Genesys Intelligent Automation Help

Script Block

4/30/2025

Contents

Script Block

> **Important**
> This page is only applicable to users with the role **Application Designer**.

Script blocks are great for adding ‘presentation’ logic to your app - comparing values and branching out to different blocks, performing date calculations, combining prompts. It’s possible to invoke RESTful web services directly from a Script block, and this may be sufficient if the services are simple. It is advised not to use the Script block for tasks like handling security certificates, or calling onto databases or SOAP-based web services.

Integration Hub is a simple and powerful way to do this, and brings many other benefits such as support for multiple environments’ endpoints (dev/test/production, for example) and the ability to create automated test scripts. Using Integration Hub is a good way to keep your Intelligent Automation apps and their assorted presentation logic separate from the details of how to call onto your enterprise’s backend systems.

You can use Script blocks to perform complex operations, such as loops and if clauses, and define their own methods.

Adding this block to the callflow

To add and configure **Script** blocks in a callflow:

1. Drag and drop a **Script** block onto the appropriate position in the callflow.
2. Click the **Script** block to view its properties.
3. In the **Script** tab, select a script type and enter the script in the text box. Both script types are based on Groovy Script.
    - **Easy script** - Click **Add Entry** to add an entry to the script in a simplified interface. This script always returns **success**.
        - **Variable** - Specify a variable name.
        - **Function** - Select whether to specify a value for this variable, or determine the value based on another variable.
        - **Value** - If you selected the **Set to Value** function, specify a value. If you selected the **Set to Variable** function, the value of the variable you set in the **Variable** field is used.
        - **Attach to call** - Enable this option to attach this variable data to the call, which also means it becomes available to agents in CTI Viewer.

- **Remember** - Store this variable data in the database.

- **Complex script** - Allows advanced users to use Groovy Script to perform more complex operations, such as loops, *if* clauses, and define their own methods. The value returned by the script is used to select the next path. In most cases, the block returns **success**, but you can also use **Script** blocks for callflow branching or to trigger a global event handler such as **agent** or **recognition failure**.

4. (Optional) In the **Unit Test** tab, you can run tests on **Complex scripts** you have configured. For example, if your script defines a method to perform certain operations, and you want to ensure the results are correct, you can define a unit test for the specific method as follows:
   a. Call a defined method into the script with known set values.
   b. Get the results from the method.
   c. Compare the expected results with the results calculated in the method.
5. Click **Update**.

To learn how to use the Script block to add custom rich media, refer to the Rich Media page in this manual.

Scripting methods

Click the link below to download a zip file that documents the scripting methods that you can call from the **Script** block.

- Script API Reference

Example

```
    //
    //  Call the Web Service (note that session variables can be included in the URL and we
then use context.expandVariables() to replace the variable with whatever is in session at
runtime)
    //
    def iTimeout = 5000;
    def sURL = "https://testwebservice.com/[var:WebServiceName]"

    sURL = context.expandVariables(sURL);
    def dataResult = context.getRemoteHttpData(sURL, "POST", params, iTimeout);

    def xml = dataResult.responseXml;


    //
    //  Parse the response
    //
    assert null != xml.status;
    assert "" != xml.status.text();

        def sStatus = xml.status.text();

    for (variableDeclaration in xml.variables?.variable)
    {
        def sName = variableDeclaration.@name;
```

```
        def sValue = variableDeclaration.@value;

        context.setVariable(sName, sValue);
    }

    if ("success" == sStatus)
    {
        context.log("Found");
        return "success";
    }
    else if ("not found" == sStatus)
    {
        context.log("Not found");

        return "not found";
    }
```

Unit Tests

Sample Script

```
if (context.getVariable(“AuthMethod”) == “AccountNumber”)
{
      return “account number”;
}
else
{
      return “social security number”;
}
```

Sample Unit Test

```
context.setVariable(“AuthMethod”, “AccountNumber”);
def result = callScript();
assert “account number” == result;

context.log(“Passed test”);
```

Splitting results based on recognition failures

To handle different behaviors in case of a recognition failure, you can use a script to split the results based on `maxnomatches` and `maxnoinputs`.

```
if (context.getLastResultDetail().contains("maxnomatches"))
{ return "max retries"
}
else {
return "max timeouts"
}
```