

Slurp Data Services Platform

Table of Contents

Overview	2
The Service Designer	3
Creating a Service	3
Defining Service Inputs and Outputs	5
Adding Components to a Service	6
Deleting a Service	9
Testing a Service	9
Field Mapping Editor	13
Debugging a Service	15
Service Security Model	17
Menu Options	19
Programming Pallet	22
Operations Pallet	24
Running Slurp as a Stand-alone Data Service	28
Windows (Service)	28
Windows (Command Line)	29
Unix (Daemon)	29
Unix (Command Line)	29
Writing and Using a Plugin	30
Server Clustering	31
Frequently Asked Questions	32

Overview

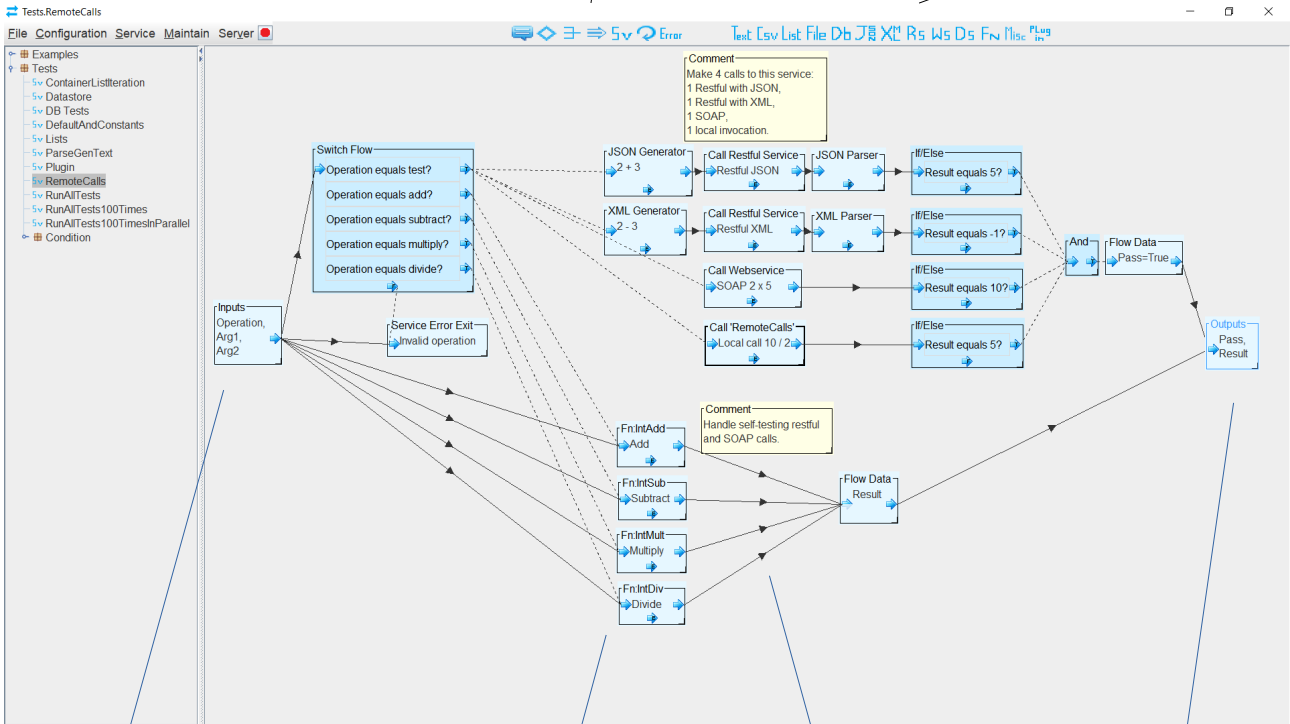
- A one-stop, code-free solution to quickly develop and deploy extremely fast/agile data services for Windows/Unix/Cloud.
- Operates as an in-memory ETL (Extract Transform Load) platform but is also able to stream data between selected components (easier than Azure Data Factory).
- Uses it's own micro-services framework to ensure maximum performance. If run on Java 21+, it makes use of [Virtual Threading](#) for outstanding performance.
- Any service can be configured to be available to an external client, which can access it via Restful or SOAP calls, using basic or token-based authentication.
- Can be deployed as a Windows service, a Unix daemon or by WAR file to a separate Java web server.
- Slurp servers are cluster-ready.
- Has built-in web and OAuth2 servers making it a stand-alone solution.
- Create/test/deploy in mixed Windows/Unix/Cloud environments.
- Requires Java JRE 1.8 or greater. If your Java installation is not recorded in Windows registry, or to choose an alternative JRE, then set the JAVA_HOME environment variable to where it is installed.
- Is a single 13MB binary (slurp.exe) which is used for:
 - a) [The Service Designer](#). The user interface to create/update services, and optionally by single click to create a Java web server deployment of the services (WAR file). It will run as a native application under Windows, and for other platforms run it with 'java -jar slurp.exe'.
 - b) To run Slurp as a [Windows service or Unix daemon](#).
 - c) The library to add to the build classpath when [creating a plugin](#) for Slurp (it is also JAR format).
- The services configuration maintained by a) or read by b) is contained in the file *slurp.xml*, which is in the same directory from which *slurp.exe* is run.
- Other, preference-type properties are kept in the file *slurp.properties*. It will be generated when you edit a service or change any File>Preferences in the Service Designer.
- There are existing plugins for:
 - a) Secure file transfer (SFTP) and secure remote shell execution (SSH).
 - b) LDAP access control checking.
 - c) Reading/writing to Windows shares (SMB/CIFS).
 - d) Reading/writing Microsoft documents (only Excel so far but more by request).

The Service Designer

Menu Options: Create, configure, save and deploy Services

Programming Pallet: Apply conditions to the data flow and other programming components.

Operations Pallet: Parse, generate and filter data, call web services and access data stores and plugins



Inputs component: Specify service inputs here

Outputs component: Specify service outputs here

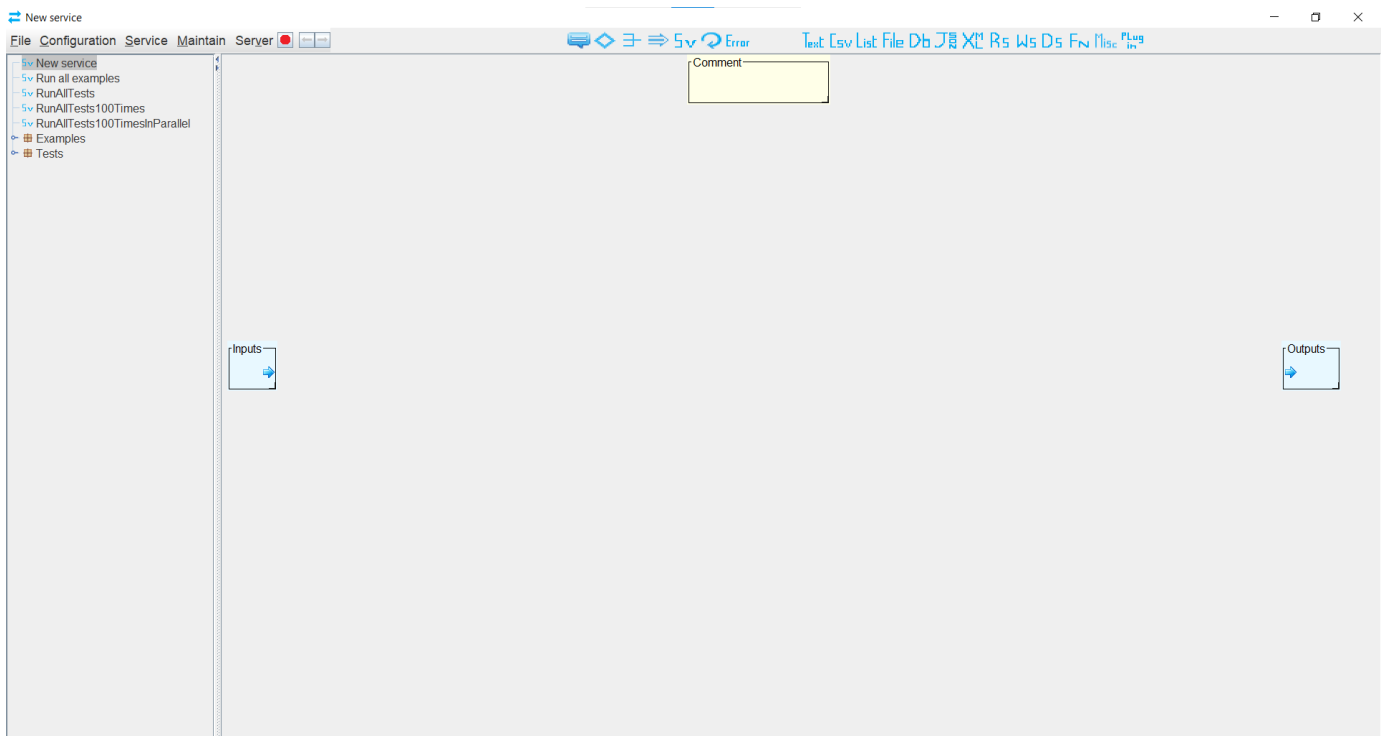
Components: Specify bespoke functionality

Data flows: Connect fields between components

Creating a Service


- Create a service using menu Service>Create (or right-click on the canvas). The following dialogue is displayed:

- Assign a 'Package' to the service so that related services can be grouped together for the convenience of, eg. importing/exporting them. Packages do not exist on their own right, but only if specified by one or more services. The 'Timeout' can be set such that the service will be aborted if its execution takes longer than the specified time. 'Required Permissions' are the permissions required to execute the service, which will be derived from an authenticated user, see [Security Model](#). Automatic (in addition to the normal on-demand) invocation of the service can be specified with a Cron expression and/or when the Slurp runtime is started and/or stopped. 'DB Transactions' specifies how to handle database transactions (if any) for this service and for other Slurp services called from it. See [Miscellaneous Operations](#) for more details. 'Execute in Background' will cause the service to run in the background so that the caller does not need to wait for it to complete. A blank service will look like this:



- The service contains 3 components by default: for input, output and a service comment.
- No service inputs or outputs have been specified so far. If this service is run it will not expect any inputs and will return no outputs. It could nonetheless do something useful in between.
- The service can be tested in the GUI by right-click 'Test Service' (or control-t). Alternatively, it can be tested from a web client, eg. browser, using the inbuilt Jetty web server (see [Testing a Service](#)).
- To switch between different services, either:
 - Double-click on a the service in the navigation panel.
 - Use mouse forward and back buttons (if you have them) to scroll through your service editing history, or the 'Next' and 'Previous' buttons in the menu.
 - Right-click the canvas and select 'Edit Service', then select a service.

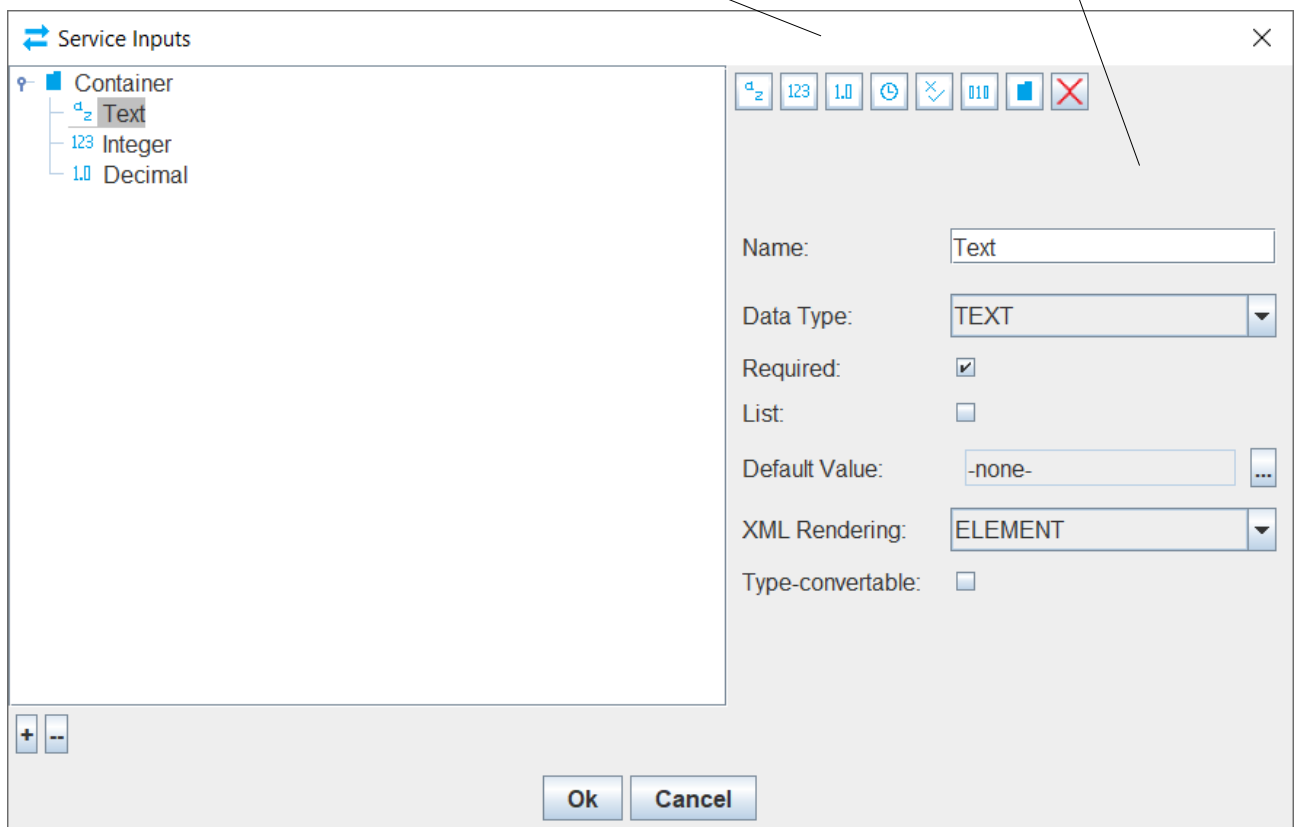
Defining Service Inputs and Outputs

- Double-click the flow icon  in either the input or output component to specify input or output fields. This will invoke the Container Editor on the selected interface, eg:

The structure and content of the interface

Buttons to add and remove fields from the interface

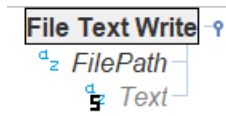
Adjust field name and other field properties



- The basic field types are: text, integer, decimal, date/time, boolean, binary and container. For a singular required field these are represented by one of the following icons respectively:



- The date/time type represents a local date/time as would be represented by a calendar date combined with time. More description in [FAQs](#).
- There are variants for each field type, depending on whether it's a list of the type and if its value is mandatory, eg. for text type:
 - List of text
 - Non-mandatory text
 - Non-mandatory list of text
- If a field has had a constant value applied to it (right-click, 'Set Constant') then it will have a yellow background and the constant value can be displayed by hovering the mouse over it, or the constant changed or removed by right-click.
- Streamable fields have a black 's' marked in the icon, eg:



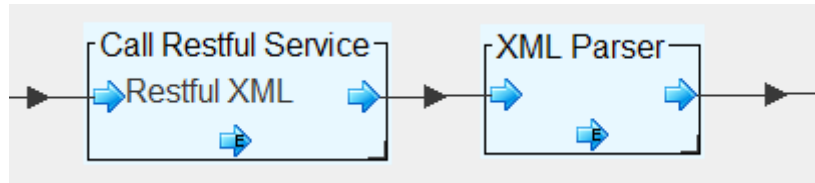
If a source streamable field/container is mapped to a destination streamable field/container and they have no other mappings then data will be streamed between them, as opposed to all being held in memory. This allows for unlimited amounts of data to be transferred between the components. A data flow that is set to stream data between components is depicted with a thick black data flow line. More details about streaming data flows can be found [here](#).

- You can also set the service input or output fields using an XSD (XML Schema Document) by double-clicking on the input or output component itself.

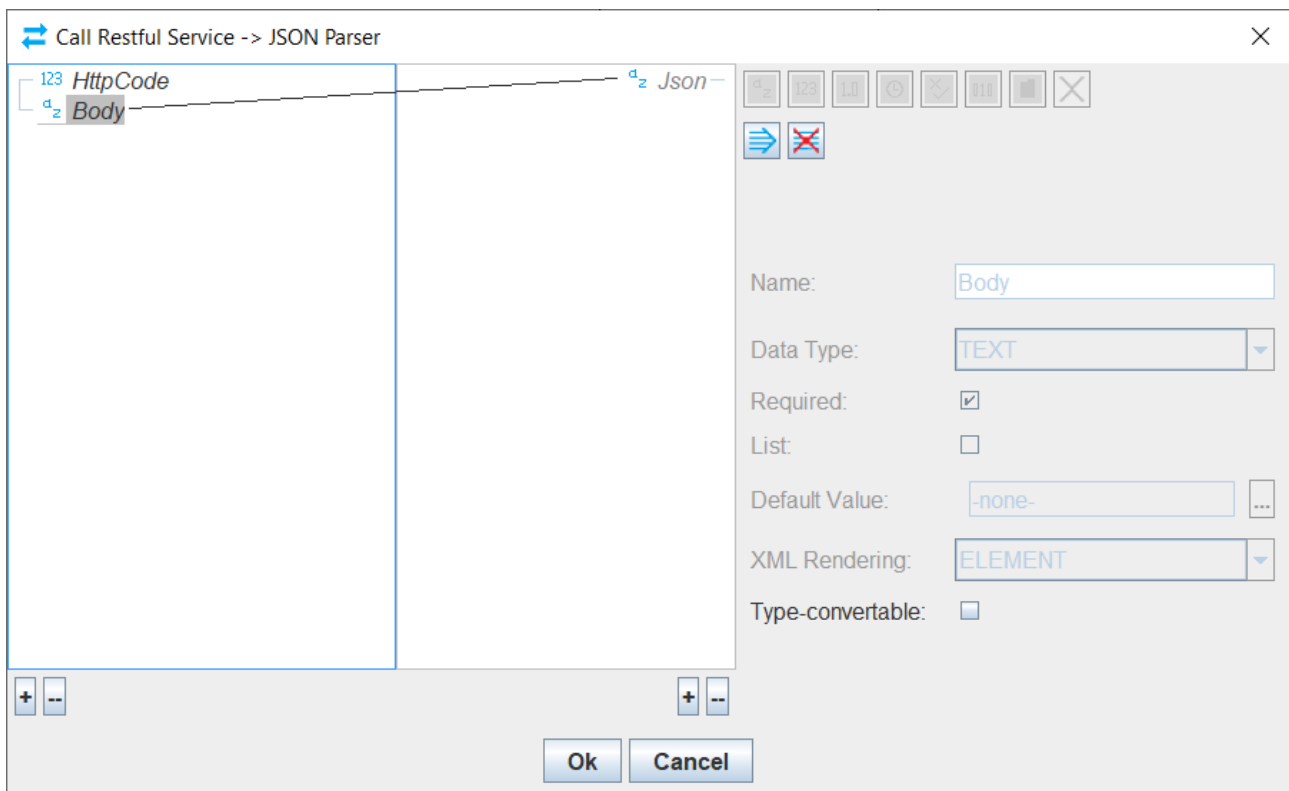
Adding Components to a Service

- This is done by dragging components from the [Programming Pallet](#) or [Operations Pallet](#) and dropping them on the canvas.
- At the point of drop, most components display a dialogue so that a variant of the component can be chosen and configured. This process will set the component's input and output fields, and usually the default value of input fields (to save you from having to set them using a data flow or constant).
- The arrow on the left is for the component inputs, the arrow on the right is for the component outputs and the arrow at the bottom can be used to handle errors incurred by the component.
- Initially, the component's titles and borders are greyed out. This is because it doesn't yet have any input flows to activate it. It is effectively commented out code until at least one input or control flow is passed to it (see [Programming Pallet](#) for a description of component activation).




- Once the component is on the canvas, it's conventional to specify where its inputs are coming from (if it has any). This is done by dragging a line (creating a flow) from the output of a previous component to the input of the current component, eg: to parse the output of a Restful service call:



- Components and flow lines can be removed by selecting them with the mouse and then using the delete key. Most components panels can be written into to document their intended purpose.
- Double-click a data flow line to specify the data mappings between the output fields of the previous operation to the input fields of the next operation (at runtime, this will signify that the contents of the output fields are copied to the input fields). This invokes the [Field Mapping Editor](#), as follows:



- Create data mappings by dragging a line from a source field on the left to a destination field on the right. In-line transformations on the data can be specified by right-click and selecting 'Add Function' to wire-in a function call (the same functions can be dropped onto the service canvas by using a [Call Function Fn](#)). Illogical mappings, eg. decimal field mapped to binary field or trying to lose the context of a container list element cannot be made or will be highlighted as an error.

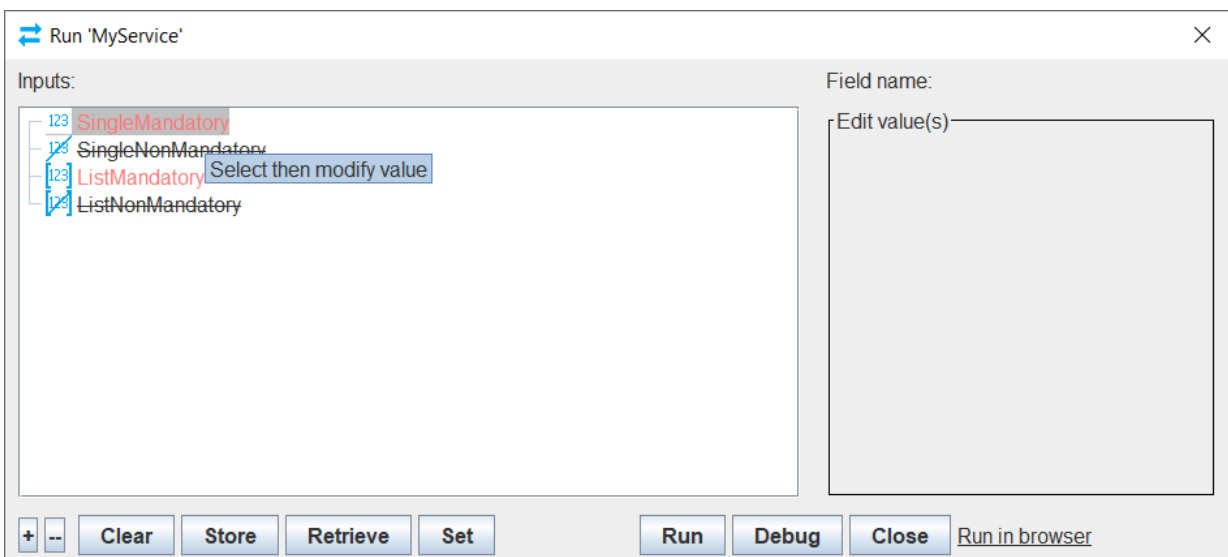
- Data mapping lines can be removed by clicking with the mouse and then using the delete key. Once happy with the mappings, click Save.
- Now that the inputs are available to the component it may require further configuration (which is often dependent on the inputs). To do this, double-click the component to (re)configure it.
- Editing data flows and configuring components can be done as many times as necessary.
- Constants can be assigned to input fields instead of using field mapping. Right-click a field to get options. Input fields can also be assigned a default value in case the field is not set by a mapping or constant.
- Components can be moved or copy/pasted using multi-selection: component selection can be done by holding down the left mouse button and dragging across component titles and/or holding the Control key and clicking component titles. Then to move the selection drag a selected component title, or right-click for copy/paste options.
- Right-click on a component to get other component options, including:
 - Edit/View the component inputs or outputs (if it has them). This can also be achieved by double-clicking the component input or output icons.
 - Turn container list iteration on/off for the component. A component with this setting expects to operate on the elements of a top-level container list (see [program iteration constructs](#)).
 - Generate XSD or JSON Schema for the component inputs or outputs (if it has them).
 - Log inputs, outputs or both. Requires logging level set to at least 'custom'.
- If the component doesn't have any data inputs it will still require an incoming flow to active it, ie. makes it dependent on the execution of a previous component. This can be either an empty data flow from another component or the control flow output from a [Conditional Flow Control](#) component or error flow from an operation (see description of the [Programming Palette](#)).
- The different styles of flow lines are:
 - Data flow with no field mappings 
 - Data flow with field mappings 
 - Control flow 
- You can right-click the canvas for other options. If a data flow or component shows any red colouration (indicating an issue) then hover the mouse over it to find the reason.

Deleting a Service

- You can either use menu Service>List/Delete to delete services or right-click on services in the navigation bar and select 'Delete Service(s)'.
- As services can be dependent on other services, it can be helpful to use the former method, as having deleted some services you may decide to not save the result, in which case you can just cancel the dialogue. The dialogue informs you of any missing dependencies as you remove services and allows you to inspect the affected services before dismissing the dialogue.
- Whichever method you choose, if your configuration had unsaved changes before service deletion(s) then a snapshot of the configuration is taken prior any deletions. Prior configurations can be reverted to using menu Configuration>Edit Log.

Testing a Service

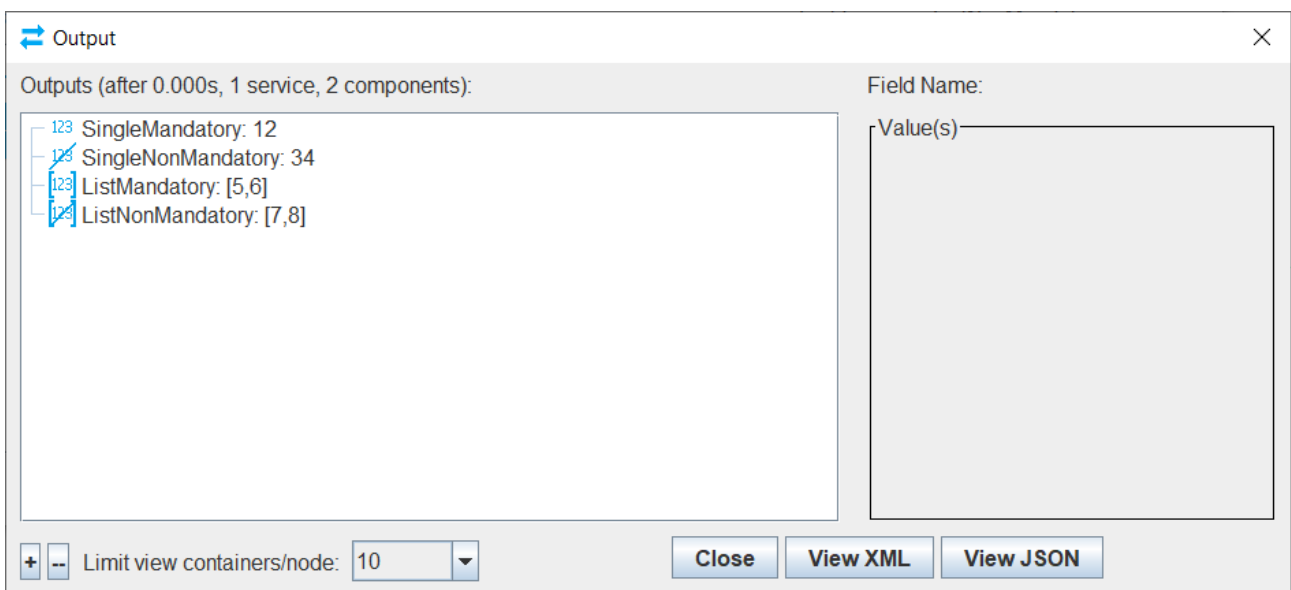
- This can be done easily in either of three ways: a) Select menu Service>Run (or control-r) to provide data in the GUI and see the response in the GUI (or browser), or b) Use the built-in Jetty web server and call the service from an external client (web browser, Restful or SOAP client), or c) Create a deployment WAR file for an external Java web server and test the service there.
- For test method a), you will be prompted to enter test input data:



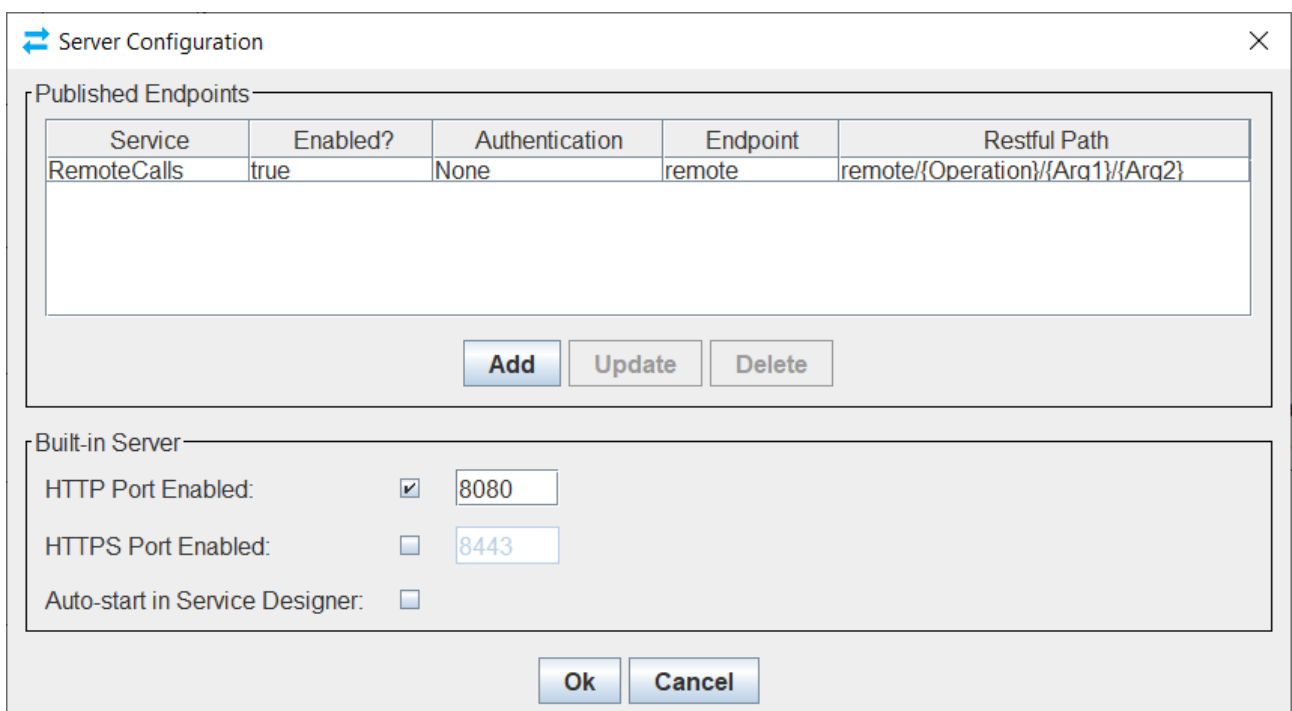
The screenshot shows a dialog box titled "Run 'MyService'". It features a tree view on the left under the heading "Inputs:" with four items: "SingleMandatory" (highlighted in red), "SingleNonMandatory" (with a strike-through), "ListMandatory" (with a blue selection box and a tooltip that says "Select then modify value"), and "ListNonMandatory" (with a strike-through). To the right of the tree is a "Field name:" label and a large text area labeled "Edit value(s)". At the bottom of the dialog are several buttons: "+", "--", "Clear", "Store", "Retrieve", "Set", "Run", "Debug", "Close", and "Run in browser".

- Unset mandatory fields will be marked in red, until set. Unset non-mandatory fields are displayed with strike-through. Select a field and then click in the right-hand box to set its value. If the field is a list then multiple values can be entered. Right-click the field or its value to unset or remove it. A summary of a field's current value is displayed to the right of the field.

- Test data can be created, saved or recovered using the buttons on the left ('Clear', 'Store', 'Retrieve', 'Set').
- To run the service you have 3 options:
 - a) 'Run': will run the service until it completes normally, or a non-handled error occurs, or a service times out (timeout is a service property).
 - b) 'Debug': will enable you to debug your services, using breakpoints, single-stepping, etc (see [Debugging a Service](#)).
 - c) 'Run in browser': will run the service from your browser (only enabled for a service with non-container parameters).
- Click 'Run' to run the service immediately, and display the results:



- For test methods b) and c), the server must be configured by selecting menu Server>Configure. The following dialogue is displayed:



- By default, no services can be called from outside Slurp. To make a service available externally, assign it a service 'Endpoint' and/or 'Restful Path' by using the 'Add' button, and enabling it (only 'Endpoint's are available via SOAP). The choice of authentication methods are: none, basic authentication or access token-based authentication. For more detail, see [Security Model](#).
- For test method b), specify which protocols, HTTP and/or HTTPS, can be used to access the services. For HTTPS, additional details will appear for configuration of the site certificate store. Then start the local server using menu Server>Start Local Server, or click the circular server status icon (red=stopped, green=running).
- So, for example, with 'HTTP Port enabled' set to 8080, service "Service XYZ" assigned endpoint name "xyz" and the local server started, then a JSON encoded response from the service can be invoked from a browser at *http://localhost:8080/xyz.json*.
- For test method c), the 'Built-in Server' details are not used. Having specified the Endpoint and/or Restful Path and optional authentication, use menu Server>Create Deployment to create the deployment WAR file.
- There are 3 ways for a request to pass parameters to an invoked service:
 - a) As URL parameters, eg. *http://localhost:8080/remote.json?Operation=add&Arg1=3&Arg2=5*, or
 - b) As components of the restful path, eg. *http://localhost:8080/remote/add/3/5.json* (where Restful Path for service has been set to *remote/{Operation}/{Arg1}/{Arg2}*), or
 - c) As a JSON object or XML content in the request body (if it starts with '{' it is assumed to be a JSON object, if '<' then XML).

Example JSON request body:

```
{"Operation": "add", "Arg1": 3, "Arg2": 5}
```

Example XML request body (where input fields are defined as 3 attributes):

```
<root Operation="add" Arg1="3" Arg2="5"/>
```

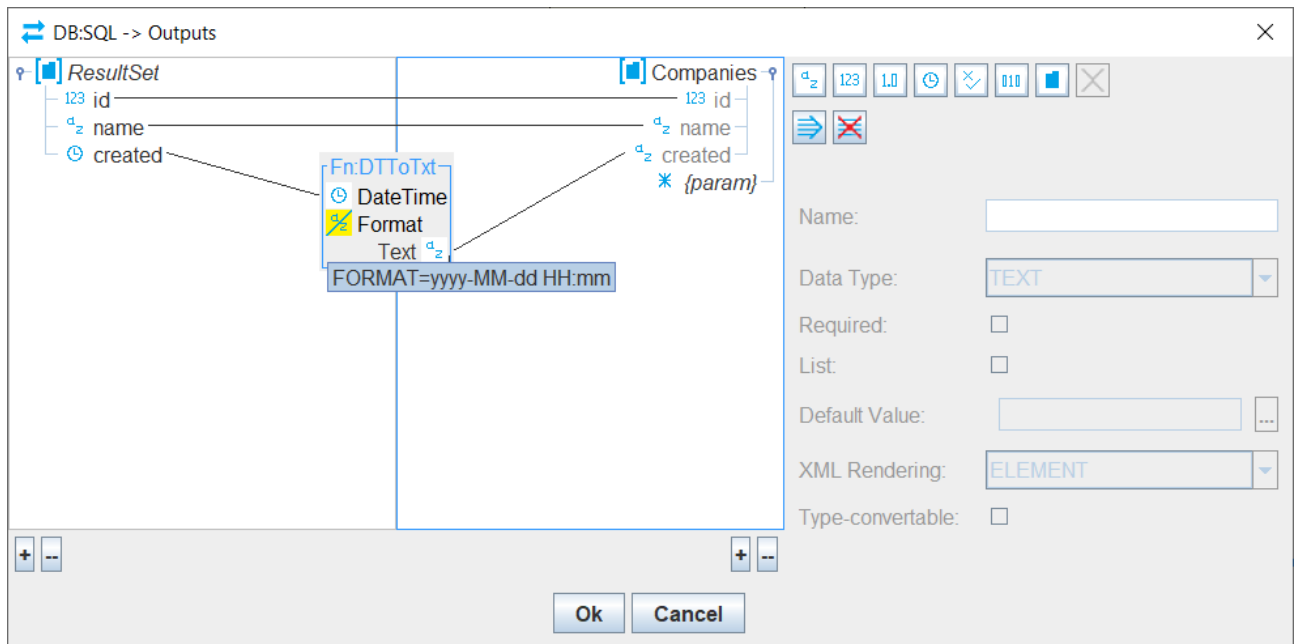
Example XML request body (where input fields are defined as 3 elements):



```
<root><Operation>add</Operation><Arg1>3</Arg1><Arg2>5</Arg2></root>
```

- The format of the response from the service is specified in the extension placed after the endpoint name in the request URL: ".json" for JSON, ".xml" for XML, or ".txt" for any other content (for this format, the output container must have a single text field; use the function `Fn setResponseHeader` to set the Content-Type header of the HTTP response according to the type of text).
- The full set of endpoint extensions are:
 - .json - execute the service and return the results as JSON.
The returned content type will be 'application/json'.


- .xml - execute the service and return the results as XML.
The returned content type will be 'application/xml'.
- .txt – execute the service and return the results of the single output text field.
The returned content type will be 'text/plain' unless function `FnsetResponseHeader` has been called to set 'Content-Type' otherwise.
- .jsoni - return a JSON Schema of the request body (no service execution).
- .jsono - return a JSON Schema of the response body (no service execution).
- .xsdi - return an XSD (XML Schema Definition) for a request body (no service execution).
- .xsdo - return an XSD (XML Schema Definition) for a response body (no service execution).
- Alternative, use your favourite SOAP client to test the service. The WSDL for available service 'Endpoints' (not 'Restful Path's) can be obtained while the local server is running from, eg. *<http://localhost:8080?wsdl>*.

Field Mapping Editor



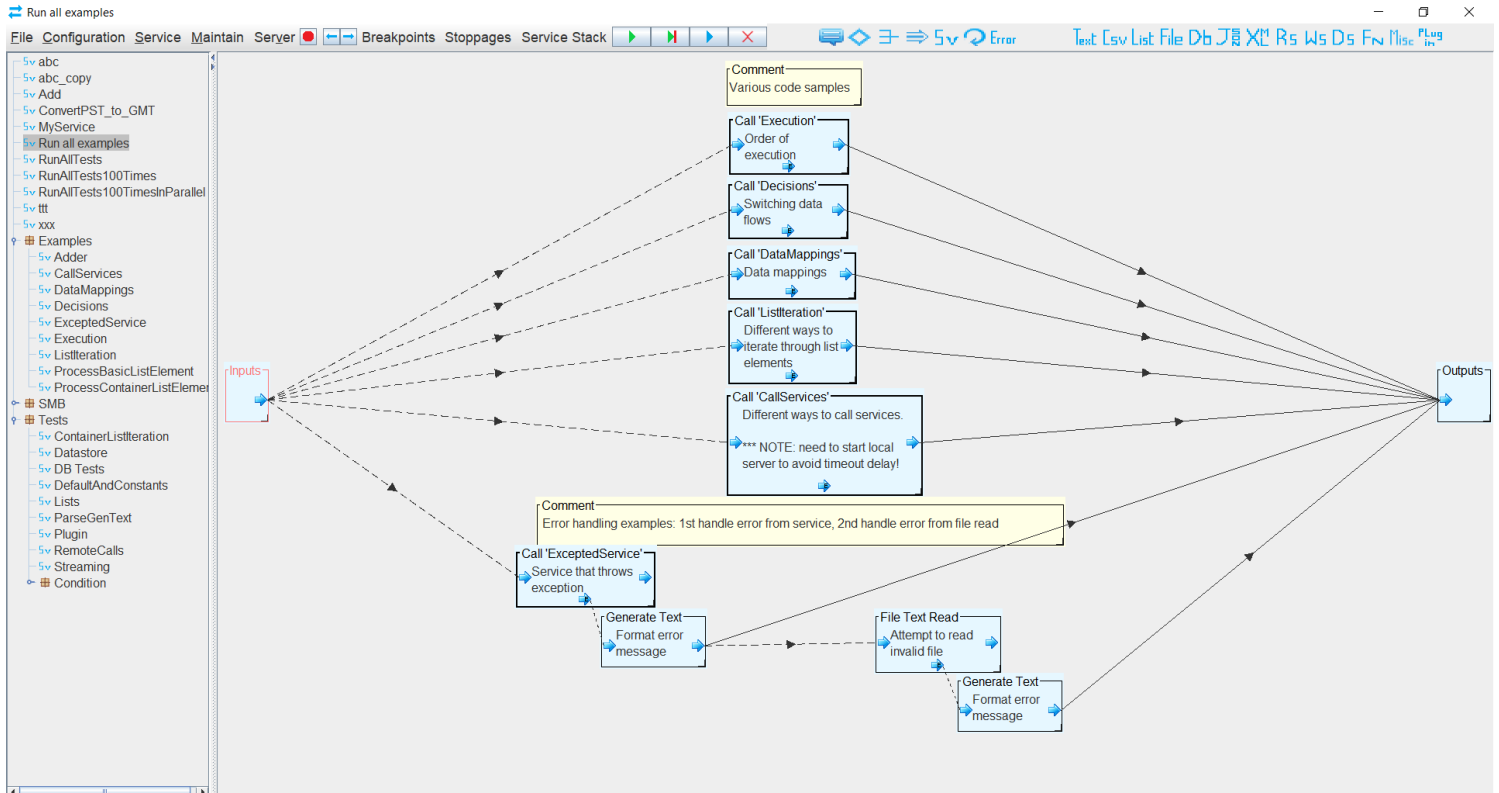
- The Field Mapping Editor is invoked by double-clicking on a data flow that connects the output container of a previous operation to the input container of the next operation.
- It can be used to specify the field mappings between the containers (ie. how the contents of the fields will be copied at runtime) as well as the properties of the fields in the containers.
- Fields can be created and removed using the top-right toolbar or delete key. To specify a field mapping, drag a source field to a destination field using the mouse. Invalid mappings will not be allowed, or highlighted in red. If the destination container has a pseudo field called '{add}' then dragging a mapping line to it will create a copy of the source field as well as the mappings to it (not for container field in some circumstances). This pseudo field mechanism is used as a prompt to where input (parameter) fields are expected to be created as well as facilitating their creation.
- Click on a field to view its properties, and change them (component-defined fields are generally read-only but their default value can nonetheless be updated).
- Right-click for other functions including copy, paste and for adding an in-line function call.
- A destination field does not have to be mapped to and could, if necessary, be set to a constant value (right-click field, 'Set Constant'), or have a default value set.
- Dragging from  a source container to a destination container will attempt to map fields between  the two containers. This auto-mapping only applies to fields of the

same name (case-insensitive) and type (as might be the due to copy/paste of a field hierarchy). The Auto-map tool performs the same function but at the top level.

- All mappings can be removed with tool 
- Slurp has 7 field types, made up of 6 data (text, integer, decimal, date/time, boolean and binary) plus the container type. It is not possible to map from one type directly to a different type without either using a conversion function [Fn](#) or specifying that the source field permits auto-conversion ie. by default, no automatic type conversions are performed.
- The toolbar elements and field properties may be disabled, depending on field selection and whether the field is read-only (a component-defined field).
- Incorrect mapping lines are red dashed. If a field is in labelled in red then hover the mouse over it to find the reason.
- Normally, the source field and destination field must be of the same data type to be able to connect them together. However, if the source field is marked as 'Type-convertable' then some standard type conversions are permitted, e.g text to integer, but this will cause a runtime error if, eg. text isn't valid integer text.
- If the destination field is a container that allows user-added parameters then the pseudo-destination field '{add}' specifies where additional component parameter fields are to be populated. These are usually data fields (ie. not a container), but for some components a container can also be specified.

Debugging a Service

- If you click 'Debug' in the run panel, the Service Designer will be displayed with 3 extra menus and 4 extra buttons, and service execution will be stopped on the Inputs component (component title and border coloured red). Service time outs are not honoured in debug mode:







- The additional debug menus and buttons are:



- The 'Breakpoints' menu lists breakpoints that have been set on components (by right-clicking them and selecting 'Set Breakpoint'. You also have the option of clearing all breakpoints. Individual breakpoints can be cleared by right-clicking a component and selecting 'Clear Breakpoint'. When a component has a breakpoint set on it, it will be displayed with a thick red border (only visible when in debug mode).
- The 'Stoppages' menu lists service runners where components are waiting to be executed. It may contain different entries for the same service as the same service may be being executed by different service runners at the same time.
- The 'Service Stack' menu lists the service runner stack in the context of the currently select stoppage (if you haven't selected a particular stoppage then it will

default to displaying the service runner stack of the first stoppage found for the current service).

- You can select a service runner from the stack to see the inputs and outputs of executed components in the service call chain.
- The  button runs any single 'awaiting execution' component in the current stoppage service (select a particular stoppage if the service is currently being run by different service runners). The icon will be greyed out if there are none to execute. To run a specific 'awaiting execution' component, right-click and select 'Execute' (or 'Execute Into' if it's a service call and you want to step into the service call).
- The  button will resume executing all 'awaiting execution' components across all services and stop execution at the first component which is set with a breakpoint. If no breakpoint is hit then execution will continue without stopping. The icon will be greyed out if there are no breakpoints set across all services.
- The  button will resume executing all 'awaiting execution' components across all services and continue without stopping (ie. breakpoints will have no effect).
- The  button will terminate the debug session immediately and return you to the run panel.
- Additionally, if you have a service with any 'awaiting execution' components then you can right-click on any component and select 'Execute to Here'. This will resume executing all 'awaiting execution' components in the current service runner and either stop at the selected component, or if it is not hit, execution will stop on the output component (giving you the opportunity to examine what just happened).
- During debugging, right-clicking a component will display component-related debug options, below the divider in the pop-up menu. These include:
 - a) Execute. This option is only available if the component is awaiting execution.
 - b) Execute Into. This option is only available if the component is awaiting execution and is a service invoker.
 - c) Set/Clear Breakpoint. This option is always available.
 - d) Execute to Here. This option is only available if you are in the context of a stopped service.
 - a) View Inputs/Modify Inputs. The View Inputs option is available if the component has already been executed. The View/Modify Inputs option is available if the component is awaiting execution.
 - a) View Outputs. This option is available if the component has already been executed.
- Note: during debugging, you are free to edit any service. However, editing a currently executing service may cause it's execution to hang or complain about a missing value.

Service Security Model

- Slurp uses a permissions-based security model where the permissions granted to a request can be derive from either basic authentication (HTTP Authorization, type 'Basic') or indirectly via an access token (HTTP Authorization, type 'Bearer').
- By default:
 - a) No system permissions, static access tokens, basic authentication or OAuth2 services are pre-defined (menu Maintain>Security).
 - a) The default authentication for service endpoints is 'None' (menu Server>Configure).
 - a) Services cannot be stipulated to have 'Required Permissions', since none are pre-defined (menu Service>Create or when editing service properties).

A service cannot be called from outside of Slurp unless it has been given either an Endpoint and/or Restful path in menu Server>Configure, and which is 'enabled'.

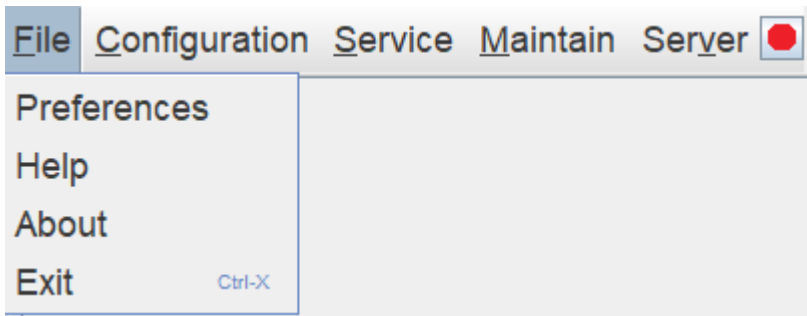
- If basic authentication is required for any service then go to menu Maintain>Security and select the '--- Create new ---' option of the 'Authentication Service' drop-down. This will create a skeletal authentication service, then edit it to suit your needs. For any endpoint which is specified to use 'Basic' authentication, this service will be called to authenticate the value contained in the 'Authorization' (type Basic) request header. The authentication service does not need to supply any permissions to an authenticated user, unless the services they are attempting to call require them.
- If token-based authentication is required for any service then go to menu Maintain>Security and either create a static access token or configure the built-in OAuth2 server.
- To obtain a static token create a new row in the 'Static Access Token' table and then use the 'Copy Value' button to get a copy of the value of the token into the clipboard.
- To configure the built-in OAuth2 server select the '--- Create new ---' option of the 'Token Service' drop-down. This will create a datastore-based token service (together with datastore and housekeeping service). You can update the implementation to suit your needs. Then, for each 3rd party service create a 'Client Registration' entry.
- To obtain an OAuth2 token, 3rd party clients must login at local server path '/token_request'. Successful authentication will redirect the client to the 'Redirection URI' with the token.
- For token-based authentication, a service invoker must use either the static token value or, for OAuth2, the token delivered to the 'Redirection URI' as the value of the 'Authorization' (type Bearer) request header. The access token does not necessarily

need to supply any permissions to the authenticated request, unless the services being called require them.

- Basic and token-based authentication associate an authenticated user name, optional numeric Id and optional permissions to a request. These can be checked by:
 - a) Specifying which permissions are required to execute a particular service by setting the 'Required Permissions' property of the service in menu Service>Properties. Or,
 - b) A service can programmatically test for the user or their associated permissions by calling Function `Fn`, either *RequestUserName*, *RequestUserId* or *HasUserPermission*.
- An unauthenticated request will have a blank user name, 0 for user id and empty list of permissions.
- Slurp has a built-in login throttle in case of repeated authentication failures, to defeat brute-force attacks.

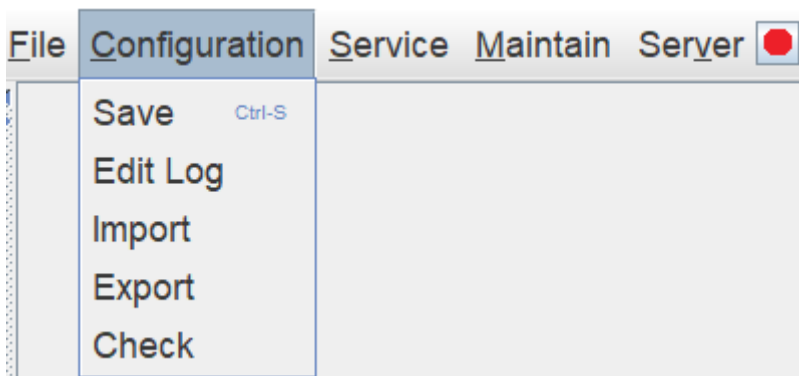
Menu Options

File Menu



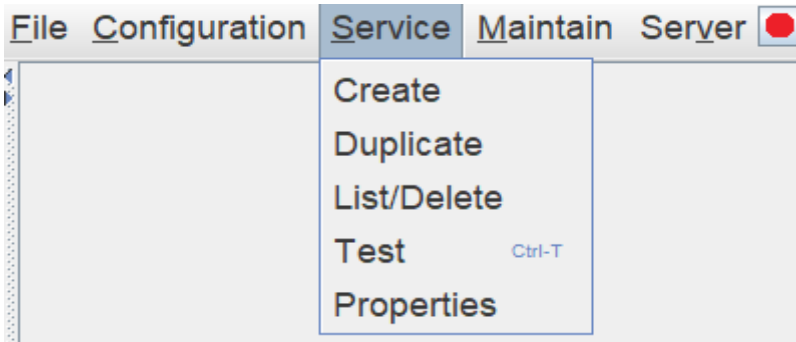
- Preferences. To adjust GUI and runtime preferences.
- About. The Slurp version and included software credits.
- Help. Displays this PDF in your browser.
- Exit. To terminate the GUI (and embedded Jetty server if it is running, indicated by a green spot instead of red).

Configuration Menu



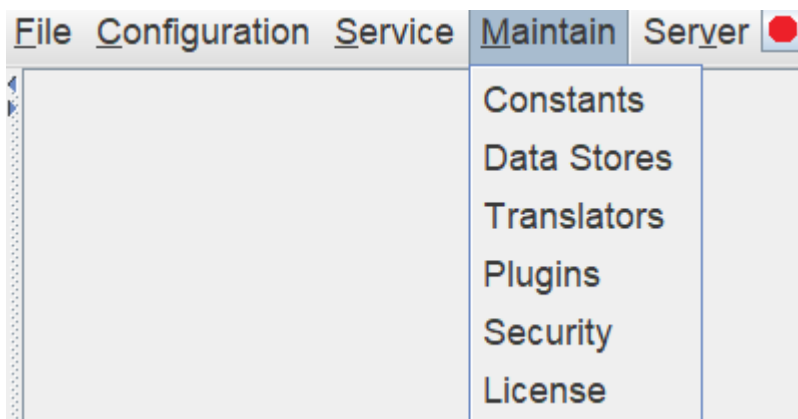
- Save. Writes the current configuration of services to its configuration file (slurp.xml in the current directory).
- Edit Log. Lists snapshots of your edited configuration since the Service Designer was started so that you can switch to a previous instance of your editing.
- Import. Imports service definitions that have been exported from another project into the current project.
- Export. Exports service definitions so that they can be imported into another project.
- Check. Checks for errors/inconsistencies in the current configuration. Most of these checks are performed routinely during editing but this function will also warn you about service invokers that are missing their target services. Generally, missing services are allowed but they will cause a runtime error if you try to invoke them. A warning will be issued when you start the Service Designer.

Service Menu



- Create. Create a new service (also achieved by right-clicking canvas and selecting Create).
- Duplicate. Use a copy of the current service to create a new service.
- List/Delete. View details of existing services and perform housekeeping.
- Test. Test the current service manually from within the GUI.
- Properties. Update the properties of the current service (name, description, package, ad-hoc scheduling). Note: ad-hoc scheduling of a service will only take place when the internal Jetty web server is running (ie. GUI testing or when run as a service/daemon) or a deployment is made to an external web server.
- Delete. Delete the current service.

Maintain Menu

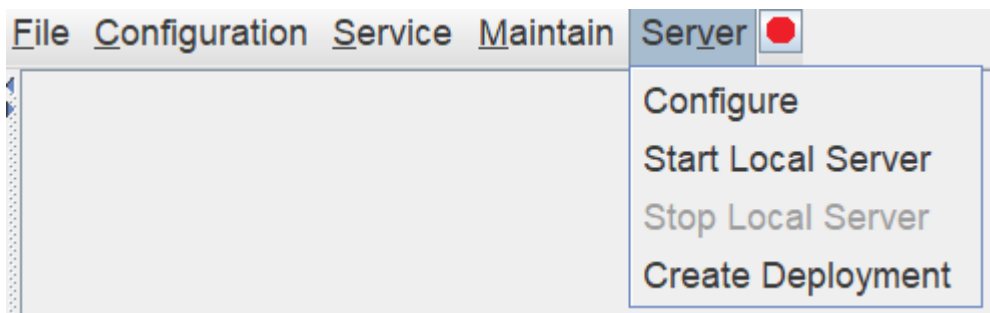


- Constants. Are literal values that are used when configuring components, they can be defined and updated here (or while editing components).
- Data Stores. This is where data stores can be created and updated. The structure of the data can be changed and whether it is persisted to file or not. Persistent data stores can be used to synchronise service activities across different JVMs to facilitate clustering. Operations on data stores can be added to a service by using the [Access Data Store](#) operation **D5**
- Translators. A translator maps values between one simple type and another simple type. For instance, a number to a name, or a short name to a longer equivalent

name. Translators can be defined and updated here. Translators act like functions and can be called from within a data flow (right-click 'Add Function') or on canvas via the [Call Function](#) operation.

- **Plugins.** The plugin architecture allows user-defined Java code to be integrated into a service. Having created a class for a plugin, it can be made known to the Service Designer here. See [Writing and Using a Plugin](#). JDBC drivers must also be added here.
- **Security.** Here you can define system permissions, static access tokens, basic authentication service or OAuth2 configuration, see [Security Model](#).
- **License.** A license is only required to run a production data service. Without a license the built-in Jetty server will stop serving after about 1 hour. But you don't need a license to use Slurp as a non-production service for any number of collaborators, or to contact our [support](#). We're only too happy to help.




Server Menu




- **Configure.** Select which services are available to clients and with what type of authentication (none, basic or token-based), what service (if any) to use for basic authentication, and HTTP and/or HTTPS configuration.
- **Start Local Server.** To start the internal Jetty server (or use the Start/Stop button).
- **Stop Local Server.** To stop the internal Jetty server (or use the Start/Stop button).
- **Create Deployment.** To create a deployment WAR file which can be deployed on any Java web server, eg. Tomcat.
- **Local Server Start/Stop button.** A shortcut to stop and start the local Jetty web server. It's colour will be red if the server is stopped and green if the server is running.







Programming Pallet



- Drag a component and drop it at the desired location on the canvas. Once on the canvas it can be relocated by dragging its title bar or resized by dragging its bottom right corner. A few components use their panel to describe themselves, but most don't, leaving the panel available for documentation purposes. Generally, double-click a component to (re)configure it or right-click it for other options.
- Initially, the component's titles and borders are greyed out. This is because it doesn't yet have any input flows to activate it. It is effectively commented out code until at least one input flow is passed to it.
- A component with a red background means it's configuration is incomplete or invalid. Hover your mouse over the component to obtain the reason.
- Connect components together in a left to right manner by dragging flow lines from components to dependent components.
- There are two types of flow line, which together control the sequence of component execution. A *control flow* line is drawn finely dashed and used for conditional activation of one or more dependent components. On the other hand, a *data flow* line is used to map the output fields of one component to the input fields of a dependent component and is usually drawn as a continuous line [Note: a data flow line with no mapped fields is slightly dashed because it is only providing dependency control].
- A control flow line can originate from the following icons, and at runtime is considered *activated* if the associated event occurred:
 -  Conditional test or And/Or evaluates to 'true'.
 -  Default 'false' result when none of the conditional tests evaluate to 'true'.
 -  If operation error is to be handled by a dependent component and an error occurred (otherwise unhandled errors result in service error exit).

Control flow lines do not require configuration, so double-clicking them is of no use.

- A data flow line on the other hand can only originate from the output of a preceding operation component, from icon  At runtime, the data flow is activated if the preceding operation was activated and completed without error. The field mappings in the data flow can be edited by double-clicking on the data flow line. [Note: a data flow line might originate from a component that does not have an output container and is therefore only providing dependency control].
- So, the rules for a component to be activated are as follows:
 - a) It has at least 1 input control or data flow.

- b) By default, ALL input flows must have been activated. But this can be changed to '1 or more' by right-clicking the component and selecting 'Set Activation by Flows: 1 or more' and the icon will change to  or .
- The And/Or component is the only exception, where only condition a) applies.
 - A component that successfully executes will activate its output flow(s) . However a component that fails will activate its error flow(s) .
 - Hence, sections of programming can effectively be commented out by removing all of its input flows. However, if the result of this is that required fields in uncommented out components are no longer being set then you may have to make other changes to accommodate the commented out code. Commented out flows lines are ignored by the component activation mechanism.
 - A component that incurs an unhandled error (ie. its  not connected to a handler component) will cause service termination with an out-of-band message specifying the error. The format depends on the requested result type (XML, JSON or SOAP fault). To handle component errors (and not have the service terminated prematurely) simply connect one or more control lines from the component's error icon  to activate component(s) that will handle the error.
 - A component that is activated but has any missing 'required' inputs will not actually be executed, so its output will not be set (makes allowance for missing optional elements and is equivalent to a mapping functions not called if one of its 'required' inputs is not set).

Comment

Can be added anywhere on the canvas for additional documentation.

Conditional Flow Control

Used to control execution flow as with If/Then/Else/Switch. Input fields contents can be tested to control execution flow. Multiple test conditions can be configured, which will be evaluated in sequence. Each has a 'true' control output and there is a default 'false' control output from the bottom. The first condition that evaluates to 'true' (activated) completes the evaluations, otherwise 'false' is activated.

And/Or Flow Control

Used to combine activations. Takes data and/or control flows as inputs and outputs a 'true' activation depending on the activation state of its inputs and the selected logic.

Flow Data

Copies its inputs to its outputs, so that data flow can be activated by a control flow, or when an extra data merge step is required.

Invoke Service

Invoke another Slurp service. Once on the canvas, double-click to switch to the service.

Rerun Service

Reruns the current service with new inputs. Allows for repetition (see [program iteration constructs](#)).

Service Error

Exit the current service with a formatted error message and optional HTTP status code (defaults to 500 if not set). If this service is called in a hierarchy of Slurp services which do not handle errors then the error is propagated to the ultimate caller of the initial Slurp service. As with any other unhandled service error, this will result in an out-of-band XML, JSON or SOAP fault message to the service caller. A Slurp service can choose to handle any error (including error by this mechanism) that might be generated in a call to another Slurp service.

Operations Pallet



- As for the Programming Pallet: drag an operation and drop it at the desired location on the canvas.
- Connect operations together in a left to right manner by dragging flow lines from operations to dependent operations.
- Double-click data flow lines to specify the mapping of output fields to input fields.
- Double-click an operation to (re)configure it or right-click it for other options. In most cases, (re)configuration just sets the default value of one or more input fields, which if necessary could be overridden by suitable field mappings or setting a constant value.
- The operation panel can be written into, to describe its intended purpose.
- An operation with a red background means it's configuration isn't complete yet. Hover your mouse over the component to obtain the reason.

Text Parser/Generator

Parse or generate formatted text. There are 3 variants of this operation to choose from:

- Generate text from fields: Map input fields to the operation then double-click to configure, ie. format the desired output text. If you have a need for printf-style formatting then use function 'TxtPrintf' (functions can be called from within a data flow (right-click 'Add Function') or on canvas via [Call Function Fn](#)).
- Generate text from list: Map a single input list (either simple list or container list) and other input fields to the operation then double-click to configure, ie. format output text for the list.
- Parse text: Map a single input text field to the operation then double-click to configure, ie. specify how output text fields are scraped from the input text. You can parse by specifying a template or a regular expression.

CSV Parser/Generator CSV

Parse or generate CSV (comma separated values) text. There are 2 variants of this operation to choose from:

- Generate CSV: Map a container list to the operation to generate the equivalent CSV output for the list.
- Parse CSV: Map a CSV text to the operation and set the output container list fields as expected.

To generate no column titles use an empty list for input 'Headings'. To optionally choose a specific CSV format set the 'Flavour' input to one listed in [CSVFormat.Predefined](#). A null text field will be formatted as for an empty one. If you want to preserve the significance of null text then you can use Fn:ToNullValue and Fn:FromNullValue to replace null with a representation of null (can also be used for XLS if you want to avoid having blank cells).

List Operations List

These are list operations which require a measure of configuration. As opposed to the generic list operations, such as Append Item, which are available via function calls (functions can be called from within a data flow (right-click 'Add Function') or on canvas via [Call Function Fn](#)). There are 8 variants of this operation to choose from:

- Sort: map an input list (either simple list or container list) to the operation then double-click to set the sort key(s).
- Filter: configuration as for Sort except other input fields can also be used to set the filter condition(s).
- Convert container list to simple list: map an input container list then double-click to select the simple list(s) required.
- Convert simple list to container list: reverse of the previous variant. Doesn't require any configuration.
- Group by fields: map an input container list then double-click to select the fields to group by.
- Ungroup fields: reverse of the previous variant. Double-click to select the container list that would be common to each group.
- Dedupe: de-duplicate list by selected fields, or return the duplicates.
- Enlist/delist fields. Enlist takes multiple input fields of the same type and puts them into a list of name/value pairs. Delist does the opposite, and can be simply achieved by copy/pasting the enlist operation and reconfiguring it by double-click.
- Compile/decompile basic list. Compile takes multiple input fields of the same type and puts them into a basic list. Decompile does the opposite, and can be simply achieved by copy/pasting the Compile operation and reconfiguring it by double-click.

File Operations

Read/write binary or text files. Select file path to file and operation type (text/binary, read/write/append). Operation on directories are available via [Call Function Fn](#).

Database Operations

Execute an SQL statement or call a stored procedure. The JDBC driver (Java Database Connectivity) must be known to Slurp (add JAR file(s) using menu Maintain>Plugins). Query substitution parameters can be inserted from fields mapped to the 'Parameters' input container. If using any '?' replacement parameters in the query then these must appear as the first items in the 'Parameters' input. For a non-select query, 'Parameters' is a list so that eg. multiple inserts or deletes can be performed per operation (preferably within a single transaction for efficiency). For select queries and stored procedure calls, the operation can optionally self-determine input/output fields and result set(s), or you can set them yourself.

By Default, database modification operations are auto-committed. To enable database transactions, modify the 'DB Transactions' setting in the service properties. Then, for subsequent database operations, the same transaction will be used to update the database, even if the operation is invoked by a called Slurp service (assuming it uses the same transaction isolation level). Then to commit or rollback all outstanding transactions, a service can invoke the corresponding operation in [Miscellaneous Operations](#). Note: some databases do not support all transaction isolation levels and some also do not give a warning that a selected isolation level is not supported.

JSON Parser/Generator

Parse or generate JSON text. There are 2 variants of this operation to choose from:

- Generate JSON: Map arbitrary input containers/fields to JSON text.
- Parse JSON: Map a single input text field to the operation and set the output containers/fields as expected for the JSON text.

XML Parser/Generator

Parse or generate XML text. There are 2 variants of this operation to choose from:

- Generate XML: Generate XML text from arbitrary input containers/fields.
- Parse XML: Parse XML text into arbitrary output containers/fields.

A Java compiler is needed to analyse an XSD or WSDL (see this [answer](#)). Normally, Slurp should be able to determine the compiler from your Java installation, but if not then consult that answer. A description of how Slurp maps XML to and from its fields/containers can be found in this [answer](#).

Call Restful Service

Call Restful web service. Setting the URL and method just sets the defaults for the operation inputs.

Call Webservice

Call web service. Limited to web services that are document literal. First, specify the WSDL for the service, which can be a local file path or a URL stating with 'http'. Then select the web service operation to be called. The input and output fields will be set automatically. A Java compiler is needed to analyse a WSDL (see this [answer](#)). Normally, Slurp should be able to determine the compiler from your Java installation, but if not then consult that answer. If you want to use a custom SOAP header it should be wrapped in elements `<soap:Header`

```
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"></soap:Header>.
```

Access Data Store

Perform an operation a data store (datastores can be defined via menu Maintain>Datastores). Select a datastore and the operation to be performed upon it. The input and output fields will be set automatically. A data store can be defined to be persisted, to have it's contents saved to file in a directory specified by a Slurp property (menu>Preferences). For a persisted data store:

- Locking it includes a lock on the store file itself. This can be used to synchronize activities between services or across multiple Slurp server instances.
- If you change the definition of the data store then you should remove or edit the store file to match your changes.

Call Function

Call one of a number of predefined functions or translators. A function library. These functions are also available in a data flow (double-click a data flow then right-click to 'Add Function'). They include generic list functions and lots more.

Miscellaneous

Other operations that do not fall into the main operation categories, including:

- Sleep. Suspend execution for the specified number of milliseconds, using the output to (potentially) activate another component when done.
- Send Mail. Send email to one or more recipients.
- Log Message. Log a formatted message to the logging system.
- Run Program. Run an ad-hoc program and potentially collect the results.
- DB Commit. Commits all outstanding database transactions, where connections were opened by a service (or one of its parent services) having transaction isolation level ('DB Transactions') set to anything but 'Auto Commit' when the connection was first used. This operation does not have to be called for a service with 'DB Transactions' set to 'Auto Commit' as each data base operation is committed automatically.
- DB Rollback. As for 'DB Commit' but performs a transaction rollback for all outstanding database transactions.

- Parse Properties. From properties text to a list of name/value pairs.
- Format Properties. From a list of name/value pairs to properties text.
- Calculate. Evaluate an arithmetic expression using field values, literal numbers or constants.

Invoke Java Plugin

Call a defined plugin operation. To build and add a plugin to the Service Designer, follow the steps described in [Writing and Using a Plugin](#). Once the plugin has been dropped on the canvas, it is treated like any other operation: configuration and execution are the same.

Running Slurp as a Stand-alone Data Service

Windows (Service)

- Download the version of prunsvr.exe for your platform from [Apache Commons Daemon](#).
- Place prunsvr.exe with slurp.exe (and slurp.xml, slurp.properties), in their own directory.
- Determine the full path to the JVM DLL to be used to run Slurp. The DLL is normally contained in the bin\server directory of the JRE installation.
- From a command prompt in the same directory as prunsvr.exe execute the following command (having substituted the texts in {}'s with your values and making sure the double-quote characters are the correct characters):

```
prunsvr install {service name} --StartClass=com.slurp.Slurp --
StopClass=com.slurp.Slurp --Classpath=slurp.exe --StopMethod=stop --
StartMethod=start --Jvm="{full path to jvm.dll}" --StartMode=jvm --
StopMode=jvm --StdOutput=stdout.log --StdError=stderr.log
```

- In the Windows Services list, right-click on the new Slurp service and select Properties. Set LogOn as Local System Account and, if required, set StartUp Type to Automatic.
- Start and stop the service as normal.
- The service can be removed as follows. From a command prompt in the same directory execute the command:

```
prunsvr delete {service name}
```
- To set particular JVM properties for the service add the option to the *prunsvr install* command when installing the service. Eg. to set maximum heap size to 1024Mb append " *--JvmMx=1024*" (lookup description of prunsvr).

- While the service is running it will be working from an in-memory version of slurp.xml and slurp.properties. During this time you can still run Slurp's GUI to make changes to your Slurp services, but the new configuration will not be used until you restart the Slurp service.

Windows (Command Line)

- Run a Slurp server from a command shell with the command `java -jar slurp.exe start`. The command remains in the foreground so can be terminated with control-c, or when the user session is logged out [Note: DO NOT use the command `slurp start` as this will confuse the executable wrapper].
- To set particular JVM properties, eg. to set maximum heap size to 1024Mb start server with `java -Xmx1024m -jar slurp.exe start`.



Unix (Daemon)

- Slurp.exe is also in JAR (Java Archive) format so the Unix daemon can be run with:
`nohup java -jar slurp.exe start&`
- Slurp still expects to find slurp.xml and slurp.properties in the same directory as itself.
- To stop the daemon gracefully:
`java -jar slurp.exe stop`
- To set particular JVM properties, eg. to set maximum heap size to 1024Mb start server with `nohup java -Xmx1024m -jar slurp.exe start&`.
- While the service is running it will be working from an in-memory version of slurp.xml and slurp.properties. During this time you can still run Slurp's GUI to make changes to your Slurp services, but the new configuration will not be used until you restart the Slurp daemon.

Unix (Command Line)

- You can run a Slurp server in the foreground from a command shell with command line: `java -jar slurp.exe start`. To terminate the server use command line: `java -jar slurp.exe stop`, or control-c.
- To set particular JVM properties, eg. to set maximum heap size to 1024Mb start server with `java -Xmx1024m -jar slurp.exe start`.

Writing and Using a Plugin

- 1) The plugin class must implement the `com.slurp.PluginAPI` interface. The 4 methods to implement are:
 - a) `getName()`. Should return a non-null string, preferably formatted as `<plugin package name>:<plugin operation name>`, eg. `sftp:put`
 - b) `getDescription()`. A brief description of the operation. Full documentation should be provided elsewhere.
 - c) `configure()`. Called to configure the input/output fields when the component is dropped onto the canvas.
 - d) `execute()`. Called when the component is executed.
- 2) The plugin can make use of the `com.slurp.schema.interFace` classes to define fields for `configure()` and the `com.slurp.runtime.service.data` classes to obtain and set data for `execute()`.
- 3) An exception thrown during `execute()` will invoke the standard error handling for a component. I.e. if the error output of the component  is used then the error can be handled by another component. Otherwise, the service will be prematurely terminated with a corresponding out-of-band error return.
- 4) References to Slurp classes will be satisfied by putting `slurp.exe` itself on the build path (it is also in JAR format).
- 5) Package one or more plugins into a JAR file.
- 6) Configure the plugins into the Service Designer by using menu `Maintain>Plugins`. First, specify the JAR files containing the plugins, then add the plugins themselves using the `Add...` button.
- 7) To use the plugin, drag the `Invoke Java Plugin` operation  onto the canvas and select the plugin. Then treat it as any other operation with regard to connecting to its inputs/outputs and conditions for activation.

Example Plugin

```
package my.plugins.maths;

import com.slurp.PluginAPI;
import com.slurp.runtime.service.data.RunData;
import com.slurp.runtime.service.data.RunInteger;
import com.slurp.schema.interFace.SlpDataField;
import com.slurp.schema.interFace.SlpFieldContainer;
import com.slurp.schemagen.interFace.FieldType;

public class Add implements PluginAPI {
    private static final String ARG1 = "Arg1";
    private static final String ARG2 = "Arg2";
    private static final String RESULT = "Result";

    @Override
    public String getName() { return "Maths:Add"; }

    @Override
    public String getDescription() { return "My add plugin"; }

    @Override
    public void configureIO(SlpFieldContainer input, SlpFieldContainer output) {
        input.getFields().add(SlpDataField.newInstanceRO(ARG1, FieldType.INTEGER, true, false));
        input.getFields().add(SlpDataField.newInstanceRO(ARG2, FieldType.INTEGER, true, false));
        output.getFields().add(SlpDataField.newInstanceRO(RESULT, FieldType.INTEGER, true, false));
    }
}
```

```
@Override
public void execute(RunData input, RunData output) throws Exception {
    RunInteger int1 = (RunInteger)input.getElement(ARG1);
    RunInteger int2 = (RunInteger)input.getElement(ARG2);
    long sum = int1.getValue()+int2.getValue();
    output.setElement(RESULT, new RunInteger(sum));
}
}
```

Server Clustering

- Multiple instance of Slurp can be run at the same time, either as Unix daemons or Windows services. For each instance, follow the previous instructions for running Slurp as a stand-alone data service for your platform.
- For each instance, the install directory can have symbolic links to a single version of slurp.exe and slurp.xml, but each install requires it's own copy of slurp.properties, to contain it's own values for 'localServerHttpPort' and/or 'localServerHttpsPort'. You may also want to adjust other properties for each instance.
- Sharing of data between instances can be achieved by sharing persistent datastores. Synchronization of activities between instances can be achieved by using the Lock/Unlock operations on a shared persistent datastore.

Frequently Asked Questions

- **Running slurp.exe on Windows does not find Java JRE, why not?**

There are 2 possible reasons for this. Either a Java JRE has not been installed at all, or it has NOT been installed as a stand-alone product (as part of another installation). The first issue requires you to download and install a Java JRE (version 1.8+). The second issue is because Windows Registry keys have not been set up because the Java installation program was not used. To compensate for this issue you either have to set your JAVA_HOME environment to the directory of your Java JRE or add the path to the java.exe to your PATH environment variable.

- **How to run Slurp Designer when I only have Putty access to my server?**

- a) Firstly, download an X Server for your desktop. For Windows you can download Xming from <https://sourceforge.net/projects/xming>. It will only show a window once you run Slurp from a Putty session.
- b) Secondly, set X11 forwarding in Putty. In Putty settings tab Connection/SSH/X11, check 'Enable X11 forwarding' and set 'X Display location' to 'localhost:0'. Then open a Putty session to your server. The Putty session should have the DISPLAY environment variable set.

Run Slurp with 'java -jar slurp.exe' and the forwarded GUI output will be displayed by an Xming window.

- **There are some input/output fields that I can't modify/delete, why?**

These are fields (in italic font) that the component has configured for itself and that it depends on. Normally, such fields and their default values are set when the component is (re)configured. For an input field that is read-only, you can nonetheless change its default value.

- **How are date/time values treated?**



The date/time type represents a local date/time as would be represented by a calendar date combined with a time, and normally input/displayed as 'yyyy-MM-ddThh:mm:ss'. On input, only the date part is required, which will assume a time of '00:00:00'. The precision of a date/time is actually to the nearest nanosecond, so if a date/time requires fractions of a second then it can be input/displayed with a dot and up to 9 digits after the seconds part. There are [Fn](#) functions to convert the date/time type between time zones. These use zone ids instead of plain time zones so that daylight savings time can be taken into account (java.time package).

- **How to format/parse dates/times?**


The pattern codes for formatting or parsing dates/times are described at [DateTimeFormatter](#). Example: to format a date/time value as 'day/month/year' use the format string 'dd/MM/yyyy'.

- **How do I control program flow?**

This is achieved by using control flows:

- a) Control flows can only emanate from [Conditional Flow Control](#) outputs  and the error output  of operations (the [And/Or Flow Control](#) can be used to amalgamate them if necessary). None of these control flows have to be used but they can be used to activate dependent components.
- b) By default a dependent component is only activated if ALL input control and data flows have been activated (this can be changed to 1 or more by right-clicking the component and selecting 'Set Activation by Flows: 1 or more').

- **Where are the program iteration constructs?**

- a) Service flow should not be looped. It will be stuck forever or until service timeout.
- b) Field mappings can be used to apply transformations to fields in a field hierarchy. These mappings apply iteratively to the fields of a container list.
- c) However, if more sophisticated transformations are required then to iterate over the containers in a top-level container list, right-click a sequence of one or more operations and 'Turn Container List Iteration ON'. If the container list is not at the top-level then make it the top-level by iterating the container list(s) above it. [To iterate over a basic data list, use a [List Operation](#) to convert it to a container list, iterate the container list, then use another List Operation to convert it back to a basic data list].
- d) Re-run the same service with different inputs by activating the Rerun Service component . [Note: container list iteration works on each container separately. If you want to accumulate information during iteration then use this method]
- e) Set 'Schedule Expression (cron)' in the service properties to have the service automatically re-run at specified times.

- **Input fields: what's the relationship between a field with a mapping, setting a field with a constant value and setting the default value for the field?**

Applying a mapping to an input field is mutually exclusive with setting a constant on the input field. You cannot have both at the same time, and you can have neither. The default value of the field comes into play if neither a field mapping nor constant have been specified or if a mapping to the field has been specified but it didn't result in the field being set (because the data flow wasn't activated). Setting a field with a constant value will always override its default value if both have been specified.

- **How to create a streaming data flow (for unlimited data transfers)?**

Connect a data flow/field mappings between a component that support streamed output to a component that supports streamed input (output/input field icons marked with an 's'). If the field mapping(s) are point-to-point only and don't split, merge or include a mapping function then the data flow line(s) will be rendered with a thick

black line and, at runtime, data will be streamed between the components. The components that currently support streaming are:

- a) DB Select List (output result set).
- b) DB Query (input parameter list).
- c) CSV Reader (input CSV text and output container list).
- d) CSV Writer (input container list and output CSV text).
- e) File Reader (output text/binary).
- f) File Writer (input text/binary).

Thus there is no limit to: a) the size of a CSV file resulting from a database query, or b) the volume of data that can be transferred between database tables. More streaming capabilities can be made available upon request.

Note: because streamed data is point-to-point only, then if two streamed inputs are flowed to a component then at runtime only one of the inputs will be selected as the streamed input and the component must be set to 'Activation by 1 or more flows' (or the component will not be activated and result in service hang/timeout).

- **What are the supported XML Renderings?**

If you don't use XML anywhere then you can ignore the 'XML Rendering' field property. For XML, you have the option of ELEMENT, ATTRIBUTE, SIMPLE_TYPE or CDATA_SECTION. The first 2 are fairly self explanatory if you have any familiarity with XML, and are the most commonly used. The SIMPLE_TYPE is equivalent to xsd:simpleType and CDATA_SECTION will render its contents between "<![CDATA[" and "]" so text can be included without need to escape XML syntax. The special field named 'SimpleContent' (reserved field name) should be used to represent the value contained in an xsd:simpleContent (which is allowed to contain attributes and a value but no sub-elements). This field can have rendering SIMPLE_TYPE or CDATA_SECTION, and only one of these should be present in a container that has attributes and no sub-elements.

- **Why does a Slurp generated XSD sometimes have a top-level element named 'root'?**

An XML document is required to have a root element, having xs:minOccurs=1 and xs:maxOccurs=1. For a Slurp input or output container to satisfy this constraint, it must have a single top-level field (container or data field with 'XML Rendering' set to 'ELEMENT') with attributes 'Required' and not 'List': If such a data or container field is the sole top-level field in the input or output container then the generated XSD will NOT have a 'root' element added. For all other cases a 'root' element will be inserted at the top-level. Whether a 'root' element is inserted or not, the XSD is the correct schema for the XML that can be used to populate the input container or that would be generated for an output container.

- **How are XML namespaces specified?**

The [Web Service Component](#) will use the *targetNamespaces* specified in the WSDL that was used to configure it. The [XML Parser/Generator Component](#) will use the *targetNamespaces* (if any) specified via the XSD used to configure it: if no XSD was used to configure it, and in all other cases, the namespace applied will be <https://slurpdata.com/io>.

If you happen to want to generate XML without namespace then configure the XML Generator with an XSD that doesn't specify a *targetNamespace*. This can be achieved as follows: having set the input fields for the XML Generator as you desire, then right-click on the actual input fields (not the operation inputs) and select 'Generate XSD for Field(s)'. Edit the XSD to remove *targetNamespace="https://slurpdata.com/io"* then use it to finally set the XML Generator.

- **Can I have a service-local variable?**

This can be achieved by defining an optional service input field, where its default value is effectively the initial value of the 'variable'. If you don't want to expose the field then wrap the service with another service that doesn't expose the 'variable' field. This caters for any type or shape of variables.

- **Why does Service Designer need a compiler to parse XSDs and WSDLs?**

The Service Designer uses the [Jakarta](#) JAXB and JAXWS tools to parse these file types, avoiding bespoke parsing and ensuring more accurate results. But these tools only create the equivalent Java classes that then need to be compiled so that they can be introspected. It's not a runtime requirement, and is the only dependency on the JDK (Java Development Kit). The Service Designer will use the compiler in the version of Java being used to run it. However, if you are only using a JRE (Java Runtime Environment) to run the Service Designer then you will have to tell it where to find the compiler in an installation of the JDK. You can use the Java compiler provided by [OpenJDK](#) without having to pay a license fee. The JDK version used should be the same or less than that used to run the Service Designer (which itself requires Java 1.8 or greater). Then specify the path for 'Java Compiler' in menu File>Preferences. Only a JRE is required to run Slurp services, not a JDK.

- **How to headlessly generate a license request string?**

From a command shell run: `java -jar slurp.exe license`

- **What to do if I can't or don't know how to implement some logic?**

You don't need a license to get help from [support](#), we're only too happy to help. In particular, if the function you need isn't currently built into Slurp when it should be then we'll build it into the next version (either natively or as a plugin), which will be available for download within hours (depending on your timezone). We envisage that larger/niche functionality will be provided as a plugin so as not to bloat the standard binary. There are currently plugins for SSH/SFTP, LDAP, SMB/CIFS and reading/writing Excel files.