Elixys Web Server Interface

Contents

[Overview 2](#_Toc294083357)

[Configuration 2](#_Toc294083358)

[User 3](#_Toc294083359)

[Client State 4](#_Toc294083360)

[Server State 4](#_Toc294083361)

[Sequences 5](#_Toc294083362)

[Components 8](#_Toc294083363)

[CASSETTE 8](#_Toc294083364)

[ADD 8](#_Toc294083365)

[EVAPORATE 8](#_Toc294083366)

[TRANSFER 8](#_Toc294083367)

[ELUTE 9](#_Toc294083368)

[REACT 9](#_Toc294083369)

[PROMPT 9](#_Toc294083370)

[MOVE 9](#_Toc294083371)

[INSTALL 9](#_Toc294083372)

[COMMENT 9](#_Toc294083373)

[ACTIVITY 9](#_Toc294083374)

[Reagents 10](#_Toc294083375)

[Run 10](#_Toc294083376)

[Run Sequence 10](#_Toc294083377)

[Manual Run 11](#_Toc294083378)

[Thoughts 13](#_Toc294083379)

# Overview

This document describes the internal interface between the web server and the core Python server. Many of these functions return a simple value indicating success or failure which can be used to inform the user if something went wrong. A user with sufficient privileges can then view the system logs to obtain additional error information in the event of a failure.

# Configuration

**GetConfiguration()** – Returns details of the Elixys system.

Parameters:

* None

Returns:

* Name – Name of this Elixys system (e.g. “Mini cell 3”).
* Version – System version (e.g. “2.0”).
* Debug – Boolean values that specifies if the client will display additional debug information to the user.
* Supported operations – Array of operations supported by this system (e.g. “Add”, “Evaporate”, etc.).

**GetUserAccessLevels()** – Returns the user access levels recognized by this system.

Parameters:

* None

Returns:

* User access level strings – Array of valid user access level strings (e.g. “Administrator”, “Tech”).

# User

**GetUser()** – Returns details of the given user.

Parameters:

* Username – Name of the user (e.g. “devel”).

Returns:

* User access string – String describing the user access level (e.g. “Administrator”).
* User access value – Word describing the permissions of the user’s access level. This value will be the same for all users with the same access level. Each bit of this string corresponds to a given action, e.g. create/edit sequence, run sequence, create/edit/delete users, etc.

**SaveUser()** – Creates a new user in the system or updates an existing user.

Parameters:

* Username – Name of the user (e.g. “devel”). If this name corresponds to an existing user then that user will be updated, otherwise a new user will be created.
* Password – MD5 hash of the user’s password if (1) this is a new user or (2) the user’s password is being changed. This field can be blank otherwise.
* User access string – String describing the user’s desired access level if (1) this is a new user or (2) the user’s access level is being changed. This field can be blank otherwise. This string must match a predefined user access level as returned by **GetUserAccessLevels()**.

Returns:

* Result – Boolean value (true on success, false otherwise).

**DeleteUser()** – Deletes the specified user from the system.

Parameters:

* Username – Name of the user (e.g. “devel”).

Returns:

* Result – Boolean value (true on success, false otherwise).

# Client State

**GetClientState()** – Returns the state of the client.

Parameters:

* None

Returns:

* Client state – String describing the state of the client (e.g. “HOME”). This string may be delimited and contain state-specific information that will be understood by the web server (e.g. “VIEWSEQUENCE.14.52”) but should be treated by the core server as just a string.

**SaveClientState()** – Saves the client state to the database.

Parameters:

* Client state – String describing the state of the client (e.g. “HOME”).

Returns:

* Result – Boolean value (true on success, false otherwise).

# Server State

**GetServerState()** – Returns the state of the server.

Parameters:

* None

Returns:

* Running – The following values will be returned if the system is currently running a synthesis:
  + Status – String describing the current system status (e.g. “Reacting, 8:23 minutes”).
  + Active reactor – The active reactor number.
  + Username – The username of the person operating the system.
* Pressure regulators – Details of each pressure regulator in the system:
  + Name – String describing the pressure regulator (e.g. “Main value pressure”).
  + Set pressure – The target pressure in millimeters of mercury.
  + Actual pressure – The actual pressure in millimeters of mercury.
* Cooling – Boolean values that specifies if the cooling system is on.
* Vacuum – Boolean value that specifies if the vacuum system is on.
* Door – Boolean value that specifies if the main door is open.
* Reagent Robot – Details of the reagent robot:
  + X – Gives the X position of the robot in millimeters.
  + Y – Gives the Y position of the robot in millimeters.
  + Actuator – String that specifies the state of the actuator. Possible values are “up”, “down” and “indeterminate”.
  + Gripper – Boolean value that specifies if the robot gripper is closed.
* Reactors – Details of each reactor:
  + Number – The reactor number.
  + Set temperature – The set temperature of the reactor in degrees Celsius.
  + Actual temperature – The actual temperature of the reactor in degrees Celsius.
  + Position – The reactor position.
  + Vial – The vial state. Possible values are “up”, “down” and “indeterminate”.
  + Activity – The last know radiation activity level of the active reactor in millicuries.
  + Activity time – The time the activity was last measured.
  + Evaporation valves – Boolean value that specifies if the evaporation values (nitrogen and vacuum) are open.
  + Transfer valve – Boolean value that specifies if the transfer valve is set to waste.
  + Reagent 1 transfer valve – Boolean value that specifies if the first reagent transfer valve is open.
  + Reagent 2 transfer valve – Boolean value that specifies if the second reagent transfer valve is open.
  + Stopcock 1 valve – Boolean value that specifies if the first stopcock valve is activated.
  + Stopcock 2 valve – Boolean value that specifies if the second stopcock valve is activated.
  + Stopcock 3 valve – Boolean value that specifies if the third stopcock valve is activated.

# Sequences

**GetSequenceList()** – Returns a list of sequences in the database:

Parameters:

* Type – String describing the type of sequence. Possible values are “Saved” and “Manual” for saved sequences and manual runs, respectively.

Returns:

* The following data are returned for each sequence:
  + Name – Sequence name.
  + Date – The date the sequence was created.
  + Time – The time the sequence was created.
  + Comment – Any comment associated with the sequence.
  + ID – Unique ID that is sent from the client to the server when the user selects the sequence.
  + Creator – User that created the sequence.
  + Operations – Number of operations.

**GetSequence()** – Returns full details of a sequence.

Parameters:

* Sequence ID – String that uniquely identifies the sequence.
* Component ID – Optional string that uniquely identifies a single component in the sequence. If this parameter is specified then only details for that specific component will be returned, otherwise details for all sequence components will be returned.

Returns:

* Sequence metadata – Returns metadata for this sequence as described above in **GetSequenceList()**.
* Component information – Returns the following information for either the single sequence component or all sequence components:
  + Component Type – String that specifies the type of component. Possible values are the subheading under **Components** (e.g. “EVAPORATE”).
  + Component ID – Unique ID that is used by the client to refer to the component when communicating with the server.
  + Reactor – The reactor associated with this component.
  + Validation error – Boolean value that indicates if this component has a validation error.
  + Additional details – Each component type contain additional information as documented below under **Components**.

**SaveSequence()** – Creates a new sequence in the system or updates an existing sequence’s metadata.

Parameters:

* Sequence ID – The ID of the sequence to update or blank to create a new sequence.
* Type – String describing the type of sequence. Possible values are “Saved” and “Manual” for saved sequences and manual runs, respectively.
* Sequence metadata – Metadata for the sequence that is being created or updated:
  + Name – Sequence name.
  + Comment – Any comment associated with the sequence.
  + Creator – User that created the sequence.

Returns:

* The ID of the newly created or updated sequence on success, blank otherwise.

**SaveSequenceComponent()** – Creates a new component in an existing sequence or updates an existing sequence component.

Parameters:

* Sequence ID – The ID of the sequence to associated with the component.
* Component information – Returns the following information for the sequence component:
  + Component Type – String that specifies the type of component. Possible values are the subheading under **Components** (e.g. “EVAPORATE”).
  + Component ID – The ID of the component to update or blank to create a new component.
  + Reactor – The reactor associated with this component.
  + Additional details – Each component type contain additional information as documented below under **Components**.

Returns:

* The ID of the newly created or update sequence component on success, blank otherwise.

**DeleteSequence()** – Deletes a sequence from the database.

Parameters:

* Sequence ID – The ID of the sequence to delete.

Returns:

* Result – Boolean value (true on success, false otherwise).

**DeleteSequenceComponent()** – Deletes a component from a sequence.

Parameters:

* Sequence ID – The ID of the sequence to associated with the component.
* Component ID – The ID of the component to delete.

Returns:

* Result – Boolean value (true on success, false otherwise).

**CopySequence()** – Duplicates a sequence. The source sequence may be either a saved sequence or a manual run. The sequence copy will always be a saved sequence.

Parameters:

* Sequence ID – The ID of the sequence to copy.
* Sequence metadata – Metadata of the new sequence:
  + Name – Sequence name.
  + Comment – Any comment associated with the sequence.
  + Creator – User that created the sequence.

# Components

Each component type has specific information associated with it in addition to the common information listed under **GetSequence()** and **SaveSequenceComponent()**. All of this information is writable by the latter unless explicitly indicated below as read-only.

## CASSETTE

The cassette component contains the following additional information:

* Used – Boolean value that indicates if this cassette is used.
* Reagents – Array of reagent IDs (read only).

## ADD

The add component contains the following additional information:

* Reagent – The reagent ID to add to the reactor.

## EVAPORATE

The evaporate component contains the following additional information:

* Duration – The length of the reaction.
* Evaporation temperature – The evaporation temperature in Celsius.
* Final temperature – The final temperature in Celsius.
* Stir speed – The stir speed in rotations per minute.

## TRANSFER

The transfer component contains the following additional information:

* Target – The target ID.

## ELUTE

The elute component contains the following additional information:

* Reagent – The reagent ID.
* Target – The target ID.

## REACT

The react component contains the following additional information:

* Position – The react position.
* Duration – The length of the reaction.
* Reaction temperature – The reaction temperature in Celsius.
* Final temperature – The final temperature in Celsius.
* Stir speed – The stir speed in rotations per minute.

## PROMPT

The prompt component contains the following additional information:

* Message – Text to display to the user.

## MOVE

The move component contains the following additional information:

* Position – The react position.
* State – Boolean value that specifies if the reactor is in the closed state.

## INSTALL

The install component contains the following additional information:

* Message – Text to display to the user.

## COMMENT

The comment component contains the following additional information:

* Comment – User-specified comment.

## ACTIVITY

The activity component contains no additional information.

# Reagents

**GetReagent()** – Returns details of a specific reagent.

Parameters:

* Sequence ID – Unique ID of the sequence associated with this reagent.
* Reagent ID – The unique ID of the reagent.

Returns:

* Used – Flag that indicates if this reagent position is used in this cassette.
* Position – The reagent position in the cassette.
* Name – The short name of the reagent.
* Description – The long description of the reagent.

**SaveReagent()** – Updates an existing reagent.

Parameters:

* Reagent ID – Unique ID that specifies the reagent to update.
* Component ID – The unique ID of the cassette associated with this reagent.
* Sequence ID – The unique ID of the sequence associated with this reagent.
* Used – Flag that indicates if this reagent position is used in this cassette.
* Position – The reagent position in the cassette.
* Name – The short name of the reagent.
* Description – The long description of the reagent.

Returns:

* Result – Boolean value (true on success, false otherwise).

# Run

## Run Sequence

**RunSequence()** – Starts executing a sequence from the database.

Parameters:

* Sequence ID – Unique ID of the sequence to run.

Returns:

* Result – Boolean value (true on success, false otherwise).

**AbortRun()** – Aborts the run that is in progress.

Parameters:

* None

Returns:

* Result – Boolean value (true on success, false otherwise).

**ContinueRun()** – Continues the run that has paused for a Prompt or Install unit operation.

Parameters:

* None

Returns:

* Result – Boolean value (true on success, false otherwise).

## Manual Run

**StartManualRun()** – Starts a manual run and create a new manual run sequence in the database.

Parameters:

* None

Returns:

* Sequence ID – Unique ID of the newly created manual run sequence or blank on error.

**PerformOperation()** – Performs a unit operation that has been added to the manual run sequence using **SaveSequenceComponent()**.

Parameters:

* Component ID – The unique ID of the sequence component. This unit operation will always be the last one in the sequence and will have just recently been added.
* Sequence ID – The unique ID of the manual run sequence that is associated with the component.

Returns:

* Result – Boolean value (true on success, false otherwise).

**AbortOperation()** – Abort the unit operation that is in progress.

Parameters:

* None

Returns:

* Result – Boolean value (true on success, false otherwise).

**ContinueOperation()** – Continues the operation that has paused for a Prompt or Install unit operation.

Parameters:

* None

Returns:

* Result – Boolean value (true on success, false otherwise).

**FinishManualRun()** – Completes the manual run and releases the lock on the Elixys system.

Parameters:

* None

Returns:

* Result – Boolean value (true on success, false otherwise).

# Thoughts

* This document currently contains the parameters and return values for each call but says nothing about the actual data format. What do we want to do in terms of data format? Possible options include:
  + Python dictionaries and lists – Each of the above functions takes and returns loosely-typed python data structures.
  + Classes – Create a set of strongly-types classes that encapsulate each data type. Some of these may be extensions the existing UML classes such as unit operations.
* The client will allow the user to create and save a sequence with validation errors, meaning the server will need to validate the entire sequence before starting a run. This same code can be used to set the validation flag for each component returned by the **GetSequence()** function. Note that this validation is independent from the validation done by the client. The client validation is based on the validation string it receives from the server.
* We need to provide functions for viewing, filtering and exporting the system logs. This functionality also needs to be added to the HTTP protocol and client user interface.
* Do we need to provide some way to adjust the pressure regulator setting?
* What units do we want to use for pressure, robot position, temperature and activity?