Creating an example plugin: Hello World

This guides describes the steps required to create a Deployment Automation plug-in that prints “Hello World!” to the console and how this can be extended.

1.0 Plugin anatomy

A Plugin will consist of the following three major elements:

1. A directory name to identify the plugin, under which the relevant plugin source and description files will exist
2. A .groovy files contains the source code that describes the actions of the plugin
3. Several .xml files that describe the plugin

Note: For comprehensive documentation on using and creating plug-ins, please see the Serena Deployment Automation Plug-ins Guide here.

2.0 Plugin creation steps

2.1 Create your folder:

First create a folder and name it **HelloWorld-1.0**. All the necessary Groovy and XML files will be placed in here. The **1.0** in the filename represents the current version of the plug-in. This is useful as it makes it easy to quickly figure out which version of the plug-in is currently being used.

2.2 Create HelloWorld.groovy:

First a groovy file called **HelloWorld.groovy** in the the HelloWorld-1.0 with the following content:

```
println "Hello World!"
```

This code will print *Hello World!* to the console.

2.3 Create a plugin.xml file:

Create the file **plugin.xml** in the HelloWorld-1.0 folder, paste the following code in the file:

```
<?xml version='1.0' encoding='UTF-8'?>

<plugin xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xmlns='http://www.urbancode.com/PluginXMLSchema_v1'>

<header>

    <identifier version='1' name='HelloWorld' id='com.urbancode.air.plugin.HelloWorld' />

    <description>Plugin which prints Hello World!</description>

    <tag>System Utility/HelloWorld</tag>

</header>

<step-type name='HelloWorld'>

   <description>Print Hello World</description>

   <properties></properties>

   <post-processing>

       <![CDATA[ if (properties.get("exitCode") != 0) { properties.put(new java.lang.String("Status"), new java.lang.String("Failure")); } else {
properties.put("Status", "Success"); } ]]>

   </post-processing>

   <command program='${GROOVY_HOME}/bin/groovy'>

       <arg file='HelloWorld.groovy' />

   </command>

</step-type>

</plugin>
```

Header structure:

The **<header>** tag is used to identify the plug-in.

The **<identifier>** tag supplies the current version of the plug-in (version='1'), its name (name='HelloWorld') and a unique identifier for the plug-in (id='com.urbancode.air.plugin.HelloWorld').

The **<description>** tag usually contains some text outlining the purpose of the plug-in as a whole.

The <tag> element defines the folder structure in which the plug-in will be placed in Serena Deployment Automation. This is important as this determines where the plug-in and its steps can be found on the process creation page. In this example, the HelloWorld plug-in is placed in the System Utilities folder.

The step for the plug-in is defined using the **<step-type>** tag. The name property of the <step-type> tag labels the step. The <description> tag within the <step-type> tag summarizes the purpose of the step (or the functionality it provides). This is followed by the <properties> tag which can be used to define properties for the step (using the nested <property> tag), however, since this step will simply be printing out "Hello World!" to the console, there is no need for the step to have any properties.

The <post-processing> tag can serve many purposes. It can be used to determine whether the step executed successfully, it can also set the step's output properties and do some error handling. The <post-processing> tag is executed after the step has finished executing its primary function (which in this case is to print "Hello World!" to the console). The <post-processing> tag can contain a JavaScript script. It is recommended (but not compulsory) to wrap this script in a CDATA element. The script above accesses the "exitCode" property of a Java properties element and uses it to determine whether the step was successful or not.

The **<command>** tag performs the step's function. It has access to both the step's own properties and the properties set earlier by other steps in the process. The <command> element specifies the scripting tool that is invoked to execute the program. The program attribute of the command element defines the location of the tool. The <arg> child element is used to pass any arguments that are required for a successful execution. In this case, we have used the file attribute of the <arg> element to specify the Groovy file that is to be executed.

2.4 Create an info.xml file:

Copy the code shown below and paste into a file called **info.xml** in the HelloWorld-1.0 folder:

```
<?xml version='1.0' encoding='UTF-8'?>
  <plugininfo>
    <author name='UrbanCode'>
         <organization>UrbanCode</organization>
         <email>info@urbancode.com</email>
         <website>http://urbancode.com</website>
    </author>
    <integration type='Deploy' />
    <tool-description>HelloWorld is a sample plugin</tool-description>
    <release-version>1.0</release-version>
    <release-notes>
      <release-note plugin-version='1'>Initial release</release-note>
    </release-notes>
</plugininfo>
```

The **info.xml** file provides information to the user about the plug-in. Although this is optional, it is quite useful for a plug-in to have this file from a user's perspective. Much of the contents of the file are self-explanatory. The <integration> tag indicates where the plug-in fits into the process. Accepted types are: SCM, Build, Artifact, Automation and Deploy. Admittedly, the plug-in we are creating has nothing to do with deploying, but I have put it there for completeness.

The <tool-description> tag is used to provide a summary of the plug-in. The contents of the <release-version> tag will be injected by the build process. Finally, the <release-notes> tag provides information (such as changes, improvements, or bug fixes) about the release. The plugin-version attribute should match the plug-in version in the plugin.xml file.

2.5 Create the upgrade.xml file:

The **upgrade.xml** is mandatory for a plug-in. However, considering that we are not making an upgrade for our plug-in, the only contents that need to be in this file are the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin-upgrade xmlns="http://www.urbancode.com/UpgradeXMLSchema_v1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">


</plugin-upgrade>
```

2.6 Zip and Load the plug-in into Serena Deployment Automation:

To test that you have created the plug-in correctly, place all the contents of the **HelloWorld-1.0** folder into a zipped file. For example

```
cd HelloWorld-1.0/

zip -R HelloWorld-1.0.zip *
```

Output similar to below should be returned

```
C:\work\HelloWorld-1.0>zip -R HelloWorld-1.0.zip *

  adding: HelloWorld.groovy (140 bytes security) (stored 0%)

  adding: info.xml (140 bytes security) (deflated 48%)

  adding: plugin.xml (140 bytes security) (deflated 47%)

  adding: upgrade.xml (140 bytes security) (deflated 23%)
```

Then load the zipped plug-in into Serena Deployment Automation by going to **Administration > Automation > Automation Plugins > Load Plugin**.

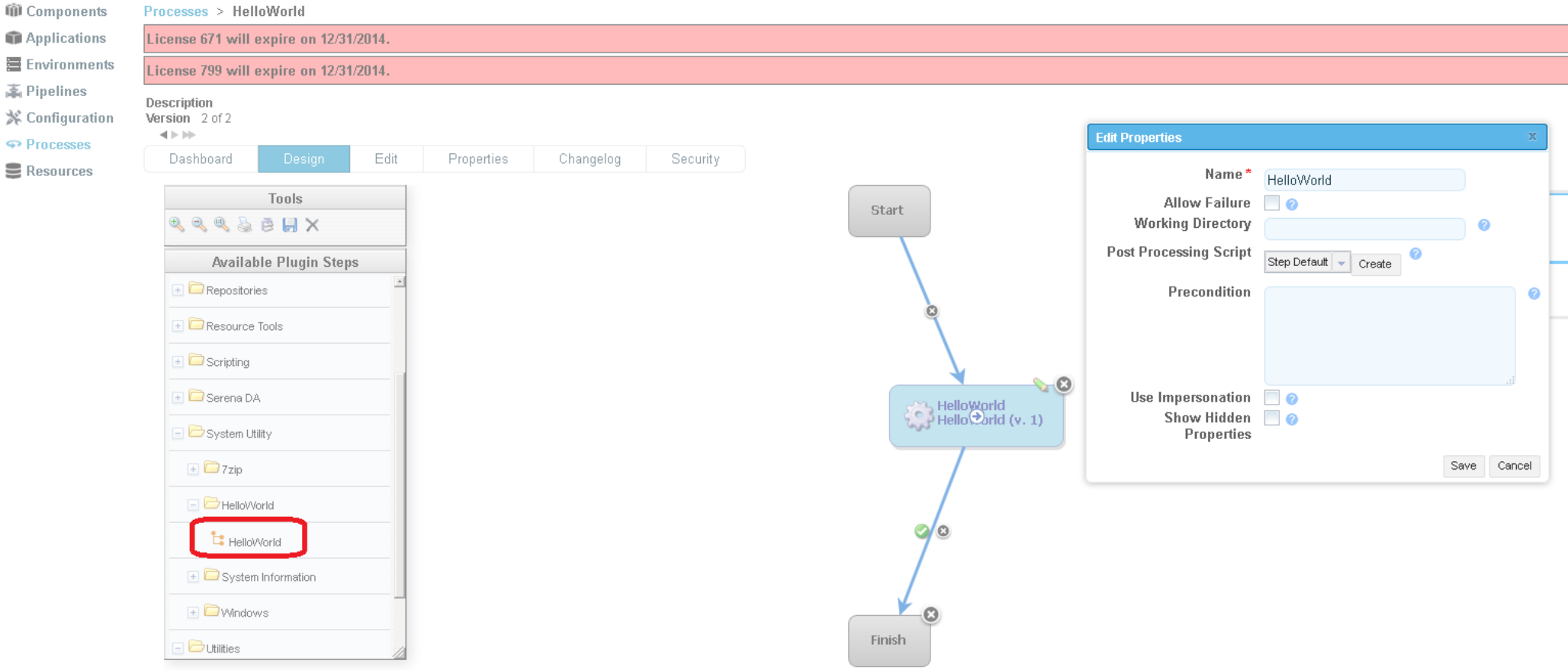
Browse for the zipped file on your computer and click the **Submit** button.

The text: **Plugin loaded successfully!** Should appear click on OK and the Hello World Plugin should also appear in the list of available plugins

Plugin	Description	Version	Ac
File Utils	The FileUtils plugin allows users to perform folder and file level tasks as part of their deployment process. Among others this plugin includes steps for deleting or creating directories and replacing tokens in a file	31_156	De
HelloWorld	Plugin which prints Hello World!	1_1.0	De
Resource Properties Collector	Tools used for retrieving env and resource properties from the resource.	2_153	De

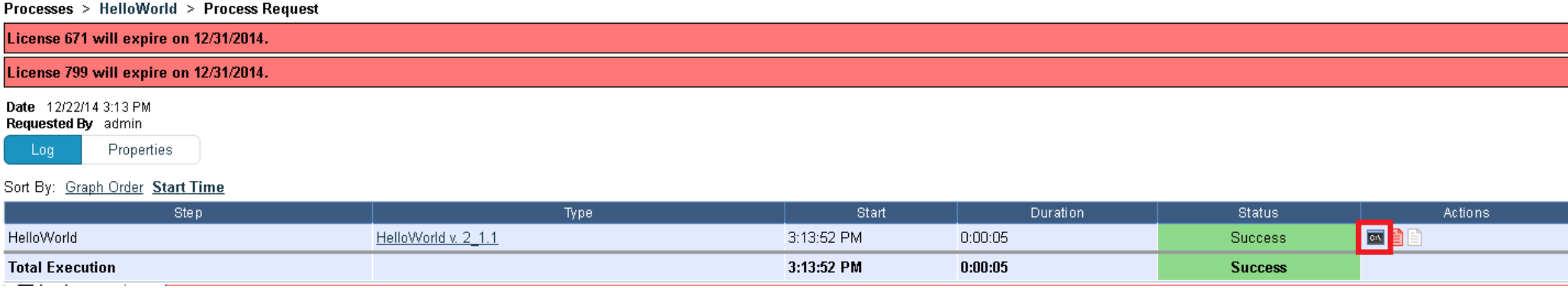
2.7 Test the plug-in in a process in SDA

- Finally, create a process that uses the step defined in the HelloWorld plug-in and view the console output after the step has been executed.
- In SDA go to **Management | Processes**
- Select **Create Process** and Specify **Hello World**
- Click on the **Design** tab and under **Available Plugin Steps** select **System Utility | Hello World | Helloworld** and add it as a process step to the process, the click on Save to store the new step.

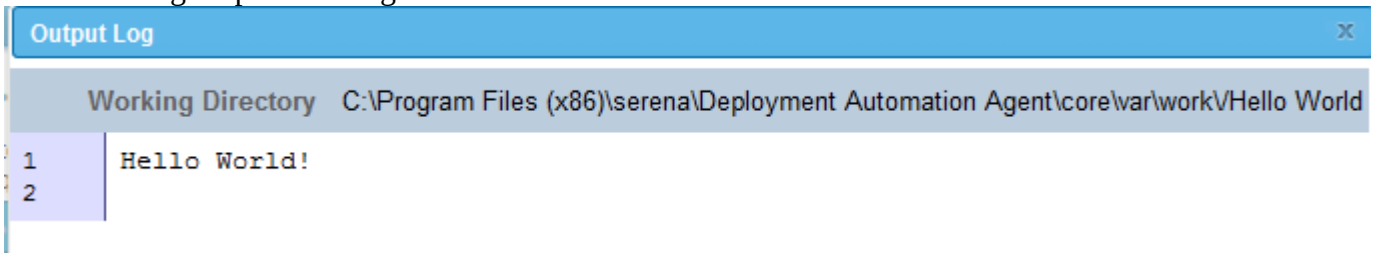


- Click on save to store the new process flow.

- Select the **Dashboard** tab ensure a resource is selected, then click on the **Submit** button.
- Once the process has completed click on **the console icon** (highlighted below)



- The following output will be generated



3.0 Extend the plugin

Now extend the plugin to utilise a property defined within the plugin step which is then referenced in the groovy script.

To do this, a new version of the existing plugin will need to be created, this will require

1. Modifying of the following files: plugin.xml, HelloWorld.groovy, info.xml
2. Loading the new version of the plugin into SDA
3. Modifying the existing process to use the revised version of the plugin step

3.1 Extend the plugin definition to contain a string property called Name

The following modifications made to **plugin.xml**, the modification is highlighted in green the additions are highlighted in blue.

```
<?xml version='1.0' encoding='UTF-8'?>

<plugin xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xmlns='http://www.urbancode.com/PluginXMLSchema_v1'>

<header>

<identifier version='2' name='HelloWorld' id='com.urbancode.air.plugin.HelloWorld' />

<description>Plugin which prints Hello World!</description>

<tag>System Utility/HelloWorld</tag>

</header>0020


<step-type name='HelloWorld'>

<description>Print Hello World</description>

<properties>

      <property name="GreetingName">

        <property-ui description="Identify yourself." label="Greeting Name" type="textBox"/>

      </property>

</properties>

<post-processing>

<![CDATA[ if (properties.get("exitCode") != 0) { properties.put(new java.lang.String("Status"), new java.lang.String("Failure")); } else {
properties.put("Status", "Success"); } ]]>

</post-processing>

<command program='${GROOVY_HOME}/bin/groovy'>

<arg file='HelloWorld.groovy' />

<arg file='${PLUGIN_INPUT_PROPS}' />

<arg file='${PLUGIN_OUTPUT_PROPS}' />

</command>

</step-type>
```

```
</plugin>
```

3.2 Modify the groovy code to print the value of the property Name rather than the text World

The following modifications made to **HelloWorld.groovy**, the additions are highlighted in blue.

```
final def workDir = new File('.').canonicalFile
final def props = new Properties();
final def inputPropsFile = new File(args[0]);

try {
    inputPropsStream = new FileInputStream(inputPropsFile);
    props.load(inputPropsStream);
}
catch (IOException e) {
    throw new RuntimeException(e);
}

def HelloName = props['GreetingName']

if (HelloName) {
    println "Hello: " + HelloName
}
else {
println "Hello World!"
}
println "Processing completed."
```

3.3 Update the XML definition files to specify a newer version of the plugin

The following modifications made to **info.xml**, the addition is highlighted in blue.

```
<?xml version='1.0' encoding='UTF-8'?>
<plugininfo>
<author name='UrbanCode'>
<organization>UrbanCode</organization>
<email>info@urbancode.com</email>
<website>http://urbancode.com</website>
</author>
<integration type='Deploy' />
<tool-description>HelloWorld is a sample plugin</tool-description>
<release-version>1.0</release-version>
<release-notes>
<release-note plugin-version='2'>Extended to include a property</release-note>
<release-note plugin-version='1'>Initial release</release-note>
</release-notes>
</plugininfo>
```

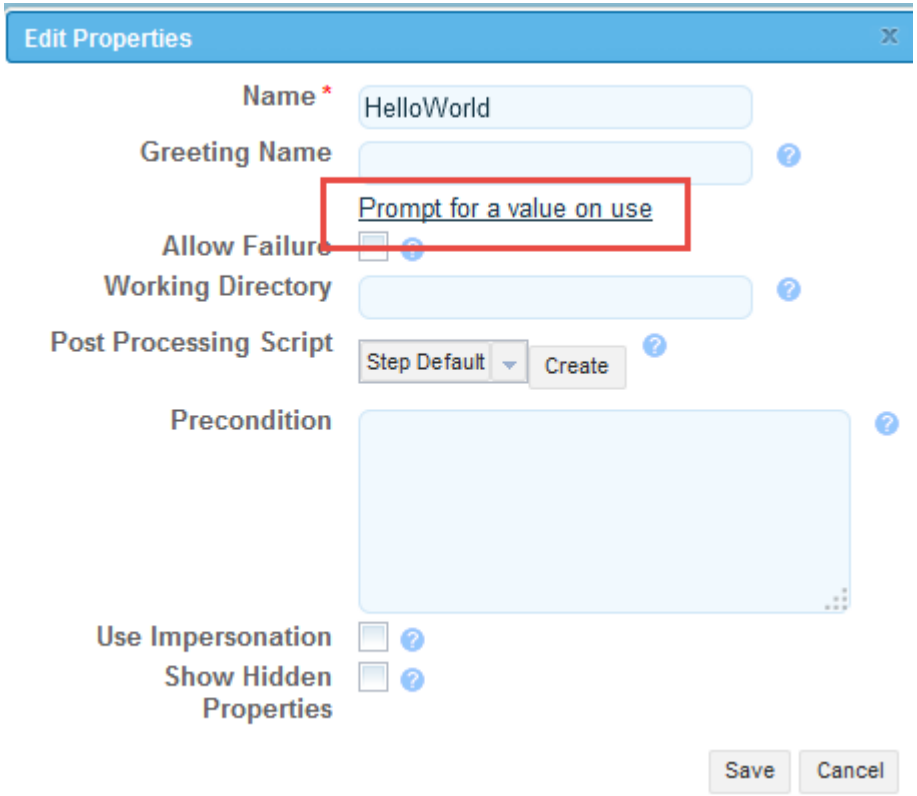
3.4 Load the new version of the plugin to the SDA server

Follow the steps described in section 2.6

3.5 Update the existing process to utilise the newer plugin

Click on Management | Processes | Hello World, then click on the design tab and remove the existing Hello World(v. 1) step. Click on OK to confirm.

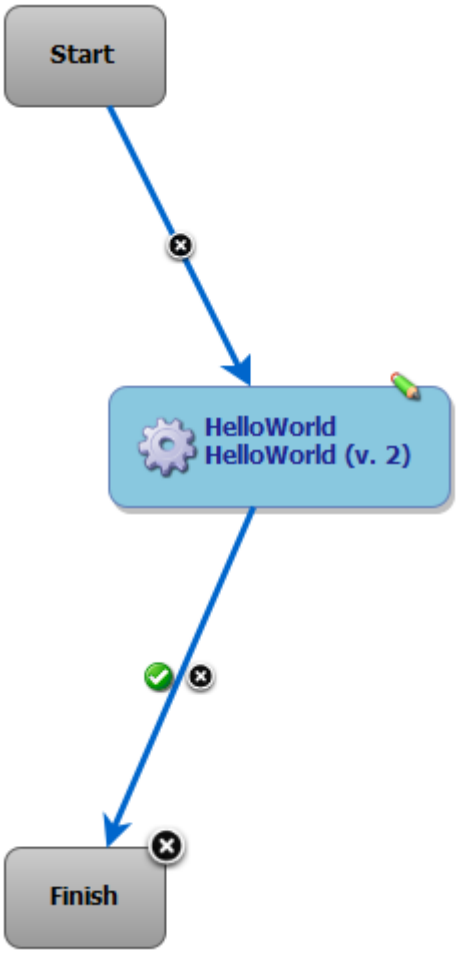
On the left-hand side under **Available Plugin Steps** select **System Utility | Hello World | Helloworld** and add it as a process step to the process, click on **Prompt for a value on use** to ensure that the user is prompted for the property value when calling the process.



Note the Greeting Name definition now appears as: Greeting Name Set a value here

Click on **Save** to store the new step.

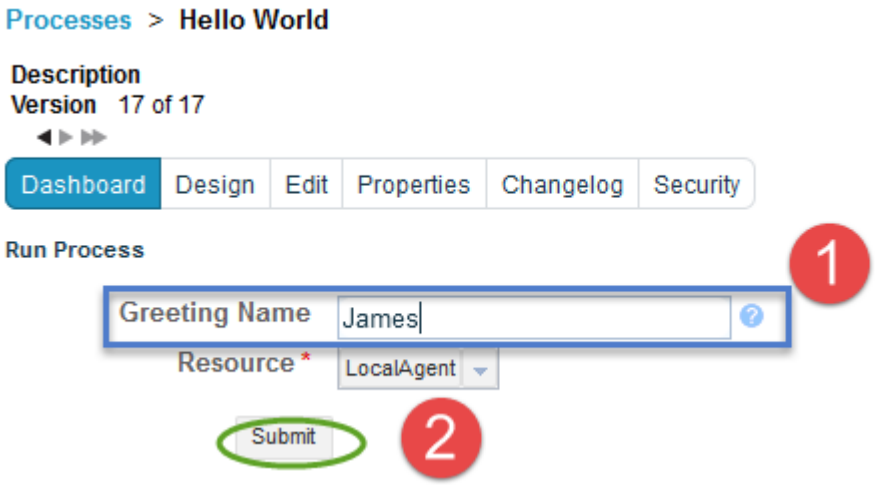
Notice version in the step is **HelloWorld (v .2)**, reflecting the updated version of the plugin.



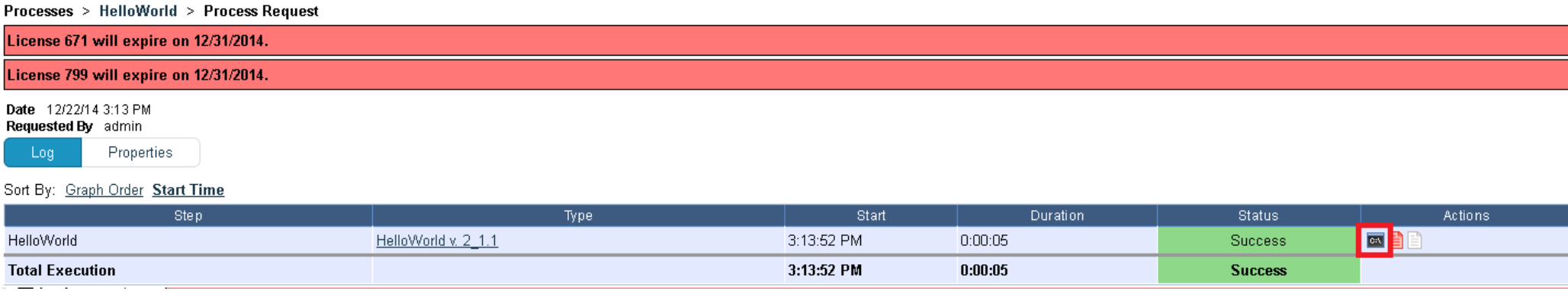
Click on the disc icon  to store the new process flow.

3.6 Run and test the new process

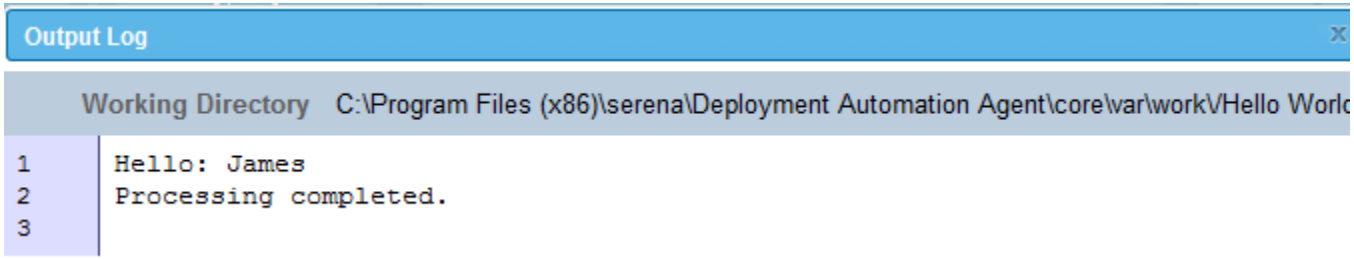
Select the **Dashboard** tab ensure a resource is selected, specify a **Greeting Name** then click on the **Submit** button.



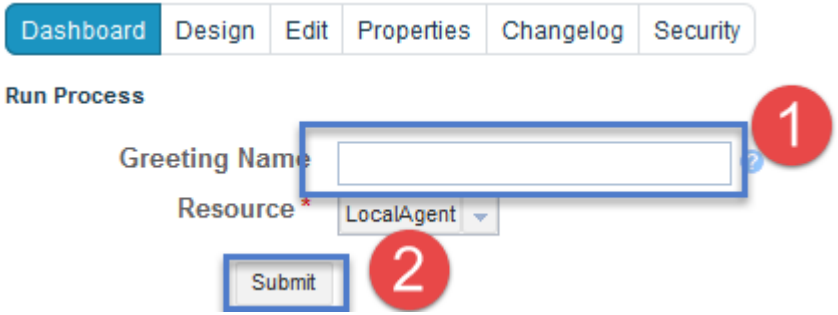
Once the process has completed click on **the console icon** (highlighted below)



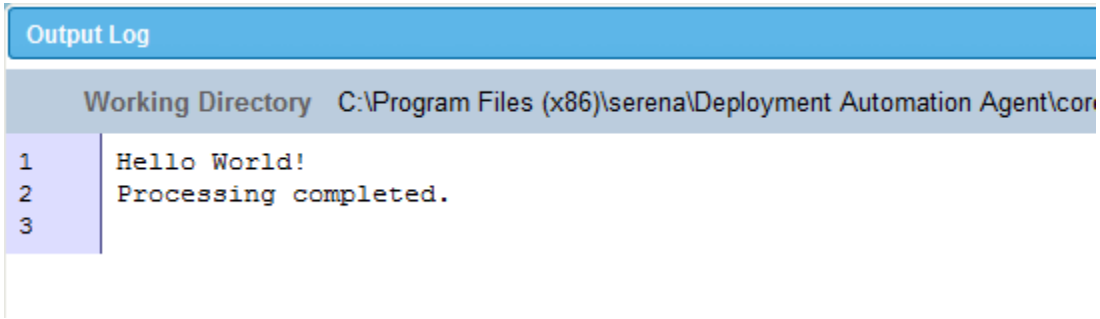
The following output will be generated



If the process is re-submitted then the Groovy script logic[1] will default to Hello World, as it detects a NULL value for the string.



Yields



[1]
```
if (HelloName) {
    println "Hello: " + HelloName
}
else {
println "Hello World!"
```

4.0 Further reference: Introduction to Groovy

In order better understand how plugins work and to be able to debug them some familiarity with the Groovy scripting language is required. Groovy itself is a scripting language similar to Java script and requires a JDK to be in place.

4.1 Groovy quick start

Use the following steps

1. Download Groovy from here: http://dl.bintray.com/groovy/maven/groovy-binary-2.3.9.zip[2]
2. Unpack the zip to a location on disc. E.g. C:\Program Files (x86)
3. Start a DOS prompt
4. Set the GROOVY_HOME variable to point to the root of the Groovy installation
   e.g. `set GROOVY_HOME=C:\Program Files (x86)\groovy-2.3.9\`
5. Install the JDK if one already does not exist and point JAVA_HOME to it
   e.g. `set JAVA_HOME=C:\Program Files\Serena\common\jre\7.0`
6. Add the Groovy bin location to the current PATH
   e.g. set PATH=%PATH%;C:\Program Files (x86)\groovy-2.3.9\bin
7. Now test by running the groovy shell, simply type: groovysh, which should return the following

```
C:\Program Files\Serena\common\jre\7.0>groovysh
Groovy Shell (2.3.9, JVM: 1.7.0_51)
Type ':help' or ':h' for help.
-------------------------------------------------------------------------------------------------------------------
groovy:000>
```

8. Run a simple program by typing: print “hello” followed by return, will produce the following output

```
groovy:000> print "hello"

hello===> null

groovy:000>
```

   To exit the shell simply type :exit

[2] Note: the current official version is 2.3.9 go to http://groovy.codehaus.org/Download confirm the latest official version

4.2 Groovy getting started guide

This can be located at http://groovy.codehaus.org/Getting+Started+Guide, and can be used for learning the Groovy scripting language as well as understanding how an existing plugin code can be correctly understood.

4.3 Locating the code for an existing plugin

The SDA server will hold all copies of the installed plugin ZIP files , these will be stored within the %SDA_ROOT%\serena\ra\var\plugins\command\repo directory. For example on a Windows this could be

C:\Users\Administrator\.serena\ra\var\plugins\command\repo

For example to interrogate the Shell plugin simply copy the file **com.urbancode.air.plugin.Shell_4.zip**[3] to another location and open the zip file and in the zip file open the groovy file in a text editor.

[3] This may have a slightly different name on different versions of SDA, version referenced here is SDA5.1.4