Resource: /runs/{id}

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Describes the sub-resources associated with this workflow run, as XML or JSON.

```
<t2sr:runDescription t2sr:owner="xsd:string"
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string" >
    <t2sr:expiry xlink:href="xsd:anyURI">
        xsd:string
    </t2sr:expiry>
    <t2sr:creationWorkflow xlink:href="xsd:anyURI" />
    <t2sr:createTime xlink:href="xsd:anyURI" />
    <t2sr:startTime xlink:href="xsd:anyURI" />
    <t2sr:finishTime xlink:href="xsd:anyURI" />
    <t2sr:status xlink:href="xsd:anyURI" />
    <t2sr:workingDirectory xlink:href="xsd:anyURI" />
    <t2sr:inputs xlink:href="xsd:anyURI" />
    <t2sr:output xlink:href="xsd:anyURI" />
    <t2sr:securityContext xlink:href="xsd:anyURI" />
    <t2sr:listeners xlink:href="xsd:anyURI">
```

```
        <t2sr:listener xlink:href="xsd:anyURI" /> *
    </t2sr:listeners>
</t2sr:runDescription>
```

Method: DELETE
Consumes: N/A
Produces: N/A
Response codes: 204 No Content
Notes: requires Destroy permission

Deletes a workflow run, cleaning up all resources associated with that run.

Resource: /runs/{id}/expiry

Method: GET
Consumes: N/A
Produces: text/plain (xsd:dateTime)
Response codes: 200 OK

Gives the time when the workflow run becomes eligible for automatic deletion.

Method: PUT
Consumes: text/plain (xsd:dateTime)
Produces: text/plain (xsd:dateTime)
Response codes: 200 OK
Notes: requires Destroy permission

Sets the time when the workflow run becomes eligible for automatic deletion. Note that the deletion does not necessarily happen at exactly that time; that depends on the internal mechanisms of the server.

Resource: /runs/{id}/createTime

Method: GET
Consumes: N/A
Produces: text/plain (xsd:dateTime)
Response codes: 200 OK

Gives the time when the workflow run was first submitted to the server.

Resource: /runs/{id}/finishTime

Method: GET
Consumes: N/A
Produces: text/plain (xsd:dateTime *or* empty)
Response codes: 200 OK

Gives the time when the workflow run was detected as having finished executing, or the empty string if the workflow run has not yet finished.

Resource: /runs/{id}/startTime

Method: GET
Consumes: N/A

Produces: text/plain (xsd:dateTime *or* empty)
Response codes: 200 OK

Gives the time when the workflow run was started, or the empty string if the workflow run has not yet been started.

Resource: /runs/{id}/status

Method: GET
Consumes: N/A
Produces: text/plain ("*Initialized*" or "*Operating*" or "*Stopped*" or "*Finished*")
Response codes: 200 OK

Gives the current status of the workflow run. Note that the "*Stopped*" state is not currently used.

Method: PUT
Consumes: text/plain ("*Initialized*" or "*Operating*" or "*Stopped*" or "*Finished*")
Produces: text/plain ("*Initialized*" or "*Operating*" or "*Stopped*" or "*Finished*")
Response codes: 200 OK

Attempts to update the status of the workflow run. This is the only mechanism for setting a workflow run operating, and an operating run can be cancelled by forcing it into the finished state.

Resource: /runs/{id}/workflow

Method: GET
Consumes: N/A
Produces: application/vnd.taverna.t2flow+xml, application/xml, application/json
Response codes: 200 OK

Gives the workflow document used to create the workflow run, as raw t2flow, wrapped XML or wrapped JSON. (Note that the last is not supported for re-upload.)

Resource: /runs/{id}/input

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Describe the sub-URIs relating to workflow inputs.

```
<t2sr:runInputs
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string" >
    <t2sr:expected xlink:href="xsd:anyURI" />
    <t2sr:baclava xlink:href="xsd:anyURI" />
    <t2sr:input xlink:href="xsd:anyURI" /> *
</t2sr:runInputs>
```

Resource: /runs/{id}/input/baclava

Method: GET
Consumes: N/A
Produces: text/plain
Response codes: 200 OK

> Gives the Baclava file describing the inputs, or empty if individual files are used. The name is relative to the main working directory.

Method: PUT
Consumes: text/plain
Produces: text/plain
Response codes: 200 OK

> Sets the Baclava file describing the inputs. The name is relative to the main working directory, and must not be empty.

Resource: /runs/{id}/input/expected

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

> Describe the *expected* inputs of this workflow run. They must be supplied by either per-input specifications or by the baclava file.

```
<port:inputDescription
        port:workflowId="xsd:string"
        port:workflowRun="xsd:anyURI"
        port:workflowRunId="xsd:string">
    <port:input port:name="xsd:string" port:depth="xsd:int"?
            xlink:href="xsd:anyURI"? /> *
</port:inputDescription>
```

Resource: /runs/{id}/input/input/{name}

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

> Gives a description of what is used to supply a particular input, which will be either a literal value or the name of a file in the working directory or a reference to a file maintained by another workflow run on the same server (which will be copied efficiently if the user has permission).

```
<t2sr:runInput t2sr:name="xsd:string"
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string" >
    <t2sr:file> xsd:string </t2sr:file>
  |
    <t2sr:reference> xsd:anyURI </t2sr:reference>
  |
    <t2sr:value> xsd:string </t2sr:value>
</t2sr:runInput>
```

Method: PUT
Consumes: application/xml, application/json
Produces: application/xml, application/json
Response codes: 200 OK

Sets the source for a particular input port (and cancels any use of baclava to supply that port). The document format for both the consumption and production side of this operation is as above.

Resource: /runs/{id}/output

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Gives a description of the outputs (in XML or JSON) as currently understood. Note that only very limited understanding of the outputs will be present before the workflow is run; the majority of information is generated *during* the execution of the run.

```
<port:workflowOutputs
        port:workflowId="xsd:string"
        port:workflowRun="xsd:anyURI"
        port:workflowRunId="xsd:string">
    <port:output port:name="xsd:string"
            port:depth="xsd:int"? >
        <port:value xlink:href="xsd:anyURI"?
                port:fileName="xsd:string"
                port:contentType="xsd:string"
                port:byteLength="xsd:int" />
      |
        <port:list xlink:href="xsd:anyURI"?
                port:length="xsd:int"? >
            <!-- Sequence of values, just as for a port -->
        </port:list>
      |
        <port:error xlink:href="xsd:anyURI"?
                port:depth="xsd:int"? />
      |
        <port:absent xlink:href="xsd:anyURI"? />
    </port:output> *
</port:workflowOutputs>
```

Method: GET
Consumes: N/A
Produces: text/plain
Response codes: 200 OK

Gives the Baclava file where output will be written (relative to the working directory); empty means use multiple simple files in the *out* directory.

Method: PUT
Consumes: text/plain
Produces: text/plain
Response codes: 200 OK

Sets the Baclava file where output will be written (relative to the working directory); empty means use multiple simple files (in a directory structure where lists of lists are involved) in the *out* directory.

Resource: /runs/{id}/listeners

The current implementation does not permit installing new listeners, and comes with a single listener called *io* which provides the *stdout*, *stderr* and *exitcode* properties, all of which do not permit update. This means that the standard output of the workflow run is available at */runs/{id}/listeners/io/properties/stdout*.

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Get a list of the listeners installed in the workflow run.

```
<t2sr:listeners
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string" >
    <t2sr:listener xlink:href="xsd:anyURI"
            t2sr:name="xsd:string"
            t2sr:type="xsd:string"
            t2s:serverVersion="xsd:string"
            t2s:serverRevision="xsd:string"
            t2s:serverBuildTimestamp="xsd:string">
        <t2sr:configuration xlink:href="xsd:anyURI" />
        <t2sr:properties>
            <t2sr:property xlink:href="xsd:anyURI"
                    t2sr:name="xsd:string" /> *
        </t2sr:properties>
    </t2sr:listener> *
</t2sr:listeners>
```

Method: POST
Consumes: application/xml, application/json
Produces: N/A
Response codes: 201 Created
Notes: Identity of created listener provided through *Location* header in response.

Add a new event listener to the named workflow run of the given type and under the conditions imposed by the contents of the configuration document (the body of the element). Note that the configuration cannot be changed after creation.

```
<t2sr:listenerDefinition t2sr:type="xsd:string">
    xsd:string
</t2sr:listenerDefinition>
```

Resource: /runs/{id}/listeners/{name}

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Get the description of a particular listener attached to a workflow run.

```
<t2sr:listener xlink:href="xsd:anyURI"
        t2sr:name="xsd:string"
        t2sr:type="xsd:string"
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string">
    <t2sr:configuration xlink:href="xsd:anyURI" />
    <t2sr:properties>
        <t2sr:property xlink:href="xsd:anyURI"
                t2sr:name="xsd:string" /> *
    </t2sr:properties>
</t2sr:listener>
```

Resource: /runs/{id}/listeners/{name}/configuration

Method: GET
Consumes: N/A
Produces: text/plain
Response codes: 200 OK

Get the configuration document for the given event listener that is attached to a workflow run.

Resource: /runs/{id}/listeners/{name}/properties

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Get the list of properties supported by a given event listener attached to a workflow run.

```
<t2sr:properties>
    <t2sr:property xlink:href="xsd:anyURI"
            t2sr:name="xsd:string" /> *
</t2sr:properties>
```

Resource: /runs/{id}/listeners/{name}/properties/{propName}

Method: GET
Consumes: N/A
Produces: text/plain
Response codes: 200 OK

Get the value of the particular property of an event listener attached to a workflow run.

Method: PUT
Consumes: text/plain
Produces: text/plain
Response codes: 200 OK

Set the value of the particular property of an event listener attached to a workflow run.

Resource: /runs/{id}/security

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Gives a description of the security information supported by the workflow run.

```
<t2sr:securityDescriptor
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string">
    <t2sr:owner> xsd:string </t2sr:owner>
    <t2sr:permissions xlink:href="xsd:anyURI" />
    <t2sr:credentials xlink:href="xsd:anyURI">
        <t2sr:credential> ... </t2sr:credential> *
    </t2sr:credentials>
    <t2sr:trusts xlink:href="xsd:anyURI">
        <t2sr:trust> ... </t2sr> *
    </t2sr:trusts>
</t2sr:securityDescriptor>
```

Resource: /runs/{id}/security/credentials

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Gives a list of credentials supplied to this workflow run.

```
<t2sr:credentials
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string">
    <t2s:userpass xlink:href="xsd:anyURI">
        ...
    </t2s:userpass> *
    <t2s:keypair xlink:href="xsd:anyURI">
        ...
    </t2s:keypair> *
</t2sr:credentials>
```

For more description of the contents of the *userpass* and *keypair* elements, see below.

Method: POST
Consumes: application/xml, application/json
Produces: N/A
Response codes: 201 Created
Notes: Identity of created credential is in *Location* header of response.

Creates a new credential. Multiple types supported. Note that none of these should have their *xlink:href* attributes set when they are POSTed; those will be supplied by the service. Take particular care with the *serviceURI* elements, which must define the URI as expected by the Taverna Credential Manager.

Password credential:

```
<t2s:userpass>
    <ts2:serviceURI> xsd:anyURI </t2s:serviceURI>
    <t2s:username> xsd:string </t2s:username>
    <t2s:password> xsd:string </t2s:password>
</t2s:userpass>
```

Key credential (note that one of the *credentialFile* and *credentialBytes* elements should be supplied, but not both):

```
<t2s:keypair>
    <ts2:serviceURI> xsd:anyURI </t2s:serviceURI>
    <t2s:credentialName> xsd:string </t2s:credentialName>
    <t2s:credentialFile> xsd:string </t2s:credentialFile> ?
    <t2s:fileType> xsd:string </t2s:fileType> ?
    <t2s:unlockPassword> xsd:string </t2s:unlockPassword> ?
    <t2s:credentialBytes>
        xsd:base64Binary
    </t2s:credentialBytes> ?
</t2s:keypair>
```

Method: DELETE
Consumes: N/A
Produces: N/A
Response codes: 204 No content

Deletes all credentials.

Resource: /runs/{id}/security/credentials/{credID}

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Describes a particular credential. Will be one of the elements *userpass*, *keypair* and *cagridproxy* as outlined above.

Method: PUT
Consumes: application/xml, application/json
Produces: application/xml, application/json
Response codes: 200 OK

Updates (i.e., replaces) a particular credential. Will be one of the elements *userpass*, *keypair* and *cagridproxy* as outlined above.

Method: DELETE
Consumes: N/A
Produces: application/xml, application/json
Response codes: 204 No content

Deletes a particular credential.

Resource: /runs/{id}/security/owner

Method: GET
Consumes: N/A

Produces: text/plain
Response codes: 200 OK

Gives the identity of who owns the workflow run.

Resource: /runs/{id}/security/permissions

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Gives a list of all non-default permissions associated with the enclosing workflow run. By default, nobody has any access at all except for the owner of the run.

```
<t2sr:permissionsDescriptor
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string">
    <t2sr:permission xlink:href="xsd:anyURI">
        <t2sr:userName> xsd:string </t2sr:userName>
        <t2sr:permission>
            none/read/update/destroy
        </t2sr:permission>
    </t2sr:permission> *
</t2sr:permissionsDescriptor>
```

Method: POST
Consumes: application/xml, application/json
Produces: N/A
Response codes: 201 Created
Notes: Identity of created permission is in *Location* header of response.

Creates a new assignment of permissions to a particular user.

```
<t2sr:permissionUpdate>
    <t2sr:userName> xsd:string </t2sr:userName>
    <t2sr:permission>
        none/read/update/destroy
    </t2sr:permission>
</t2sr:permissionUpdate>
```

Resource: /runs/{id}/security/permissions/{user}

Method: GET
Consumes: N/A
Produces: text/plain (one of: *none*, *read*, *update*, *destroy*)
Response codes: 200 OK

Describes the permission granted to a particular user.

Method: PUT
Consumes: text/plain (one of: *none*, *read*, *update*, *destroy*)
Produces: text/plain (one of: *none*, *read*, *update*, *destroy*)
Response codes: 200 OK

Updates the permissions granted to a particular user.

Method: DELETE
Consumes: N/A
Produces: N/A
Response codes: 204 No content

Deletes (by resetting to default, i.e., *none*) the permissions associated with a particular user.

Resource: /runs/{id}/security/trusts

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Gives a list of trusted identities supplied to this workflow run.

```
<t2sr:trustedIdentities
        t2s:serverVersion="xsd:string"
        t2s:serverRevision="xsd:string"
        t2s:serverBuildTimestamp="xsd:string">
    <t2sr:trust xlink:href="xsd:anyURI">
        <t2s:certificateFile>
            xsd:string
        </t2s:certificateFile> ?
        <t2s:fileType> xsd:string </t2s:fileType> ?
        <t2s:certificateBytes>
            xsd:base64Binary
        </t2s:certificateBytes> ?
    </t2sr:trust> *
</t2sr:trustedIdentities>
```

Method: POST
Consumes: application/xml, application/json
Produces: N/A
Response codes: 201 Created

Adds a new trusted identity. The *xlink:href* attribute of the *trust* element will be ignored if supplied, and one of *certificateFile* and *certificateBytes* should be supplied. The *fileType* can normally be omitted, as it is assumed to be X.509 by default (the only type seen in practice).

```
<t2sr:trust>
    <t2s:certificateFile> xsd:string </t2s:certificateFile> ?
    <t2s:fileType> xsd:string </t2s:fileType> ?
    <t2s:certificateBytes>
        xsd:base64Binary
    </t2s:certificateBytes> ?
</t2sr:trust>
```

Method: DELETE
Consumes: N/A
Produces: N/A
Response codes: 204 No content

Deletes all trusted identities.

Resource: /runs/{id}/security/trusts/{trustID}

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK

Describes a particular trusted identity.

```
<t2sr:trust xlink:href="xsd:anyURI">
    <t2s:certificateFile> xsd:string </t2s:certificateFile> ?
    <t2s:fileType> xsd:string </t2s:fileType> ?
    <t2s:certificateBytes>
        xsd:base64Binary
    </t2s:certificateBytes> ?
</t2sr:trust>
```

Method: PUT
Consumes: application/xml, application/json
Produces: application/xml, application/json
Response codes: 200 OK

Updates (i.e., replaces) a particular trusted identity. The *xlink:href* attribute will be ignored if supplied. The *fileType* can normally be omitted, as it is assumed to be X.509 by default (the only type seen in practice).

```
<t2sr:trust>
    <t2s:certificateFile> xsd:string </t2s:certificateFile> ?
    <t2s:fileType> xsd:string </t2s:fileType> ?
    <t2s:certificateBytes>
        xsd:base64Binary
    </t2s:certificateBytes> ?
</t2sr:trust>
```

Method: DELETE
Consumes: N/A
Produces: N/A
Response codes: 204 No content

Deletes a particular trusted identity.

Resource: /runs/{id}/wd
Resource: /runs/{id}/wd/{path…}

Note that everything following the */wd* is the path beneath the working directory of the workflow run; this is mapped onto the filesystem. An empty path is the same as talking about the working directory itself.

Be aware! Much of the selection between operations is done on the basis of the negotiated content type *and* the nature of the entity to which the path name matches.

Method: GET
Consumes: N/A
Produces: application/xml, application/json
Response codes: 200 OK
Notes: This only applies to directories.

Gives a description of the working directory or a named directory in or beneath the working directory of the workflow run. The contents of the *dir* and *file* elements are the equivalent pathname (for use in the SOAP interface or to be appended to *.../wd* to generate a URI).

```
<t2sr:directoryContents>
    <t2s:dir xlink:href="xsd:anyURI" t2s:name="xsd:string">
        xsd:string
    </t2s:dir> *
    <t2s:file xlink:href="xsd:anyURI" t2s:name="xsd:string">
        xsd:string
    </t2s:file> *
</t2sr:directoryContents>
```

Method: GET
Consumes: N/A
Produces: application/zip
Response codes: 200 OK
Notes: This only applies to directories.

Retrieves the contents of the given directory (including all its subdirectories) as a ZIP file.

Method: GET
Consumes: N/A
Produces: application/octet-stream, */*
Response codes: 200 OK
Notes: This only applies to files. Supports the *Range* header applied to bytes.

Retrieves the contents of a file. The actual content type retrieved will be that which is auto-detected, and the bytes delivered will be those that exist on the underlying disk file.

Method: PUT
Consumes: application/octet-stream
Produces: N/A
Response codes: 200 OK
Notes: This only applies to files.

Creates a file or replaces the contents of a file with the given bytes.

Method: POST
Consumes: application/xml, application/json
Produces: N/A
Response codes: 201 Created
Notes: This only applies to directories. The *Location* header says what was made.

Creates a directory in the filesystem beneath the working directory of the workflow run, or creates or updates a file's contents, where that file is in or below the working directory of a workflow run. The location that this is POSTed to determines what the parent directory of the created entity is, and the *name* attribute determines its name. The operation done depends on the element passed, which should be one of these:

```
<t2sr:mkdir t2sr:name="xsd:string" />

<t2sr:upload t2sr:name="xsd:string">
```

```
    xsd:base64Binary
</t2sr:upload>
```

Note that the *upload* operation is deprecated; directly PUTting the data is preferred, as that has no size restrictions.

Method: DELETE
Consumes: N/A
Produces: N/A
Response codes: 204 No content

Deletes a file or directory that is in or below the working directory of a workflow run. The working directory itself cannot be deleted (other than by destroying the whole run).

API of the SOAP Interface

Taverna 2 Server supports a SOAP interface to the majority of its user-facing functionality. The operations that it supports are divided into a few groups:

- Global Settings
- Basic Workflow Operations
- Input and Output Control
- File Operations
- Event Listeners
- Security Configuration

The connection itself is done (under recommended deployment patterns) by HTTPS, and is authenticated via Basic HTTP username and password at the connection level. All information reported is only necessarily true for a particular user; no guarantee is made that it will be the same for any other user.

Note that the information below is just a summary. The WSDL document for the server should be consulted for the full definition of the messages used with each operation.

Global Settings

These operations describe things that are across all the workflow runs owned by the connecting user on the server.

getEnabledNotificationFabrics

> This obtains the names of the protocols supported for registration for active notification. Note that the Atom feed support is always enabled as it is built into the service itself.

getMaxSimultaneousRuns

> This obtains the maximum number of runs that may be executed at once by the current user; note that this limit might not be reachable by any one user if it is due to a global limit on the number of runs and other users have several runs of their own.

getPermittedWorkflows

> Get a list of workflows that are permitted to be instantiated; if the list is empty, there is no restriction on what workflows may have runs created of them.

getPermittedListenerTypes

> Get a list of types of listeners that may be explicitly attached to a workflow run.

listRuns

> Get a list of all workflow runs on the server that the user has access to. (Workflows that they do not have permission to access even for reading will be not returned by this operation.)

Basic Workflow Operations

An ID string identifies every workflow run. All operations on a workflow run take the ID as one their arguments. (Implementation note: This ID is a UUID.)

submitWorkflow

> Submit a workflow to create a run, returning the ID of the run. Newly submitted workflows start in the *Initializing* state so that you can upload all the required support files before starting the run.

destroyRun

> Destroy the given workflow run. This kills the workflow execution if the run was in the *Operating* state removes all files associated with a run.

getRunExpiry

> Get the time that a workflow run will become eligible for automated destruction. The default lifespan of a workflow run is 24 hours.

setRunExpiry

> Set the time that a workflow run will become eligible for automated destruction.

getRunCreationTime

> Get the time that a workflow run was created (by the *submitWorkflow* operation).

getRunStartTime

> Get the time that a workflow run started executing (i.e., transitioned to the *Operating* state) or null if that has not yet occurred.

getRunFinishTime

> Get the time that a workflow run stopped executing (i.e., transitioned to the *Finished* state) or null if that has not yet occurred.

getRunWorkflow

> Get the workflow document that was used to create a workflow run.

getRunStatus

> Get the current state of a workflow run. In the current implementation, this is one of *Initializing*, *Operating* and *Finished*. (Technically, there's also a *Stopped* state but no run implementation currently supports it, and there's conceptually a *Destroyed* state too, but the service cannot say so as it will instead give a fault stating that the run does not exist.)

setRunStatus

> Set the current state of a workflow run, which is necessary to start it *Operating*. The execution can be finished early by manually moving it to *Finished*, though the run will automatically progress to that state once it terminates naturally. It's always legal to set a run to its current state, and it's always legal to set the state to *Finished*.

Input and Output Control

getRunInputDescriptor

Returns a description of what inputs are expected by a particular workflow.

getRunInputs

Returns a list of what inputs have been configured on a particular workflow, including what file they are to be taken from or what value they are to use.

setRunInputPortFile

Configure an input to take its value from a file in/beneath the job's working directory.

setRunInputPortValue

Configure an input to take its value directly from the supplied string. (Implementation note: Not all values work well when provided this way due to a known issue in the Apache command line library.)

setRunInputBaclavaFile

Configure a run to take all its inputs from a Baclava file. The Baclava file should be uploaded to the run's working directory prior to the state being set to *Operating*.

getRunOutputDescription

Get a description of what outputs have been provided.

setRunOutputBaclavaFile

Arrange for the run outputs to be written as a Baclava file. If this is not called, outputs will be written into files in the *out* subdirectory of the workflow run's working directory.

getRunOutputBaclavaFile

Get the name of the Baclava file that will have the run outputs written to it.

File Operations

Every workflow run has a working directory that is private to itself. That working directory will be the current directory when the workflow run is executing.

getRunDirectoryContents

List the contents of a directory. The workflow run's working directory is denoted by the empty filename, and only that directory or its subdirectories may be listed.

destroyRunDirectoryEntry

Delete a subdirectory or file.

getRunDirectoryAsZip

Given a directory, return that directory plus all its contents (files, subdirectories) as a ZIP file.

makeRunDirectory

Create a subdirectory of a directory. Note, you should not create the out subdirectory; that will be created by the workflow engine.

getRunFileContents

Get the contents of a file, as XML-wrapped base-64 encoded data. (Implementation note: Consider fetching large files by the REST interface, which can handle much more data by virtue of using data streaming, or via the MTOM-enabled operation.)

getRunFileContentsMTOM

Get the contents of a file. (MTOM-enabled.)

getRunFileType

Get an estimate of the content type of a file.

getRunFileLength

Get the length of the contents of a file.

makeRunFile

Create an empty file.

setRunFileContents

Set the contents of an existing file from XML-wrapped base-64 encoded data. (Implementation note: Consider uploading large files by the REST interface, which can handle much more data by virtue of using data streaming, or via the MTOM-enabled operation.)

setRunFileContentsMTOM

Set the contents of an existing file. (MTOM-enabled.)

Event Listeners

getRunListeners

Get a list of listeners attached to a particular run.

addRunListener

Attach a new listener to a particular run. The listener must be of a recognised type.

getRunListenerConfiguration

Get the configuration document of a particular listener. The configuration document can only be read, not written.

getRunListenerProperties

Get the list of properties supported by a particular listener.

getRunListenerProperty
setRunListenerProperty

Get and set the values of individual properites; properties are always strings.

There is one standard listener, *io*, which is attached by default. This listener has an empty configuration document, and provides access to a number of properties. The properties are:

stdout

The standard output stream from the workflow executor process.

stderr

The standard error stream from the workflow executor process.

exitcode

The exit code of the workflow executor process. Empty if not yet exited.

notificationAddress

The URI to push termination notifications to. If empty, no notifications are pushed (but they are always made available by the Atom stream).

usageRecord

If non-empty, a UR1.0-format usage record describing resources consumed during the execution of the workflow.

Per-Run Security Configuration

Note that all of the operations below are restricted to the owner of the run except for discovering the identity of the owner of the run.

getRunOwner

Get the identity of the owner of the run. Note that the owner always has full permission to modify and read the run.

listRunPermissions

List the non-deny permissions granted by the owner.

setRunPermission

Grant a particular permission to a user.

getRunCredentials

List the credentials given to a run to use when contacting other services.

setRunCredential

Give a credential to a run to use when contacting other services.

deleteRunCredential

Stop a run from using a particular credential when contacting other services.

getRunCertificates

List the server certificates that will be trusted when contacting other services.

setRunCertificates

Add to/update the server certificates that will be trusted when contacting other services.

deleteRunCertificates

Remove from the server certificates that will be trusted when contacting other services.