Java based Scripts

Scripts can also be written in Java, which allows for tight integration with Agiloft and direct access to all user data. Note this page refers to scripts written in Java, not JavaScript.

Java scripts are a flexible and performance-effective way to implement custom logic. You can retrieve only the necessary data and also skip export to XML, running an external program, and the import results steps. The result is a significant performance boost in the case of a complex table structure with many linked fields.

The script class must implement `com.supportwizard.actions2.interfaces.ExternalScript`.

This interface has only one method: `ScriptOutput runScript (final ScriptInput input) throws ActionException;`

Please consult Javadoc documentation for more details.

Input Data

A ScriptInput instance provides input data: project and user table IDs, user's Seance, and the record data.

The record data is passed in a map form: an instance of com.supportwizard.dml.SWDataMap class. Specific field values can be accessed by name, as visible in GUI in **Setup > Tables > [Select Table] > Edit > Field**. The script gets one or two instances of the data map: the "old" one and the "new" one. At least one will be present, depending on how or when the script is invoked:

- Create is only for "new" instances
- Delete is only for "old" instances
- Modify can be used for both instances.

This is the mapping of specific data types returned for different field types:

- Auto-Increment => class java.lang.Long
- Elapsed Time => class java.lang.Long
- DAO3 link field => class com.supportwizard.functionalities.dao3.util.SWDao3LinkHolder
- Long integer field => class java.lang.Long
- Billing field => class com.supportwizard.functionalities.dao3.util.SWDao3LinkHolder
- Integer => class java.lang.Integer
- Choice => class com.supportwizard.dictionary.SWChoiceLine
- EMail => class java.lang.String
- EMail Pager => class java.lang.String
- Telephone/Fax => class java.lang.String
- Multi-Choice => class com.supportwizard.dictionary.MultichoiceLines
- Short Text => class java.lang.String
- Password => class java.lang.String - however it comes in ***** form for security reasons and is generally useless to a script writer

- Text => class java.lang.String
- URL => class java.lang.String
- File => class com.supportwizard.functionalities.blob.SWBlobRefHolder
- Image => class com.supportwizard.functionalities.blob.SWBlobRefHolder
- History => class java.lang.String
- DAO3 multiple field => class com.supportwizard.functionalities.dao3.util.SWDao3MultiValue
- WMI Field => class java.lang.String
- Append Only Text => class com.supportwizard.dictionary.appendtext.AppendOnlyTextContainer
- Floating Point => class java.lang.Double
- Percentage => class java.lang.Double
- Currency => class java.lang.Double
- Date/Time => class java.sql.Timestamp
- Date => class java.sql.Timestamp
- Time => class java.sql.Time
- Compound => class java.lang.String
- Calculation on Multiple Linked Records => class java.lang.Double

A linked field that imports N field from the source table into the target one will be represented by N+1 entries in the "input" Map on the data level.

If a single record is imported each of the N entries corresponding to N fields will have a single value of a corresponding type. If multiple records are imported, each of the N entries corresponding to the N fields will have an instance of SWDao3MultiValue, a list of String values (one for each imported record), with a "*_{NULL}_*" value marking the null one. The sequence of values in different multi-value fields is the same for the same set of imported records.

The extra entry mentioned above will be under the name that can be seen in the GUI in **Setup > Table > [Table Name] > Edit > Fields > [Linked Field Name] > Edit > Fields** at the bottom of "Linked Column Name." It will contain an instance of the SWDao3LinkHolder, which is essentially an array of identifiers for the records in the source table(s).

To access the N-th identifier:

```
List<SWDao3LinkHolder.Link> links = linkHolder.getLinks();
SWDao2LinkHolder.Link link = links.get;
Long pk = link.getLinkPK();
```

A script call always starts with the input object and is called for a single record only. A script, however, can use the CRUD+S (select) API underlying WebServices and REST access: SimpleAPI.

```
Context jndiContext = new InitialContext();
EWSimpleAPILocalHome localHome = (EWSimpleAPILocalHome)
                                 jndiContext.lookup("ew/EWSimpleAPI");
EWSimpleAPI EWSimpleAPI = localHome.create();
long[]  ids = EWSimpleAPI.EWSelectFromTable("allocation",
                "general_issue=" + general_issue +
                " and specific_issue=" + specific_issue +
                " and default_team = " + line.getId(), seance);
if (ids.length == 0) {
        return blockedScriptOutput(output,
       "There is no default team defined for this combination of Issue Types.");
}
```

For more details on SimpleAPI and other Web Services APIs, please consult the following Javadoc: WS_API_Javadoc.zip. Unzip the file, open index.html, and navigate to EWSimpleAPI on the left pane. Please contact Agiloft Support if you need assistance.

Output Data

A ScriptOutput instance describes script output, an instance of this class that is intended to be created with ScriptInput- final the ScriptInput object's createOutput() method. ScriptOutput can optionally contain modified user record data - don't change the ScriptInput.getRecord() instance.

Note: Do not copy any values your script leaves unchanged from the input record to output. Doing so is unnecessary, and could have unexpected results if you are filling linked fields with multiple columns.

Modifications will take place if ScriptOutput.getExitCode() is ExternalScript.SUCCESS_CODE. If exit code is equal to BLOCKED_CODE, then the user action will be blocked and the user will get ScriptOutput.getMessage() as an error message; this message will also be shown in case of ACCEPT_CODE. In cases where the user action is undefined, like in the section about timer based rules above, nothing will be blocked and the error message will only be logged.

BLOCK_REDIRECT_CODE will also block user action and will force the current user to log out. The user will be redirected to ScriptOutput.getRedirect(). ACCEPT_REDIRECT_CODE will log out the current user, but won't block an action, and script changes will take place.

Unhandled exceptions propagate to Agiloft where they are caught, logged, and displayed to the user depending on the way the script was run. This display is often too excessive, and is not really suited for ordinary users. For ordinary users, the recommended way of handling exceptions is to either suppress them in the script itself and instead return a ScriptOutput instance with a blocking exit code and an optional message, or wrap in action-related exceptions.

```
    private ScriptOutput blockedScriptOutput(ScriptOutput output, String s,
    SWDataMap newRecord, Seance seance) {
        output.setExitCode(ExternalScript.BLOCK);
        output.setMessage(s + " ID:" + newRecord.getSWRecordPK(seance).getID());
        return output;
    }
```

Wrapped exceptions can be used to change control flow. A plain ActionException indicates some system-level error, and leads to a transaction rollback. Throwing an ActionBlockedException String message amounts to returning BLOCKED_CODE with the ScriptOutput.getMessage() being simply 'message'.

Throwing ActionBlockedException, like String message and String redirectURL, will do the same thing as returning BLOCK_REDIRECT_CODE.

General Considerations

The call to the script is done in a synchronous manner. The calling code waits. There is a potential to the encompassing business transaction to time out and be rolled back by Agiloft's application server if it takes too long. However, if this occurs frequently, you may consider implementing an asynchronous mode of operation on the caller side.

A new classloader is created on each script run, as well as the script class instance. It is not isolated as far as it delegates to a parent classloader in a java 2 compliant manner. The script doesn't have to be thread safe, since each call is being served by a different instance.

If a script relies on some 3rd party libraries that are not available in the Agiloft environment, these have to be repackaged into the script archive.

The Agiloft .jar library files and their locations depend on your Agiloft version:

	2019_01 or later	**2018_02 or prior**
Directory	{Agiloft.installation.dir}/wildfly/modules/system/layers/base/com/agiloft/main/lib	{Agiloft.installation.dir}/jboss/server/sw/lib/sw/ *and* {Agiloft.installation.dir}/jboss/server/sw/deploy
Main Library Files	▪ agiloft-core.jar	▪ SWFunctionalities.jar ▪ SW2Interfaces.jar ▪ SWSeance-ejb.jar
Additional Library Files	▪ commons*.jar ▪ httpclient*.jar ▪ xml*.jar	▪ commons*.jar ▪ httpclient*.jar ▪ xml*.jar

The Agiloft .jar files you may need while developing are in { Agiloft.installation.dir}/wildfly/modules/system/layers/base/com/agiloft/main/lib. In older versions of Agiloft, you can find them in { Agiloft.installation.dir}/jboss/server/sw/lib/sw and { Agiloft.installation.dir}/jboss/server/sw/deploy.

Normally, the following classes are enough:

- SWFunctionalities.jar
- SWSeance-ejb.jar

The .jar files that make the JBoss environment would be under { Agiloft.installation.dir}/jboss/lib, { Agiloft.installation.dir}/jboss/server/lib.

Deployment Considerations

Scripts must be stored in the scripts directory. By default, the scripts directory is: {Agiloft.installation.dir}/data/{kb.name}/scripts

If your script consists of a single class, you can simply drop the .class file in the scripts directory. This .class must not belong to any package. If you have several classes, put them in a .jar file with a special key in the manifest file:

```
Script-Class: {name of the class implementing ExternalScript, such as com.mycompany.test.TestExternalScript}
```

Now your script name will be the name of the script .jar or .class file.

A Java custom script is run by Agiloft within the same Java Virtual Machine. The script can use all libraries present in the Agiloft application server instance classpath: {Agiloft.installation.dir}/wildfly/modules/system/layers/base/com/agiloft/main/lib

Or, in older versions, these classpaths:

- {Agiloft.installation.dir}/jboss/server/sw/lib/
- {Agiloft.installation.dir}/jboss/server/sw/lib/sw/
- {Agiloft.installation.dir}/jboss/lib

If you want to add a new library to run a custom Java script, choose one of these two options:

- Set the Class-Path parameter in MANIFEST.MF file of your script .jar file.

  ```
  Class-Path: <list of additional jars to be used with this jar>
  ```

  This parameter specifies these additional .jars or the path to the additional .jars. For example: {Agiloft.installation.dir}/data/{kb.name}/scripts/proprietary_libs directory. When the custom script starts, it will load all these dependent .jars from the specified directory.
- Add a reference to the new library in the module.xml file in the /wildfly/modules/system/layers/base/com/agiloft/main directory. However, these changes will be lost when you upgrade to a newer version of Agiloft.

Agiloft runs on Java 6. For JAXWS, the script has access to the libraries that are distributed with the application server, rather than those available from the runtime environment.