Using FitNesse to Test Services

Version 2.0, 16th April 2019

What is FitNesse

FitNesse is a wiki-based test framework that has been around for about 15 years. It was developed by 'Uncle Bob' Martin with his son Micah. It is an evolution of an earlier non-wiki-based test framework called 'Fit'. More details here.

Essentially you develop a (lightweight) Java application (C++ and python can also be used) that interfaces between the wiki and the application you want to test.

Getting Started

Download the standalone jar file and run 'java –jar jarfile'. It accepts a number of parameters (including –p for port number), most of which have default values (more details). It has a built in web server which runs by default on port 80. Once it's up and running you can access the wiki locally with 'http://localhost'. You can ignore the occasional 'concurrency' exceptions in the console – known issues that don't impact functionality as far as I know.

If tests are likely to return a lot of data it would be advisable to allocate more memory than the default by using the **-Xmx** jvm parameter. A value of 4096 would be a sensible value to run a full production-level test suite:

```
java –Xmx4096M –jar fitnesse-standalone.jar
```

The first time you run FitNesse it creates a folder called FitNesseRoot, into which it will put some default wiki pages (Front page, user Guide, the acceptance test suite for FitNesse itself, etc.).

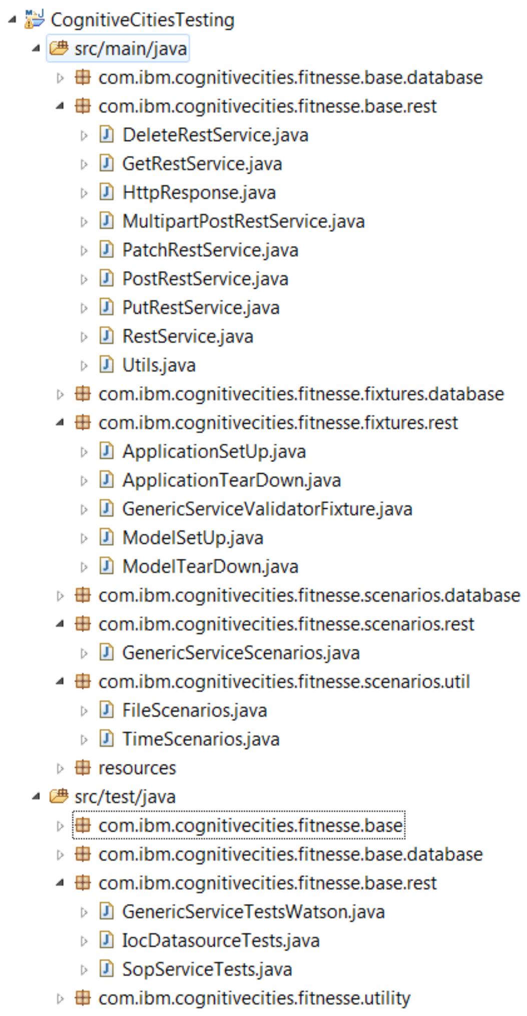
Any wiki pages you create are stored as text files and subfolders below the root folder 'FitNesseRoot'. That means you can migrate your test pages to another instance of FitNesse by simply copying the relevant files and folders from 'FitNesseRoot/FitNesse' to the target machine and the tests should run straight away.

Java Framework

The Java framework will be lightweight and is only used to interface with the application under test. All tests will be written in the wiki itself. Although the initial version of the framework should be sufficient to build most test cases it will likely evolve over time. The goal is to reach a stage where the Java framework has matured to the point where it rarely, if ever, needs to be updated at all.

The wiki test suite will include a scenario library which should contain all the scenarios, or building blocks, necessary to create useful test cases. It will also contain references to what FitNesse refers to as 'fixtures'. These are used to build decision tables, perhaps the most common way to implement a test case. The other very useful table type is a 'script' table. This can be used to assemble a number of scenarios into a sequence that simulates a full use case. The scenarios and fixtures, details of which are given later, are implemented in the Java framework.

The Java framework and wiki test suites will likely be under version control in RTC/GitHub. The current structure of the project is shown below. Fixtures and scenarios are (currently) either Rest or Databases services. In line with best practice, base functionality in the framework should have good junit test coverage (helpful for debugging issues if nothing else):



```
CognitiveCitiesTesting
  src/main/java
    com.ibm.cognitivecities.fitnesse.base.database
    com.ibm.cognitivecities.fitnesse.base.rest
      DeleteRestService.java
      GetRestService.java
      HttpResponse.java
      MultipartPostRestService.java
      PatchRestService.java
      PostRestService.java
      PutRestService.java
      RestService.java
      Utils.java
    com.ibm.cognitivecities.fitnesse.fixtures.database
    com.ibm.cognitivecities.fitnesse.fixtures.rest
      ApplicationSetUp.java
      ApplicationTearDown.java
      GenericServiceValidatorFixture.java
      ModelSetUp.java
      ModelTearDown.java
    com.ibm.cognitivecities.fitnesse.scenarios.database
    com.ibm.cognitivecities.fitnesse.scenarios.rest
      GenericServiceScenarios.java
    com.ibm.cognitivecities.fitnesse.scenarios.util
      FileScenarios.java
      TimeScenarios.java
    resources
  src/test/java
    com.ibm.cognitivecities.fitnesse.base
    com.ibm.cognitivecities.fitnesse.base.database
    com.ibm.cognitivecities.fitnesse.base.rest
      GenericServiceTestsWatson.java
      IocDatasourceTests.java
      SopServiceTests.java
    com.ibm.cognitivecities.fitnesse.utility
```

To facilitate the junit tests some key properties are read statically from a property file /fitnesse/testfiles/Test.properties in a base test class (TestBase):

```
webHost=dubperfwow2-web.mul.ie.ibm.com
webUser=sysadmin
webPassword=us3rpa88
dbHost=dubperfwow2-db.mul.ie.ibm.com
dbUser=db2i1own
dbPassword=us3rpa88
appHost=dubperfwow2-app.mul.ie.ibm.com
appHostUser=root
appHostKeyFile=/fitnesse/testfiles/windows_laptop_brendan.ppk
appHostKeyPassword=
```

Offline Tests

FitNesse can be invoked from the command line to run tests or test suites. Fitnesse will start up, run the test or suite of tests and then shut down. In other words, the test suites could be run, for example, from Jenkins. The output format of the offline tests can be junit, text, html or xml. Junit is likely to be the most suitable for our purposes. An example of such a command would be:

```
java -jar fitnesse-standalone.jar -d "/fitnesse" -p 8080 -o -b TestSuite_Results.xml -c
"Fitnesse.CognitiveCitiesSuiteofSuites.TestSuites.IocSuite.IocServicesSuite?suite&format=junit"
```

Page Types

There are four main page types:

Static Page

Used, for example, to display a contents list

Test Page

Used for an individual test or a set of tests. A 'Test' button is displayed at the top of the page to run the test.

Suite Page

Used for a suite of tests, or a suite of suites. A 'Suite' button is displayed at the top of the page to run the suite.

Special Pages

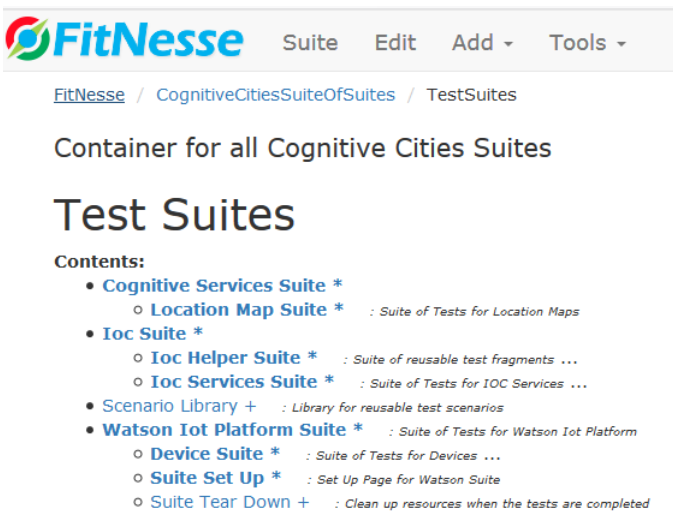
These are documented in more detail here. The main ones of interest are shown below. These are set up as regular test pages with special names.

- *ScenarioLibrary*: library of reusable scenarios, the building blocks of test cases
- *SuiteSetUp*: Set up page for suites (e.g., create database connections, set global variables like 'host name'). The contents of this page are accessible to all child suites and tests. The suite set up page will be invoked once at the start of all suites at and below its level. *If more than one SuiteSetUp page is configured in hierarchy of suites and tests only the innermost page is executed.* We can use the !include page functionality if we need to include code from a parent SuiteSetUp page (see example below)
- *SuiteTearDown*: Clean up page for suites (e.g., close database connections). The suite tear down page will be invoked once at the end of all suites at and below its level. *If more than one SuiteTearDown page is configured in hierarchy of suites and tests only the innermost page is executed.* We can use the !include page functionality if we need to include code from a parent SuiteTearDown page
- *SetUp*: Set up page for tests (e.g., set up model-specific parameters). This will be executed before all tests at the same level and below. E.g., if you have 3 test cases at and below the level of the SetUp page the scripts in the SetUp page will be executed three times. If multiple

SetUp pages are defined the innermost SetUp page takes precedence and the outer ones are not executed.

- *TearDown*: clean up page for resources allocated in SetUp. This will be executed after all tests at the same level and below. E.g., if you have 3 test cases at and below the level of the TearDown page the scripts in the TearDown page will be executed three times. If multiple TearDown pages are defined the innermost TearDown page takes precedence and the outer ones are not executed.

The hierarchy used during development of the framework is shown below:

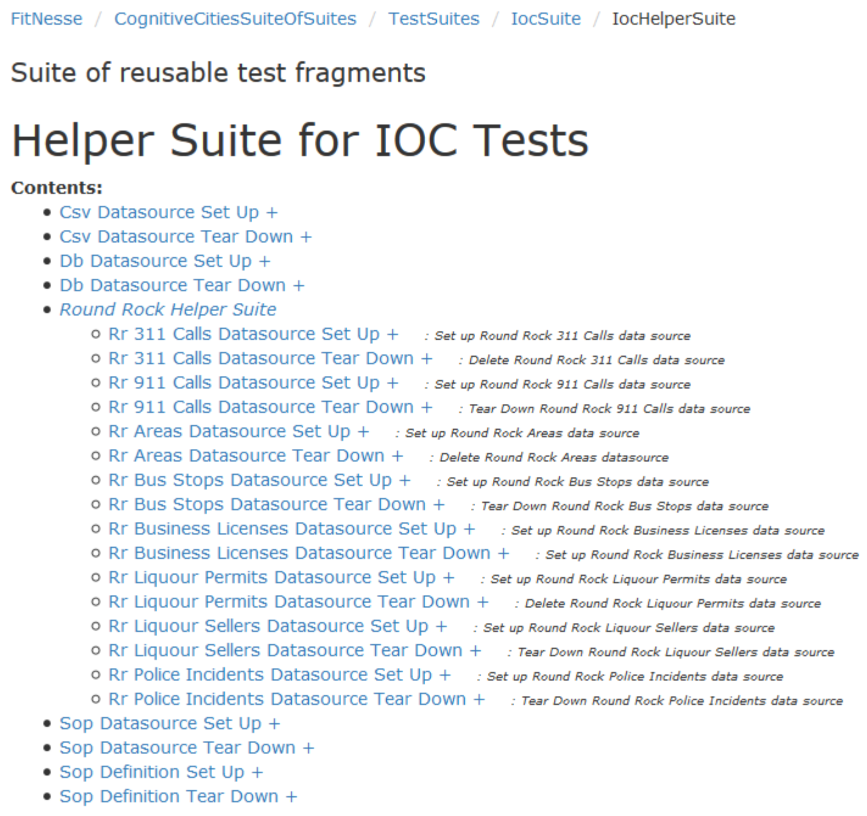


FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites

Container for all Cognitive Cities Suites

Test Suites

Contents:
- Cognitive Services Suite *
  - Location Map Suite * : Suite of Tests for Location Maps
- Ioc Suite *
  - Ioc Helper Suite * : Suite of reusable test fragments ...
  - Ioc Services Suite * : Suite of Tests for IOC Services ...
- Scenario Library + : Library for reusable test scenarios
- Watson Iot Platform Suite * : Suite of Tests for Watson Iot Platform
  - Device Suite * : Suite of Tests for Devices ...
  - Suite Set Up * : Set Up Page for Watson Suite
  - Suite Tear Down + : Clean up resources when the tests are completed

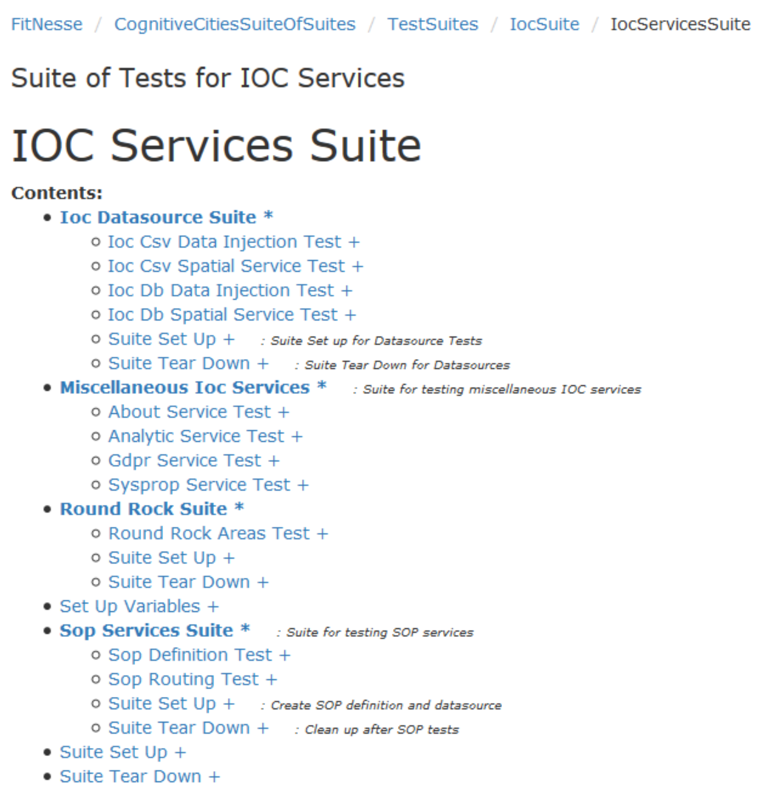


FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites / IocSuite / IocHelperSuite

Suite of reusable test fragments

Helper Suite for IOC Tests

Contents:
- Csv Datasource Set Up +
- Csv Datasource Tear Down +
- Db Datasource Set Up +
- Db Datasource Tear Down +
- *Round Rock Helper Suite*
  - Rr 311 Calls Datasource Set Up + : Set up Round Rock 311 Calls data source
  - Rr 311 Calls Datasource Tear Down + : Delete Round Rock 311 Calls data source
  - Rr 911 Calls Datasource Set Up + : Set up Round Rock 911 Calls data source
  - Rr 911 Calls Datasource Tear Down + : Tear Down Round Rock 911 Calls data source
  - Rr Areas Datasource Set Up + : Set up Round Rock Areas data source
  - Rr Areas Datasource Tear Down + : Delete Round Rock Areas datasource
  - Rr Bus Stops Datasource Set Up + : Set up Round Rock Bus Stops data source
  - Rr Bus Stops Datasource Tear Down + : Tear Down Round Rock Bus Stops data source
  - Rr Business Licenses Datasource Set Up + : Set up Round Rock Business Licenses data source
  - Rr Business Licenses Datasource Tear Down + : Set up Round Rock Business Licenses data source
  - Rr Liquour Permits Datasource Set Up + : Set up Round Rock Liquour Permits data source
  - Rr Liquour Permits Datasource Tear Down + : Delete Round Rock Liquour Permits data source
  - Rr Liquour Sellers Datasource Set Up + : Set up Round Rock Liquour Sellers data source
  - Rr Liquour Sellers Datasource Tear Down + : Tear Down Round Rock Liquour Sellers data source
  - Rr Police Incidents Datasource Set Up + : Set up Round Rock Police Incidents data source
  - Rr Police Incidents Datasource Tear Down + : Tear Down Round Rock Police Incidents data source
- Sop Datasource Set Up +
- Sop Datasource Tear Down +
- Sop Definition Set Up +
- Sop Definition Tear Down +


FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites / IocSuite / IocServicesSuite

Suite of Tests for IOC Services

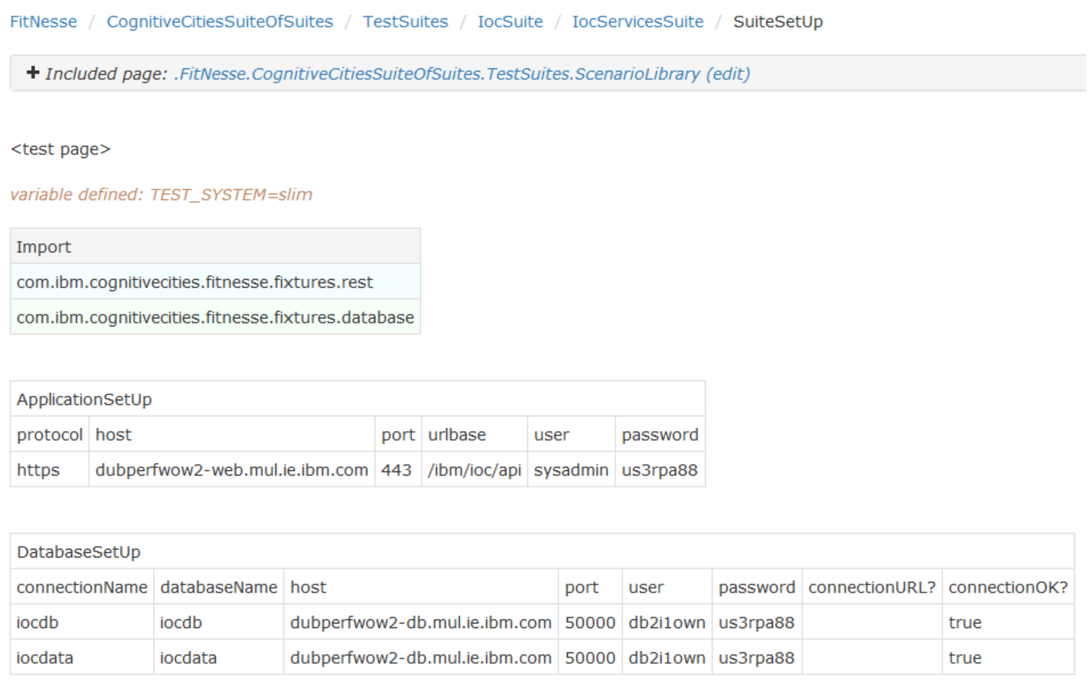
IOC Services Suite

Contents:

- **Ioc Datasource Suite ***
  - Ioc Csv Data Injection Test +
  - Ioc Csv Spatial Service Test +
  - Ioc Db Data Injection Test +
  - Ioc Db Spatial Service Test +
  - Suite Set Up + *: Suite Set up for Datasource Tests*
  - Suite Tear Down + *: Suite Tear Down for Datasources*
- **Miscellaneous Ioc Services *** *: Suite for testing miscellaneous IOC services*
  - About Service Test +
  - Analytic Service Test +
  - Gdpr Service Test +
  - Sysprop Service Test +
- **Round Rock Suite ***
  - Round Rock Areas Test +
  - Suite Set Up +
  - Suite Tear Down +
- Set Up Variables +
- **Sop Services Suite *** *: Suite for testing SOP services*
  - Sop Definition Test +
  - Sop Routing Test +
  - Suite Set Up + *: Create SOP definition and datasource*
  - Suite Tear Down + *: Clean up after SOP tests*
- Suite Set Up +
- Suite Tear Down +

SuiteSetUp/SuiteTearDown Pages

As mentioned, these are executed once at the start of every suite of tests at and below the level they are defined. A top-level SuiteSetUp page is a convenient way to initialise parameters that the Java Framework needs to execute all, or most, tests. For example, we use simple Application and Database classes to initialise parameters needed to run all Rest services and database actions:


FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites / IocSuite / IocServicesSuite / SuiteSetUp

+ *Included page: .FitNesse.CognitiveCitiesSuiteOfSuites.TestSuites.ScenarioLibrary (edit)*

<test page>

variable defined: TEST_SYSTEM=slim

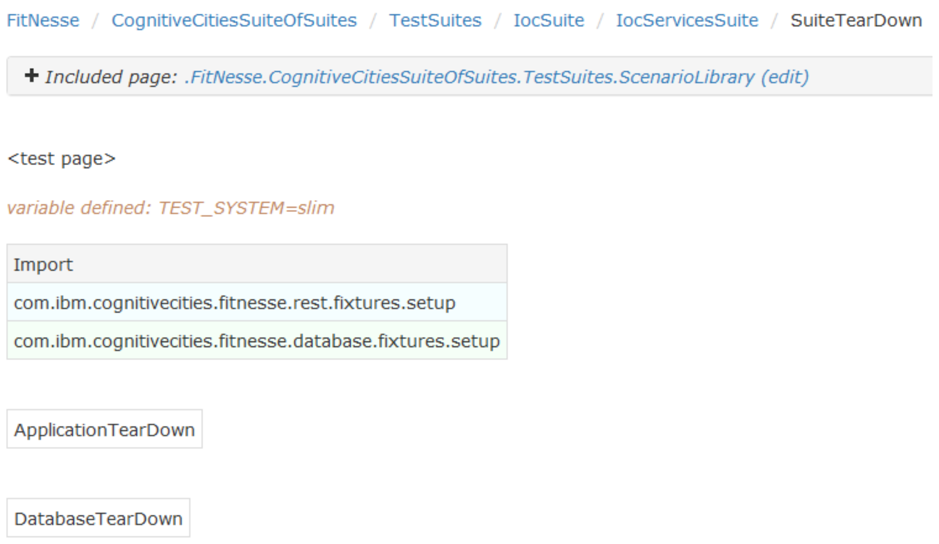
Import
com.ibm.cognitivecities.fitnesse.fixtures.rest
com.ibm.cognitivecities.fitnesse.fixtures.database

ApplicationSetUp					
protocol	host	port	urlbase	user	password
https	dubperfwow2-web.mul.ie.ibm.com	443	/ibm/ioc/api	sysadmin	us3rpa88

DatabaseSetUp							
connectionName	databaseName	host	port	user	password	connectionURL?	connectionOK?
iocdb	iocdb	dubperfwow2-db.mul.ie.ibm.com	50000	db2i1own	us3rpa88		true
iocdata	iocdata	dubperfwow2-db.mul.ie.ibm.com	50000	db2i1own	us3rpa88		true

The idea is that you will only need to update these variables once per environment. By using placeholders it should be possible to have Jenkins dynamically update these values when running the test suites after a new build is installed.

A SuiteTearDown page can then be used to clean up resources allocated in the corresponding SuiteSetUp page. In the example below we call ApplicationTearDown and DatabaseTearDown classes, the latter of which is used to close the connections opened previously:


FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites / IocSuite / IocServicesSuite / SuiteTearDown

+ *Included page: .FitNesse.CognitiveCitiesSuiteOfSuites.TestSuites.ScenarioLibrary (edit)*

<test page>

variable defined: TEST_SYSTEM=slim

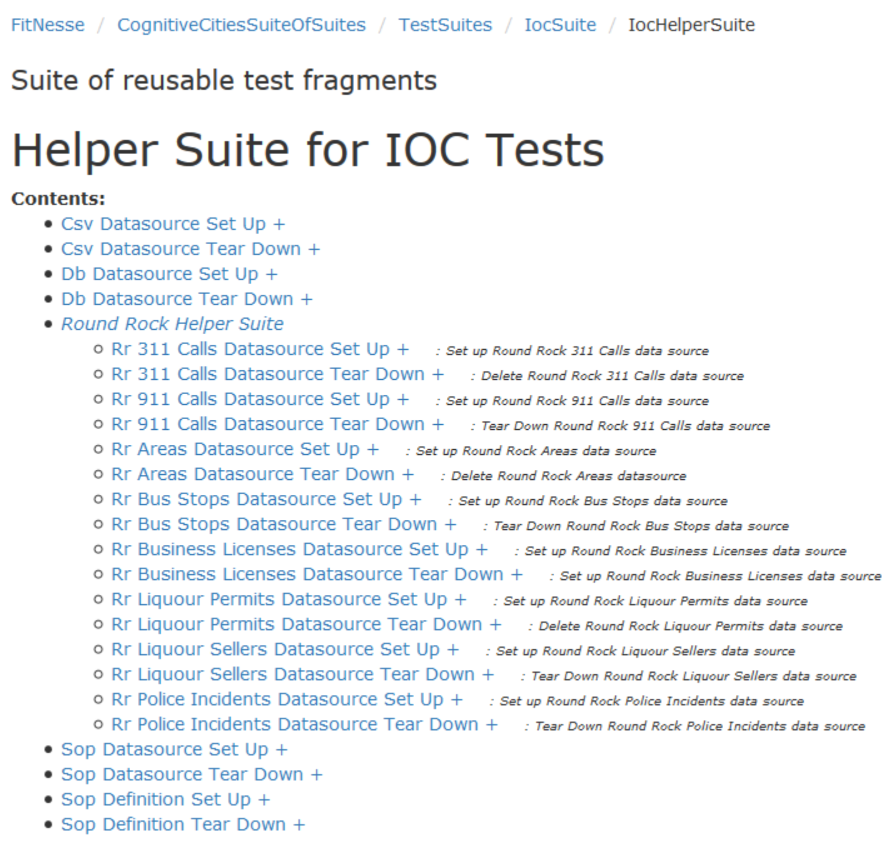
Import
com.ibm.cognitivecities.fitnesse.rest.fixtures.setup
com.ibm.cognitivecities.fitnesse.database.fixtures.setup

ApplicationTearDown

DatabaseTearDown

Hierarchical Suite Set Up/Tear Down Pages

If there is more than one Suite Set Up page in a hierarchy FitNesse will only apply the innermost one. There may be situations where we want to use Suite SetUp/TearDown pages to do special initialisations for a group of tests. For example, for IOC data source testing we might want to create one or more data sources, run a series of test cases, and then delete the data sources. We can do this in SuiteSetUp/SuiteTearDown pages. However, we will also want to inherit the initialisations from the top-level SuiteSetUp page and avoid having to duplicate code. To achieve this we can use FitNesse's include functionality to include the top-level set up/tear down pages (documentation). We can take this a stage further and define a helper suite with useful test fragments that we can include where necessary:



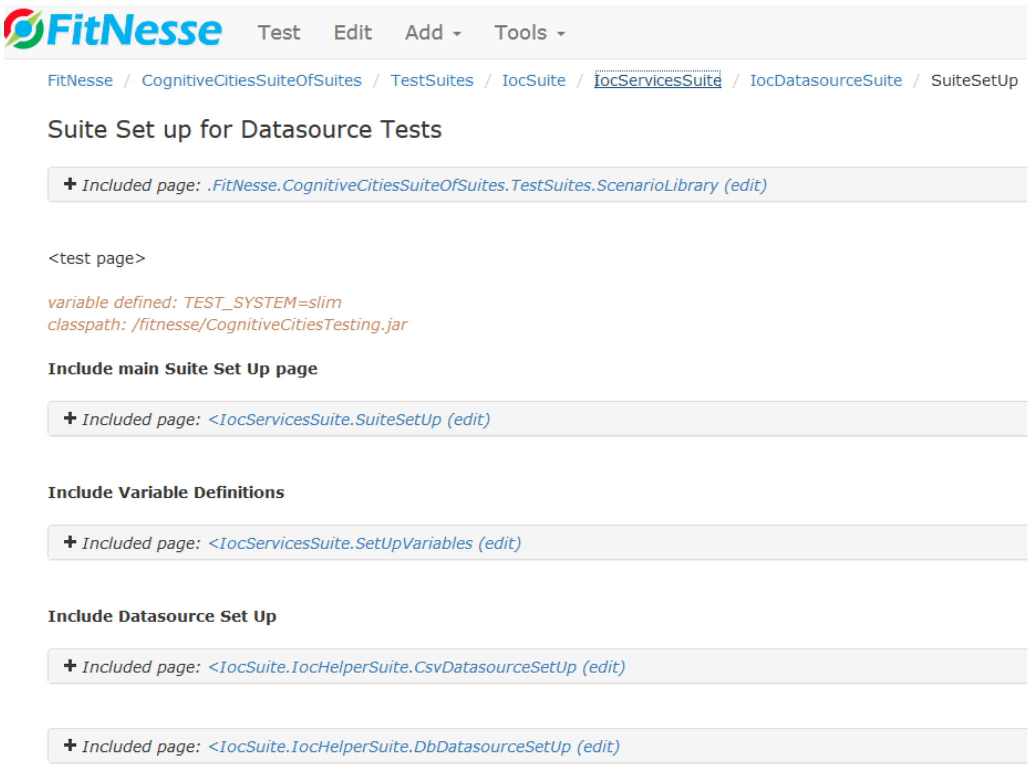
FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites / IocSuite / IocHelperSuite

Suite of reusable test fragments

Helper Suite for IOC Tests

Contents:

- Csv Datasource Set Up +
- Csv Datasource Tear Down +
- Db Datasource Set Up +
- Db Datasource Tear Down +
- *Round Rock Helper Suite*
  - Rr 311 Calls Datasource Set Up + *: Set up Round Rock 311 Calls data source*
  - Rr 311 Calls Datasource Tear Down + *: Delete Round Rock 311 Calls data source*
  - Rr 911 Calls Datasource Set Up + *: Set up Round Rock 911 Calls data source*
  - Rr 911 Calls Datasource Tear Down + *: Tear Down Round Rock 911 Calls data source*
  - Rr Areas Datasource Set Up + *: Set up Round Rock Areas data source*
  - Rr Areas Datasource Tear Down + *: Delete Round Rock Areas datasource*
  - Rr Bus Stops Datasource Set Up + *: Set up Round Rock Bus Stops data source*
  - Rr Bus Stops Datasource Tear Down + *: Tear Down Round Rock Bus Stops data source*
  - Rr Business Licenses Datasource Set Up + *: Set up Round Rock Business Licenses data source*
  - Rr Business Licenses Datasource Tear Down + *: Set up Round Rock Business Licenses data source*
  - Rr Liquour Permits Datasource Set Up + *: Set up Round Rock Liquour Permits data source*
  - Rr Liquour Permits Datasource Tear Down + *: Delete Round Rock Liquour Permits data source*
  - Rr Liquour Sellers Datasource Set Up + *: Set up Round Rock Liquour Sellers data source*
  - Rr Liquour Sellers Datasource Tear Down + *: Tear Down Round Rock Liquour Sellers data source*
  - Rr Police Incidents Datasource Set Up + *: Set up Round Rock Police Incidents data source*
  - Rr Police Incidents Datasource Tear Down + *: Tear Down Round Rock Police Incidents data source*
- Sop Datasource Set Up +
- Sop Datasource Tear Down +
- Sop Definition Set Up +
- Sop Definition Tear Down +



FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites / IocSuite / IocServicesSuite / IocDatasourceSuite / SuiteSetUp

Suite Set up for Datasource Tests

+ *Included page: .FitNesse.CognitiveCitiesSuiteOfSuites.TestSuites.ScenarioLibrary (edit)*

<test page>

variable defined: TEST_SYSTEM=slim
classpath: /fitnesse/CognitiveCitiesTesting.jar

Include main Suite Set Up page

+ *Included page: <IocServicesSuite.SuiteSetUp (edit)*

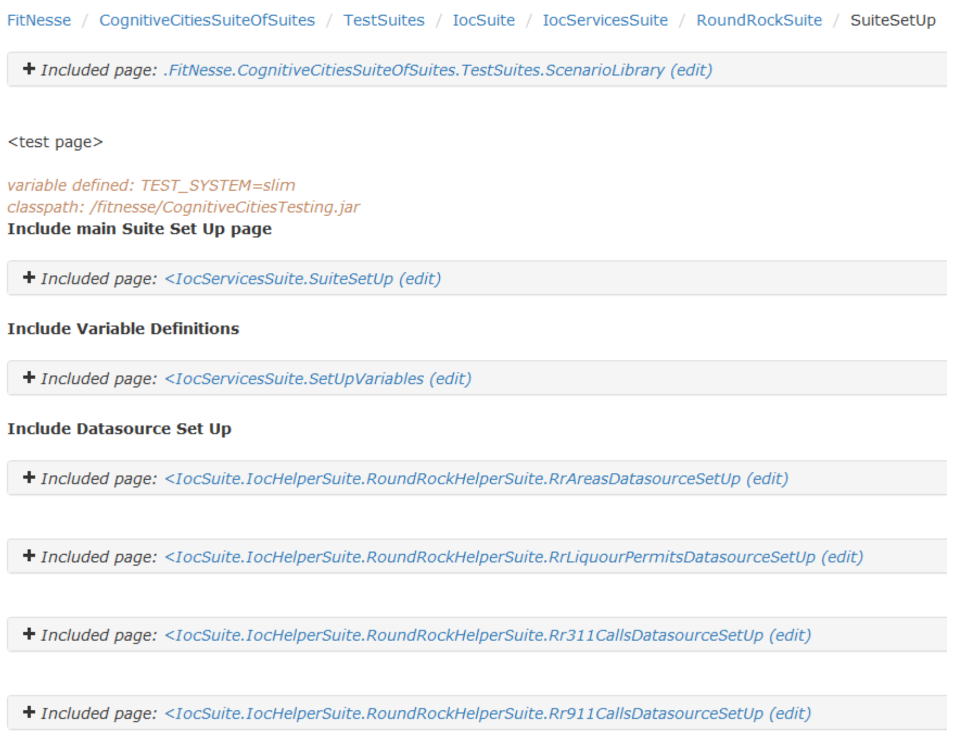
Include Variable Definitions

+ *Included page: <IocServicesSuite.SetUpVariables (edit)*

Include Datasource Set Up

+ *Included page: <IocSuite.IocHelperSuite.CsvDatasourceSetUp (edit)*

+ *Included page: <IocSuite.IocHelperSuite.DbDatasourceSetUp (edit)*


FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites / IocSuite / IocServicesSuite / RoundRockSuite / SuiteSetUp

✚ *Included page: .FitNesse.CognitiveCitiesSuiteOfSuites.TestSuites.ScenarioLibrary (edit)*

<test page>

variable defined: TEST_SYSTEM=slim
classpath: /fitnesse/CognitiveCitiesTesting.jar
Include main Suite Set Up page

✚ *Included page: <IocServicesSuite.SuiteSetUp (edit)*

Include Variable Definitions

✚ *Included page: <IocServicesSuite.SetUpVariables (edit)*

Include Datasource Set Up

✚ *Included page: <IocSuite.IocHelperSuite.RoundRockHelperSuite.RrAreasDatasourceSetUp (edit)*

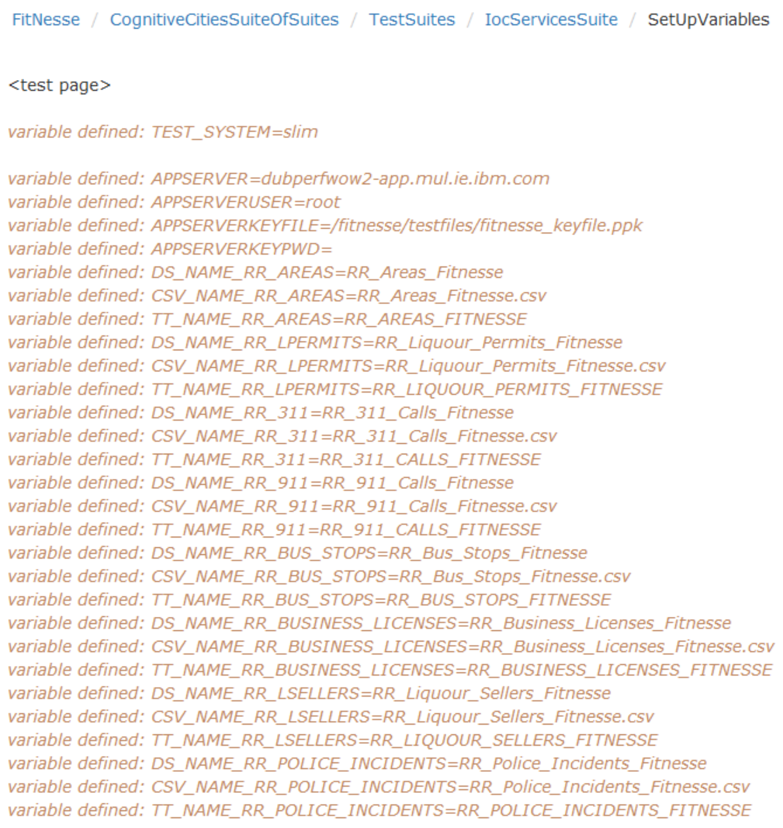
✚ *Included page: <IocSuite.IocHelperSuite.RoundRockHelperSuite.RrLiquourPermitsDatasourceSetUp (edit)*

✚ *Included page: <IocSuite.IocHelperSuite.RoundRockHelperSuite.Rr311CallsDatasourceSetUp (edit)*

✚ *Included page: <IocSuite.IocHelperSuite.RoundRockHelperSuite.Rr911CallsDatasourceSetUp (edit)*

Variable Declaration

In the above example we are also including a 'SetUpVariables' page. This is a useful technique for initialising some global variables using FitNesse's !define keyword (documentation). These kind of variables are for use within FitNesse itself, so no need to propagate them back to the Java Framework using fixtures. By using the include functionality we only need define them in one place:


FitNesse / CognitiveCitiesSuiteOfSuites / TestSuites / IocServicesSuite / SetUpVariables

<test page>

```
variable defined: TEST_SYSTEM=slim

variable defined: APPSERVER=dubperfwow2-app.mul.ie.ibm.com
variable defined: APPSERVERUSER=root
variable defined: APPSERVERKEYFILE=/fitnesse/testfiles/fitnesse_keyfile.ppk
variable defined: APPSERVERKEYPWD=
variable defined: DS_NAME_RR_AREAS=RR_Areas_Fitnesse
variable defined: CSV_NAME_RR_AREAS=RR_Areas_Fitnesse.csv
variable defined: TT_NAME_RR_AREAS=RR_AREAS_FITNESSE
variable defined: DS_NAME_RR_LPERMITS=RR_Liquour_Permits_Fitnesse
variable defined: CSV_NAME_RR_LPERMITS=RR_Liquour_Permits_Fitnesse.csv
variable defined: TT_NAME_RR_LPERMITS=RR_LIQUOUR_PERMITS_FITNESSE
variable defined: DS_NAME_RR_311=RR_311_Calls_Fitnesse
variable defined: CSV_NAME_RR_311=RR_311_Calls_Fitnesse.csv
variable defined: TT_NAME_RR_311=RR_311_CALLS_FITNESSE
variable defined: DS_NAME_RR_911=RR_911_Calls_Fitnesse
variable defined: CSV_NAME_RR_911=RR_911_Calls_Fitnesse.csv
variable defined: TT_NAME_RR_911=RR_911_CALLS_FITNESSE
variable defined: DS_NAME_RR_BUS_STOPS=RR_Bus_Stops_Fitnesse
variable defined: CSV_NAME_RR_BUS_STOPS=RR_Bus_Stops_Fitnesse.csv
variable defined: TT_NAME_RR_BUS_STOPS=RR_BUS_STOPS_FITNESSE
variable defined: DS_NAME_RR_BUSINESS_LICENSES=RR_Business_Licenses_Fitnesse
variable defined: CSV_NAME_RR_BUSINESS_LICENSES=RR_Business_Licenses_Fitnesse.csv
variable defined: TT_NAME_RR_BUSINESS_LICENSES=RR_BUSINESS_LICENSES_FITNESSE
variable defined: DS_NAME_RR_LSELLERS=RR_Liquour_Sellers_Fitnesse
variable defined: CSV_NAME_RR_LSELLERS=RR_Liquour_Sellers_Fitnesse.csv
variable defined: TT_NAME_RR_LSELLERS=RR_LIQUOUR_SELLERS_FITNESSE
variable defined: DS_NAME_RR_POLICE_INCIDENTS=RR_Police_Incidents_Fitnesse
variable defined: CSV_NAME_RR_POLICE_INCIDENTS=RR_Police_Incidents_Fitnesse.csv
variable defined: TT_NAME_RR_POLICE_INCIDENTS=RR_POLICE_INCIDENTS_FITNESSE
```

FitNesse variables are replaced inline by FitNesse at render time, for example in the file copy function shown above. Note that the documentation states that variables defined in a parent or included page are visible in the child page. That doesn't seem to be the case, so you may need to include them explicitly in child pages also.

Building a Test Case

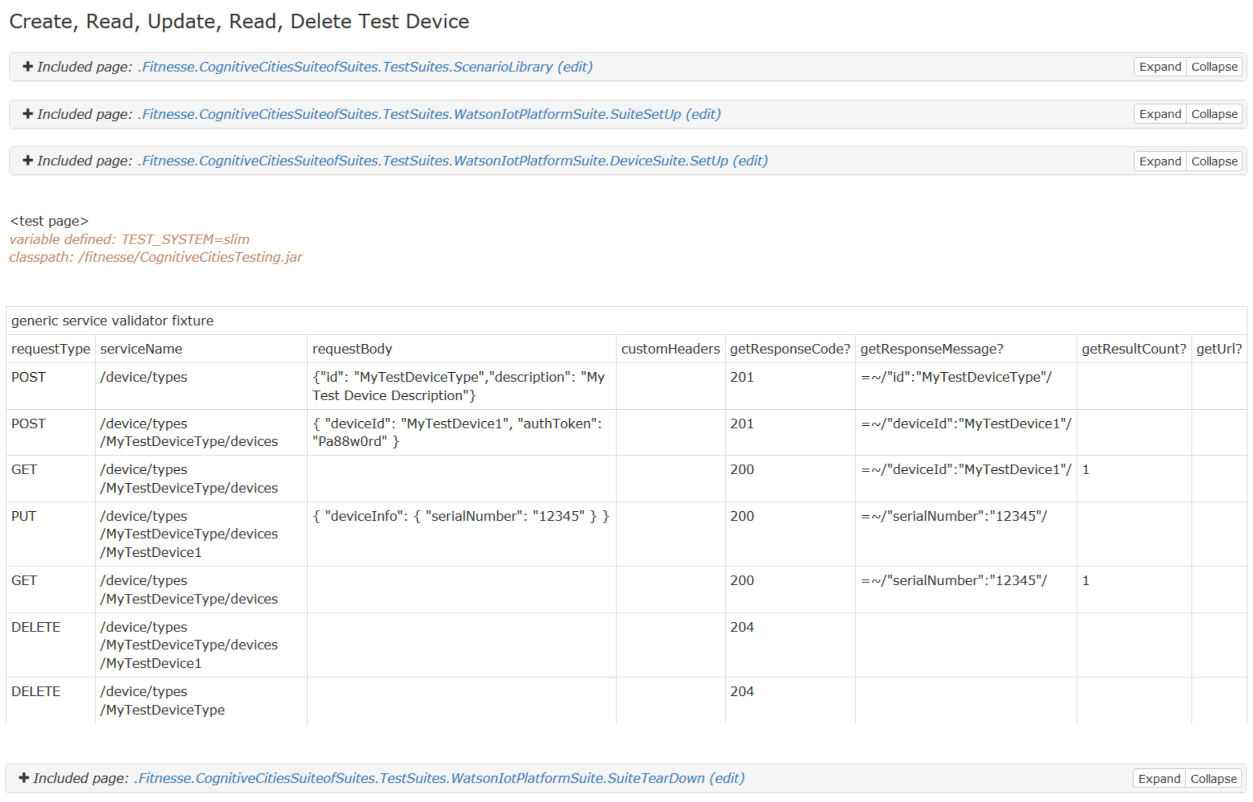
There are two test systems available in FitNesse. The more modern one is a lightweight system called Slim, and the one will be referenced and used from now on.

The two most useful table types to build test cases are decision tables (many variations available) and script tables (details of all Slim tables are here).

Decision Table

The first row of a decision table is the name of the fixture to be called (essentially a java class we implement). The columns of a decision table are a series of setters and getters (getters are indicated by a trailing ?). The setters set values in the fixture class and the getters retrieve values that are used for validation. A special method 'execute()', if defined in the fixture class, is invoked by fitnesse after the setters and before the getters, and which runs the logic of the fixture. Each row of the table represents a separate execution of the fixture. A method called reset(), if defined, is called at the start of each row to initialise variables.

Below is a typical example:



Create, Read, Update, Read, Delete Test Device

+ *Included page: .Fitnesse.CognitiveCitiesSuiteofSuites.TestSuites.ScenarioLibrary (edit)* Expand Collapse

+ *Included page: .Fitnesse.CognitiveCitiesSuiteofSuites.TestSuites.WatsonIotPlatformSuite.SuiteSetUp (edit)* Expand Collapse

+ *Included page: .Fitnesse.CognitiveCitiesSuiteofSuites.TestSuites.WatsonIotPlatformSuite.DeviceSuite.SetUp (edit)* Expand Collapse

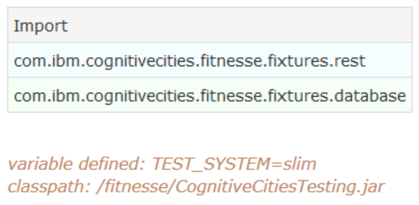
<test page>
variable defined: TEST_SYSTEM=slim
classpath: /fitnesse/CognitiveCitiesTesting.jar

generic service validator fixture							
requestType	serviceName	requestBody	customHeaders	getResponseCode?	getResponseMessage?	getResultCount?	getUrl?
POST	/device/types	{"id": "MyTestDeviceType","description": "My Test Device Description"}		201	=~/"id":"MyTestDeviceType"/		
POST	/device/types /MyTestDeviceType/devices	{ "deviceId": "MyTestDevice1", "authToken": "Pa88w0rd" }		201	=~/"deviceId":"MyTestDevice1"/		
GET	/device/types /MyTestDeviceType/devices			200	=~/"deviceId":"MyTestDevice1"/	1	
PUT	/device/types /MyTestDeviceType/devices /MyTestDevice1	{ "deviceInfo": { "serialNumber": "12345" } }		200	=~/"serialNumber":"12345"/		
GET	/device/types /MyTestDeviceType/devices			200	=~/"serialNumber":"12345"/	1	
DELETE	/device/types /MyTestDeviceType/devices /MyTestDevice1			204			
DELETE	/device/types /MyTestDeviceType			204			

+ *Included page: .Fitnesse.CognitiveCitiesSuiteofSuites.TestSuites.WatsonIotPlatformSuite.SuiteTearDown (edit)* Expand Collapse

Here we see that the special pages ScenarioLibrary, SuiteSetUp and SetUp will be included and instantiated before the test is run and SuiteTearDown will be called after the test is run.

A fixture called 'generic service validator fixture' is invoked. Fitnesse will remove the spaces, change to sentence case and then instantiate a class called GenericServiceValidatorFixture that we will have defined. The class should be in one of the packages imported in the setup pages (the 'classpath' definition on the page references the jar or jars in which the packages can be found):



Import
com.ibm.cognitivecities.fitnesse.fixtures.rest
com.ibm.cognitivecities.fitnesse.fixtures.database

variable defined: TEST_SYSTEM=slim
classpath: /fitnesse/CognitiveCitiesTesting.jar

The decision table sets the requestType, serviceName, requestBody and customHeaders (fitnesse will invoke setRequestType, setServiceName, setRequestBody and setCustomHeaders). customHeaders is a semi-colon separated list of key:value headers, for example: 'Content-Type: application/json'.

```java
public class GenericServiceValidatorFixture {
        public String serviceName;
        public String url;
        public String requestType;
        public String requestBody;
        public String customHeaders;
        HttpResponse response;
        public static String responseId = null;

        public GenericServiceValidatorFixture() {
                reset();
        }

        public void setRequestType(String type) {
                this.requestType = type;
        }

        public void setServiceName(String name) {
                this.serviceName = name;
        }

        public void setRequestBody(String body) {
                this.requestBody = body;
        }

        public void setCustomHeaders(String headers) {
                this.customHeaders = headers;
        }

        private String buildUrl() {
                return String.format(Constants.GENERIC_SERVICE_URL,
                                        ApplicationSetUp.protocol,
                                        ApplicationSetUp.host,
                                        ApplicationSetUp.port,
                                        ApplicationSetUp.urlbase,
                                        serviceName);
        }

        public int getResponseCode() {
                return response.getHttpResponseCode();
        }

        public String getUrl() {
                return url;
        }

        public String getResponseId() {
                return responseId;
        }
```

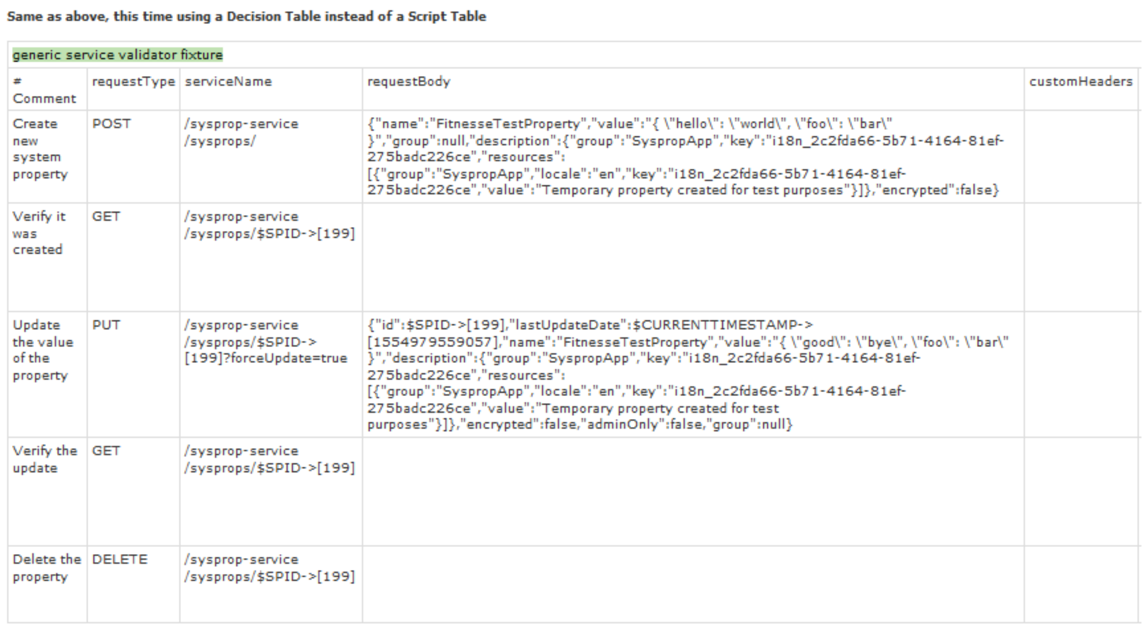
```
	public JSONObject getResponseMap() {
		return Utils.jsonAsObject(response.getHttpResponseMessage());
	}

	public String getResponseMessage() {
		return response.getHttpResponseMessage();
	}

	public int getResultCount() {
		return response.getHttpResponseObjectCount();
	}

	public HttpResponse defaultServiceResponse() {
		return new HttpResponse(500,"ERROR");
	}

	public void execute() {
		RestService restService = null;
		response = defaultServiceResponse();
		try {
			url = buildUrl();
			if (requestType.equalsIgnoreCase("GET"))
				restService = new GetRestService(customHeaders);
			else if (requestType.equalsIgnoreCase("PUT"))
				restService = new PutRestService(customHeaders);
			else if (requestType.equalsIgnoreCase("POST"))
				restService = new PostRestService(customHeaders);
			else if (requestType.equalsIgnoreCase("DELETE"))
				restService = new DeleteRestService(customHeaders);
			else if (requestType.equalsIgnoreCase("PATCH"))
				restService = new PatchRestService(customHeaders);
			else
				response.setHttpResponseMessage
					(Messages.REST_ERROR_BAD_REQUEST_TYPE);
			if (restService != null) {
				response = restService.callService(
						requestBody, url, ApplicationSetUp.user,
						ApplicationSetUp.password, true);

				if (requestType.equalsIgnoreCase("POST") ||
						requestType.equalsIgnoreCase("GET"))
					saveResponseId(customHeaders);
			}

		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public void reset() {
		requestType = "";
		requestBody = "";
		serviceName = "";
		customHeaders = "";
		response = defaultServiceResponse();
	}

	private void saveResponseId(String customHeaders) {
		String id = Utils.extractIdFromResponse(getResponseMessage(), customHeaders);
		if (id != null)
			responseId = id;
	}
}
```

The fixture (via execute()) will then call the requested Rest service. Fitnesse then calls getResponseCode(), getResponseMessage(), getResultCount() and getUrl(), and compares their return values against the contents of the cells in each row. If they match the cell is highlighted in green, otherwise they are highlighted in red. For POST services we can also capture generated IDs using getResponseId()

If the corresponding cell in the row is blank Fitnesse will just display the return value (highlighted in blue) and skip validation. In the above example we call getUrl() and fitnesse displays its value - useful for debugging if there is a failure.

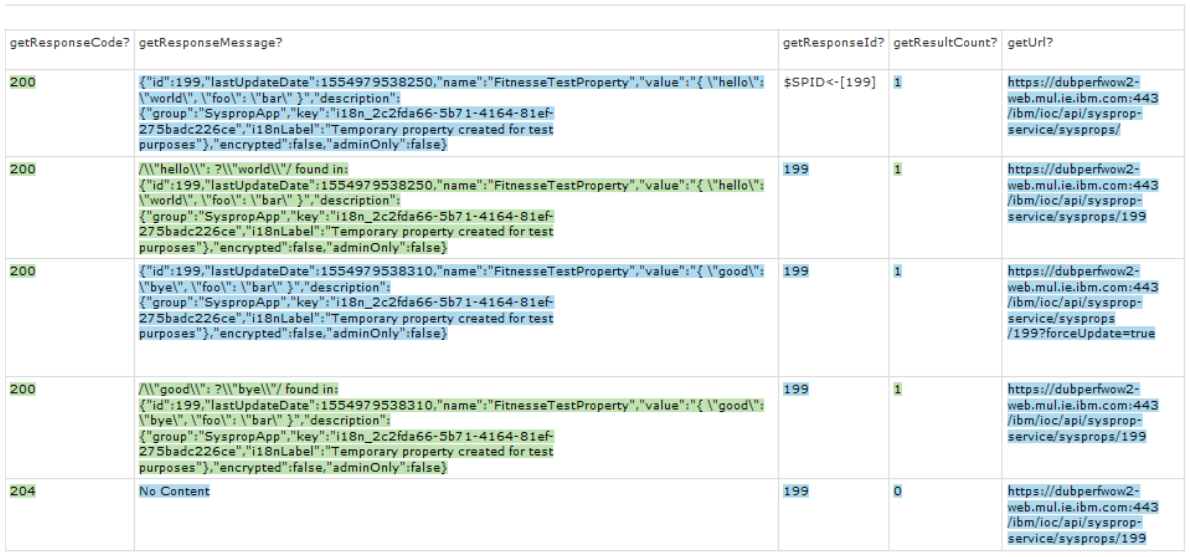
When validating a fixture invocation Fitnesse will expect an exact match between the return value and the contents of the cell. However, you can also use regular expressions if you want to match part of the return value. This is implemented with =~/{searchString}/. For numerical values you can use value comparisons (link).

An example of a fixture after successful validation:



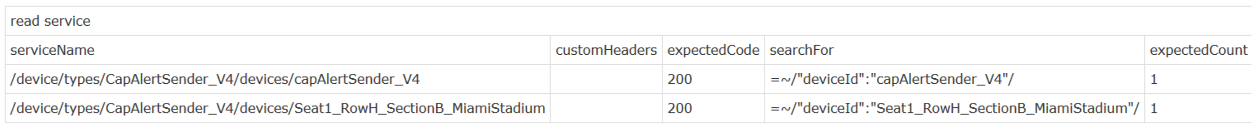
Same as above, this time using a Decision Table instead of a Script Table

generic service validator fixture				
# Comment	requestType	serviceName	requestBody	customHeaders
Create new system property	POST	/sysprop-service /sysprops/	{"name":"FitnesseTestProperty","value":"{ \"hello\": \"world\", \"foo\": \"bar\" }","group":null,"description":{"group":"SyspropApp","key":"i18n_2c2fda66-5b71-4164-81ef-275badc226ce","resources":[{"group":"SyspropApp","locale":"en","key":"i18n_2c2fda66-5b71-4164-81ef-275badc226ce","value":"Temporary property created for test purposes"}]},"encrypted":false}	
Verify it was created	GET	/sysprop-service /sysprops/$SPID->[199]		
Update the value of the property	PUT	/sysprop-service /sysprops/$SPID->[199]?forceUpdate=true	{"id":$SPID->[199],"lastUpdateDate":$CURRENTTIMESTAMP->[1554979559057],"name":"FitnesseTestProperty","value":"{ \"good\": \"bye\", \"foo\": \"bar\" }","description":{"group":"SyspropApp","key":"i18n_2c2fda66-5b71-4164-81ef-275badc226ce","resources":[{"group":"SyspropApp","locale":"en","key":"i18n_2c2fda66-5b71-4164-81ef-275badc226ce","value":"Temporary property created for test purposes"}]},"encrypted":false,"adminOnly":false,"group":null}	
Verify the update	GET	/sysprop-service /sysprops/$SPID->[199]		
Delete the property	DELETE	/sysprop-service /sysprops/$SPID->[199]		



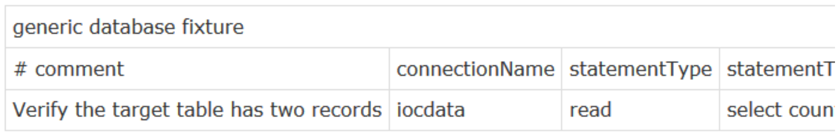
getResponseCode?	getResponseMessage?	getResponseId?	getResultCount?	getUrl?
200	{"id":199,"lastUpdateDate":1554979538250,"name":"FitnesseTestProperty","value":"{ \"hello\": \"world\", \"foo\": \"bar\" }","description":{"group":"SyspropApp","key":"i18n_2c2fda66-5b71-4164-81ef-275badc226ce","i18nLabel":"Temporary property created for test purposes"},"encrypted":false,"adminOnly":false}	$SPID<-[199]	1	https://dubperfwow2-web.mul.ie.ibm.com:443/ibm/ioc/api/sysprop-service/sysprops/
200	/\\"hello\\": ?\\"world\\"/ found in: {"id":199,"lastUpdateDate":1554979538250,"name":"FitnesseTestProperty","value":"{ \"hello\": \"world\", \"foo\": \"bar\" }","description":{"group":"SyspropApp","key":"i18n_2c2fda66-5b71-4164-81ef-275badc226ce","i18nLabel":"Temporary property created for test purposes"},"encrypted":false,"adminOnly":false}	199	1	https://dubperfwow2-web.mul.ie.ibm.com:443/ibm/ioc/api/sysprop-service/sysprops/199
200	{"id":199,"lastUpdateDate":1554979538310,"name":"FitnesseTestProperty","value":"{ \"good\": \"bye\", \"foo\": \"bar\" }","description":{"group":"SyspropApp","key":"i18n_2c2fda66-5b71-4164-81ef-275badc226ce","i18nLabel":"Temporary property created for test purposes"},"encrypted":false,"adminOnly":false}	199	1	https://dubperfwow2-web.mul.ie.ibm.com:443/ibm/ioc/api/sysprop-service/sysprops/199?forceUpdate=true
200	/\\"good\\": ?\\"bye\\"/ found in: {"id":199,"lastUpdateDate":1554979538310,"name":"FitnesseTestProperty","value":"{ \"good\": \"bye\", \"foo\": \"bar\" }","description":{"group":"SyspropApp","key":"i18n_2c2fda66-5b71-4164-81ef-275badc226ce","i18nLabel":"Temporary property created for test purposes"},"encrypted":false,"adminOnly":false}	199	1	https://dubperfwow2-web.mul.ie.ibm.com:443/ibm/ioc/api/sysprop-service/sysprops/199
204	No Content	199	0	https://dubperfwow2-web.mul.ie.ibm.com:443/ibm/ioc/api/sysprop-service/sysprops/199

A special use case for decision tables will invoke an individual scenario (from the Scenario Library) multiple times. Here we call a scenario called ‘read service’. It will invoke a generic read service with the specified service name and custom headers. It will validate that the response code matches the value under ‘expectedCode’, the response contains the text specified under ‘searchFor’ and the number of returned items matches the value under the ‘expectedCount’ header.



read service				
serviceName	customHeaders	expectedCode	searchFor	expectedCount
/device/types/CapAlertSender_V4/devices/capAlertSender_V4		200	=~/"deviceId":"capAlertSender_V4"/	1
/device/types/CapAlertSender_V4/devices/Seat1_RowH_SectionB_MiamiStadium		200	=~/"deviceId":"Seat1_RowH_SectionB_MiamiStadium"/	1

A comment column can be created anywhere in a decision table by placing # before the header. For example:



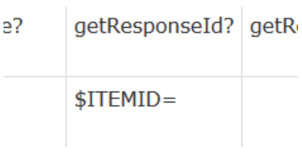
generic database fixture			
# comment	connectionName	statementType	statementT
Verify the target table has two records	iocdata	read	select coun

Values returned by getters can be stored in variables and reused at any point in a test case. This is particularly useful in the case where we want to reuse IDs generated by particular POST services. See the section ‘Capturing Generated IDs’ below for more details.

Capturing Generated IDs

Decision Tables

As mentioned above, a method called getResponseId() can be used to return IDs generated by the POST services. If a POST service returns an ‘id’ field the value is captured and stored in a static variable. The value of the variable is only overwritten when a subsequent POST service is invoked and returns a new ID value. If a POST service doesn’t return an ‘id’ field the existing value is retained and not overwritten. A READ service will also return the value of an ‘id’ field if present. For decision tables the value of the response id can be stored in a Fitnesse variable of your choosing:



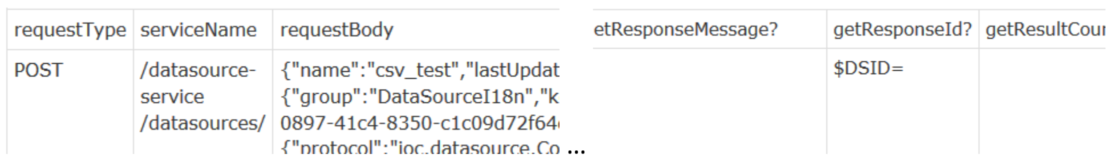
e?	getResponseId?	getR
	$ITEMID=	

Let’s say a POST service returns the following JSON response:

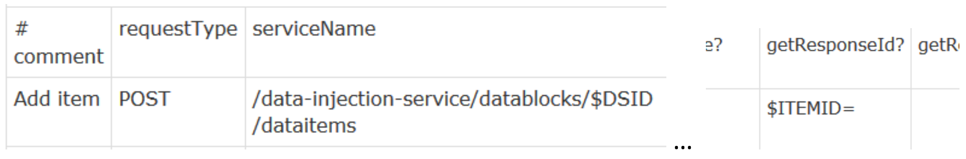
```
{
  "version":"draft",
  "created":"2019-04-05T21:12:00Z",
  "createdBy":"a-9kpzic-cfbqvnuh2h",
  "updated":"2019-04-05T21:12:00Z",
  "updatedBy":"a-9kpzic-cfbqvnuh2h",
  "name":"testSchema_LI",
  "id":"5ca7c4a0ecbfc5002863b8e4",
  "schemaType":"json-schema",
  "schemaFileName":"testSchema_LI.json",
  "contentType":"application/octet-stream",
  "refs":{
    "content":"/api/v0002/draft/schemas/5ca7c4a0ecbfc5002863b8e4/content"
  }
}
```

The value of ‘id’ here was generated by Watson. Thus ‘getResponseId()’ will return ‘5ca7c4a0ecbfc5002863b8e4’. $ITEMID can then be used in subsequent services if necessary.

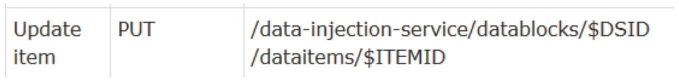
Below is an example involving IOC data sources. After creating the data source we save the returned ID as $DSID. We then call the data injection POST service to add a new item to the data source. The ID generated by that service is saved as $ITEMID. We can then use both saved IDs to update the new data source item:



requestType	serviceName	requestBody		etResponseMessage?	getResponseId?	getResultCou
POST	/datasource-service /datasources/	{"name":"csv_test","lastUpdat {"group":"DataSourceI18n","k 0897-41c4-8350-c1c09d72f64 {"protocol":"ioc.datasource.Co	…		$DSID=	



# comment	requestType	serviceName		e?	getResponseId?	getR
Add item	POST	/data-injection-service/datablocks/$DSID /dataitems	…		$ITEMID=	

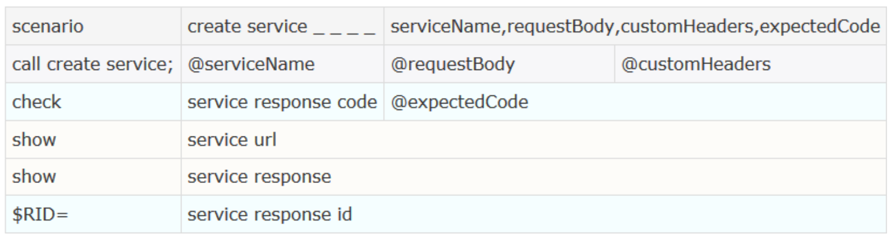


Update item	PUT	/data-injection-service/datablocks/$DSID /dataitems/$ITEMID

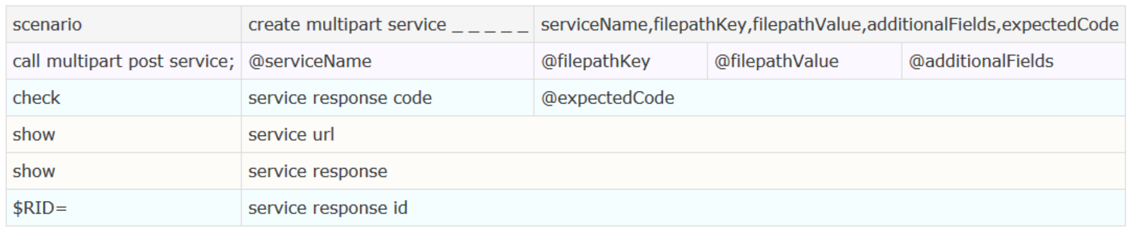
There is no practical limit to the number of variables that can be used in this way.

Script Tables

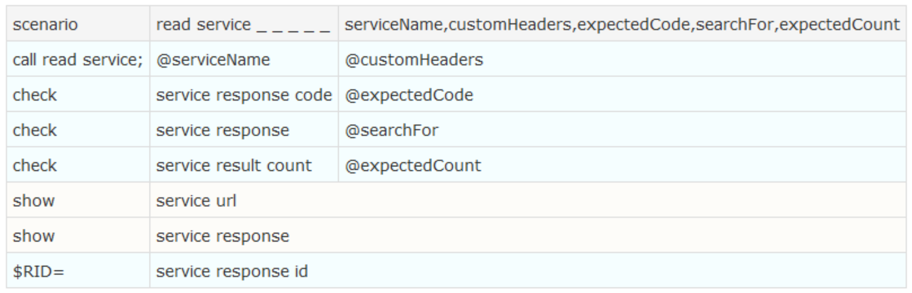
For script tables that use the scenario library the returned ID from POST and READ services is saved in a fixed variable called $RID. See below:



scenario	create service _ _ _ _	serviceName,requestBody,customHeaders,expectedCode	
call create service;	@serviceName	@requestBody	@customHeaders
check	service response code	@expectedCode	
show	service url		
show	service response		
$RID=	service response id		

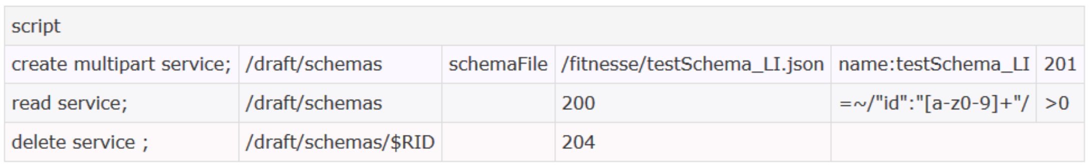


scenario	create multipart service _ _ _ _ _	serviceName,filepathKey,filepathValue,additionalFields,expectedCode		
call multipart post service;	@serviceName	@filepathKey	@filepathValue	@additionalFields
check	service response code	@expectedCode		
show	service url			
show	service response			
$RID=	service response id			



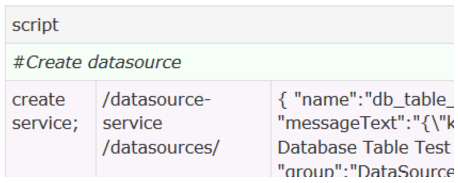
scenario	read service _ _ _ _ _	serviceName,customHeaders,expectedCode,searchFor,expectedCount
call read service;	@serviceName	@customHeaders
check	service response code	@expectedCode
check	service response	@searchFor
check	service result count	@expectedCount
show	service url	
show	service response	
$RID=	service response id	

A Watson example of how $RID might be used is shown below:



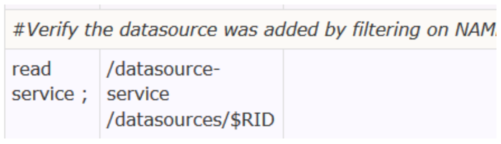
script					
create multipart service;	/draft/schemas	schemaFile	/fitnesse/testSchema_LI.json	name:testSchema_LI	201
read service;	/draft/schemas		200	=~/"id":"[a-z0-9]+"/	>0
delete service ;	/draft/schemas/$RID		204		

An IOC data source example is shown below. We first create the data source; the returned ID will be saved in the variable $RID by the library 'create service' scenario:

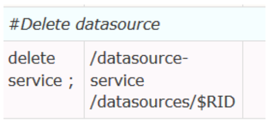


script		
#Create datasource		
create service;	/datasource-service /datasources/	{ "name":"db_table_ "messageText":"{\"k Database Table Test "group":"DataSource

Now we can use $RID to read/delete the data source:



#Verify the datasource was added by filtering on NAM		
read service ;	/datasource-service /datasources/$RID	

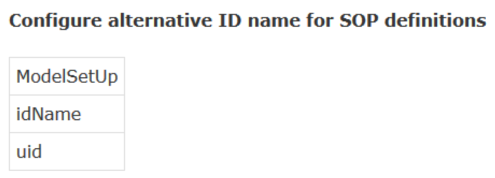


#Delete datasource		
delete service ;	/datasource-service /datasources/$RID	

The limitation with script tables is that we can't use more than one $RID at the same time. For example, a use case that creates an IOC data source and then injects a new item will generate two IDs (one for the data source and one for the item). Once the injection service has been called $RID is overwritten with its ID, so we no longer have access to the data source ID. For use cases that require more than one ID to be used at the same time you have to use decision tables, as in the earlier data source example.

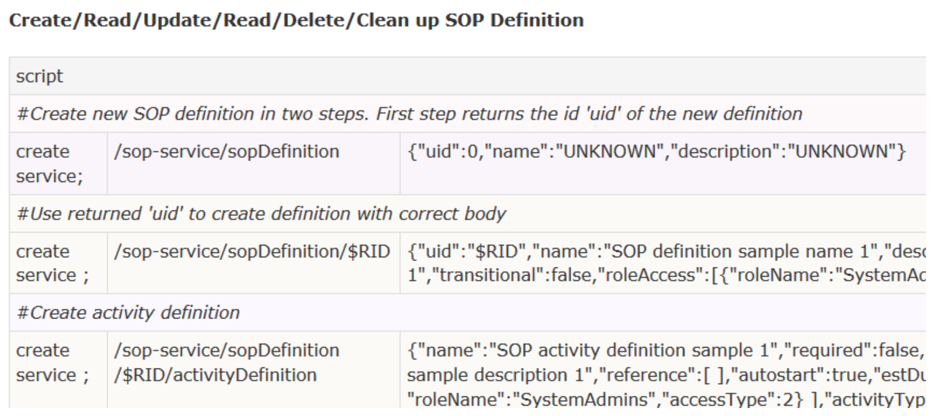
ID field with different name

By default a field name of 'id' is assumed for services that return generated IDs. However, some services might use a different field name. For example, IOC's SOP definition service uses the field name 'uid'. To handle this the fixture ModelSetUp can be used to override the default value. Calling ModelTearDown will restore the default field name 'id'. ModelSetUp/ModelTearDown can be called as part of a SetUp/TearDown or placed directly within a test case (more than once if necessary), as in this example:



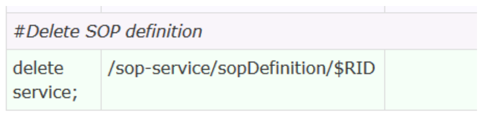
Configure alternative ID name for SOP definitions

ModelSetUp
idName
uid

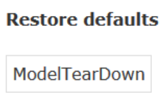


Create/Read/Update/Read/Delete/Clean up SOP Definition

script		
#Create new SOP definition in two steps. First step returns the id 'uid' of the new definition		
create service;	/sop-service/sopDefinition	{"uid":0,"name":"UNKNOWN","description":"UNKNOWN"}
#Use returned 'uid' to create definition with correct body		
create service ;	/sop-service/sopDefinition/$RID	{"uid":"$RID","name":"SOP definition sample name 1","desc 1","transitional":false,"roleAccess":[{"roleName":"SystemAc
#Create activity definition		
create service ;	/sop-service/sopDefinition /$RID/activityDefinition	{"name":"SOP activity definition sample 1","required":false, sample description 1","reference":[ ],"autostart":true,"estDu "roleName":"SystemAdmins","accessType":2}],"activityTyp



#Delete SOP definition		
delete service;	/sop-service/sopDefinition/$RID	

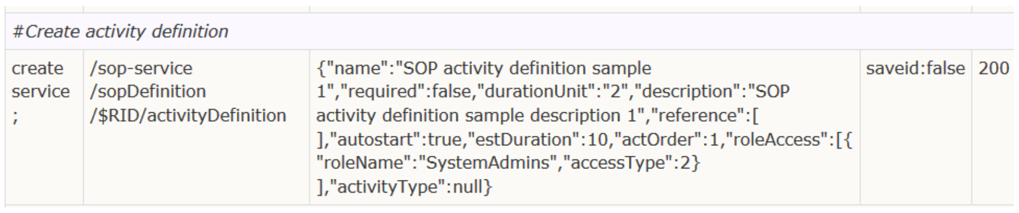


Restore defaults

ModelTearDown

Disabling the ID Save Feature

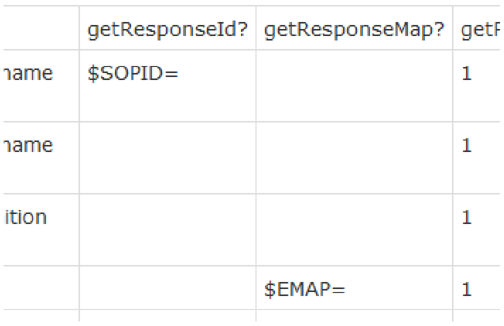
For script tables it may be useful to disable the capturing of a returned ID. This can be done by passing in a custom header 'saveid:false'. For example, the activityDefinition sop service returns a dummy 'uid' of zero that we may not want to save:



#Create activity definition				
create service ;	/sop-service /sopDefinition /$RID/activityDefinition	{"name":"SOP activity definition sample 1","required":false,"durationUnit":"2","description":"SOP activity definition sample description 1","reference":[ ],"autostart":true,"estDuration":10,"actOrder":1,"roleAccess":[{ "roleName":"SystemAdmins","accessType":2}],"activityType":null}	saveid:false	200

Full Http Response as Hash Map

For most scenarios capturing the id in a response will be sufficient. However, occasionally you may want to capture the value of an arbitrary field in a http response.
We can access field values in a Json response using Hash Maps (documentation). The custom method 'getResponseMap()' has been defined to return http and database responses as a map, which can then be saved as usual using a FitNesse variable:



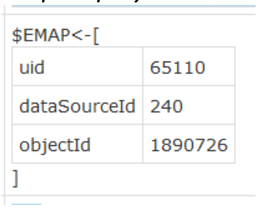
	getResponseId?	getResponseMap?	getF
ıame	$SOPID=		1
ıame			1
ition			1
		$EMAP=	1

At run time FitNesse will display as a nested table:

Http Response:

```
{
  "uid":65100,
  "objectId":1890726,
  "dataSourceId":240
}
```

Map displayed as nested table:

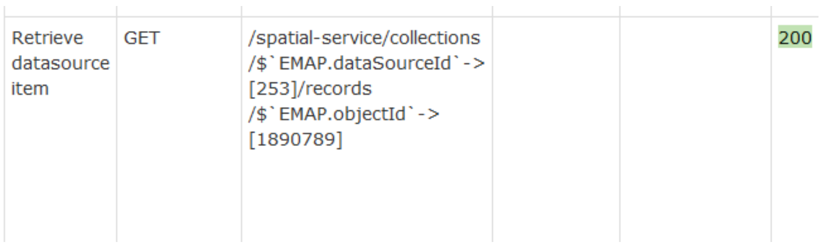


$EMAP<-[

uid	65110
dataSourceId	240
objectId	1890726

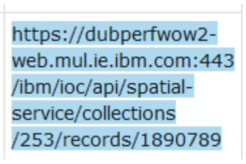
]

We can then use the back-tick operator to extract the value of any key in the map and use it in subsequent services:



Retrieve datasource item	GET	/spatial-service/collections /$`EMAP.dataSourceId`-> [253]/records /$`EMAP.objectId`-> [1890789]			200

The evaluated url:



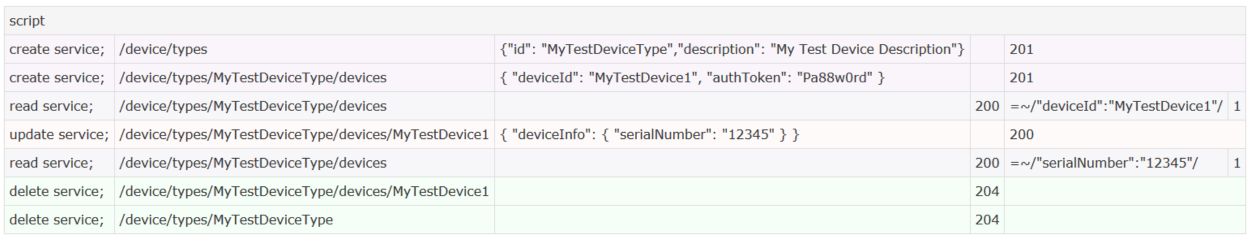
https://dubperfwow2-web.mul.ie.ibm.com:443/ibm/ioc/api/spatial-service/collections/253/records/1890789

Note that this feature can be resource intensive, so should be used only where necessary (i.e., don't always include the 'getResponseMap()' column in decision tables.

Script Table

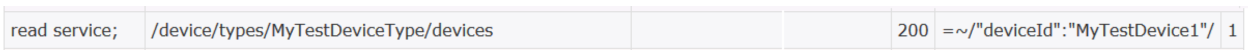
Script tables can be used to build test cases from individual scenarios. The scenarios can be a mixture of, for example, Rest and database service calls. Script tables are simpler than decision tables. However, they do have some restrictions, including the restriction mentioned in the 'Capturing Generated IDs' section above.

The first cell in each row in a script table names the scenario, followed by zero or more input parameters that are passed to the scenario routine. For example, the following (truncated) script table implements the lifecycle of a device: create, read, update, read, delete:



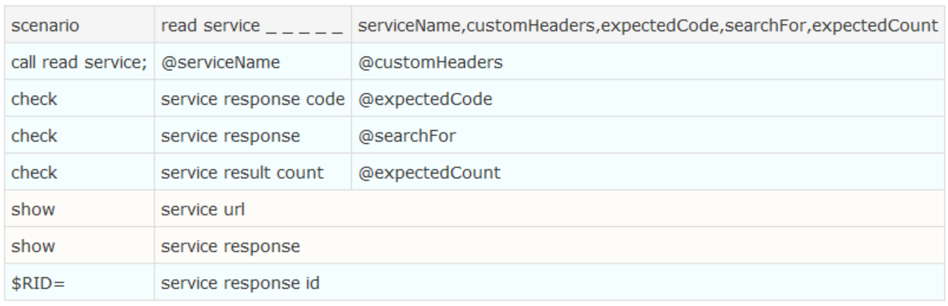
script					
create service;	/device/types	{"id": "MyTestDeviceType","description": "My Test Device Description"}		201	
create service;	/device/types/MyTestDeviceType/devices	{ "deviceId": "MyTestDevice1", "authToken": "Pa88w0rd" }		201	
read service;	/device/types/MyTestDeviceType/devices		200	=~/"deviceId":"MyTestDevice1"/	1
update service;	/device/types/MyTestDeviceType/devices/MyTestDevice1	{ "deviceInfo": { "serialNumber": "12345" } }		200	
read service;	/device/types/MyTestDeviceType/devices		200	=~/"serialNumber":"12345"/	1
delete service;	/device/types/MyTestDeviceType/devices/MyTestDevice1		204		
delete service;	/device/types/MyTestDeviceType		204		

Let’s look at the read/get service in more detail:



read service;	/device/types/MyTestDeviceType/devices			200	=~/"deviceId":"MyTestDevice1"/	1

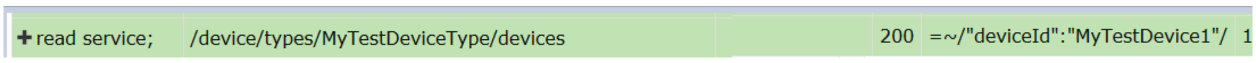
Here, we are calling the scenario ‘read service’, passing in five parameters: service name, custom headers (blank in this case), expected resonse code, search string and expected result count. The scenario looks like this:



scenario	read service _ _ _ _ _	serviceName,customHeaders,expectedCode,searchFor,expectedCount
call read service;	@serviceName	@customHeaders
check	service response code	@expectedCode
check	service response	@searchFor
check	service result count	@expectedCount
show	service url	
show	service response	
$RID=	service response id	

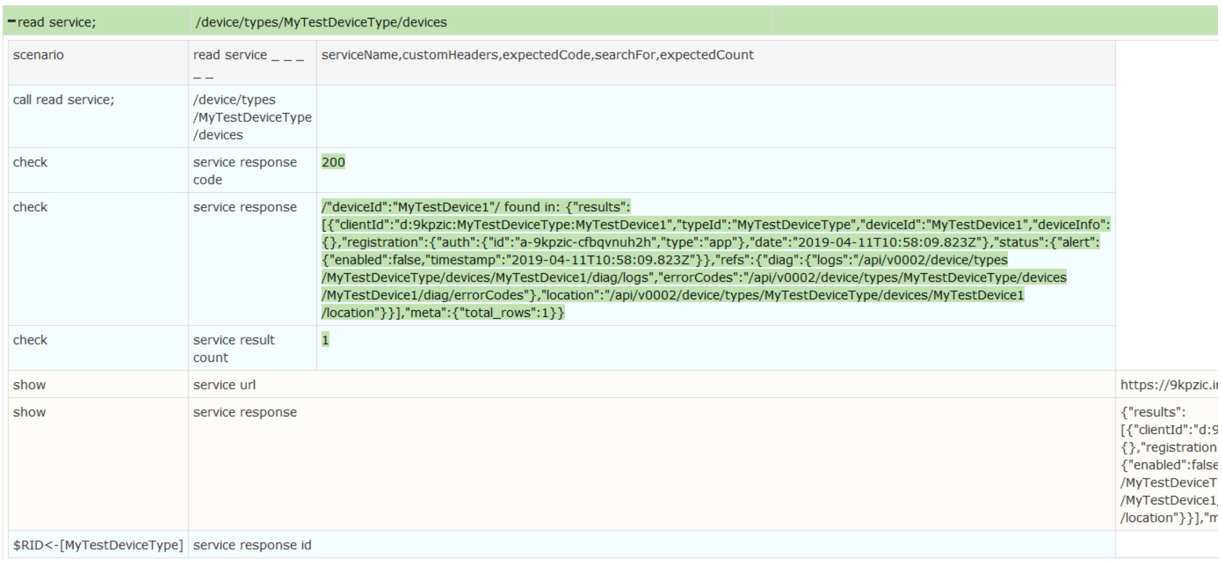
The first step in the scenario calls the method ‘callReadService’, passing in the first two parameters of the scenario (serviceName and customHeaders). The next step uses the reserved word ‘check’ to verify that the function serviceResponseCode() returns a value matching expectedCode. The next steps also uses ‘check’ to verify that the return value of the function serviceResponse() contains ‘searchFor’ (in this case we are using regular expression syntax). The next ‘check’ verifies that the function serviceResultCount() equals the value ‘expectedCount’. The two ‘show’ steps are used for debug purposes, showing respectively the url and the full response content. The final step saves the value of the response ‘id’ field, if present, in the variable $RID.

A successful execution of a scenario is displayed as below using an expandable section:



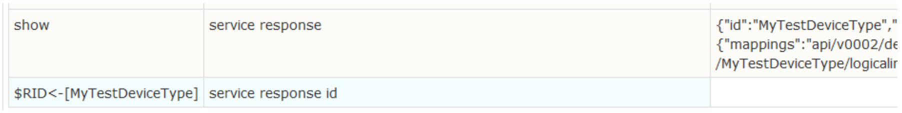
+ read service;	/device/types/MyTestDeviceType/devices			200	=~/"deviceId":"MyTestDevice1"/	1

Expanding reveals the individual steps called:



-read service;	/device/types/MyTestDeviceType/devices		
scenario	read service _ _ _ _ _	serviceName,customHeaders,expectedCode,searchFor,expectedCount	
call read service;	/device/types /MyTestDeviceType /devices		
check	service response code	200	
check	service response	/"deviceId":"MyTestDevice1"/ found in: {"results": [{"clientId":"d:9kpzic:MyTestDeviceType:MyTestDevice1","typeId":"MyTestDeviceType","deviceId":"MyTestDevice1","deviceInfo": {},"registration":{"auth":{"id":"a-9kpzic-cfbqvnuh2h","type":"app"},"date":"2019-04-11T10:58:09.823Z"},"status":{"alert": {"enabled":false,"timestamp":"2019-04-11T10:58:09.823Z"}},"refs":{"diag":{"logs":"/api/v0002/device/types /MyTestDeviceType/devices/MyTestDevice1/diag/logs","errorCodes":"/api/v0002/device/types/MyTestDeviceType/devices /MyTestDevice1/diag/errorCodes"},"location":"/api/v0002/device/types/MyTestDeviceType/devices/MyTestDevice1 /location"}}],"meta":{"total_rows":1}}	
check	service result count	1	
show	service url		https://9kpzic.i
show	service response		{"results": [{"clientId":"d:9 {},"registration {"enabled":false /MyTestDeviceT /MyTestDevice1, /location"}}],"m
$RID<-[MyTestDeviceType]	service response id		

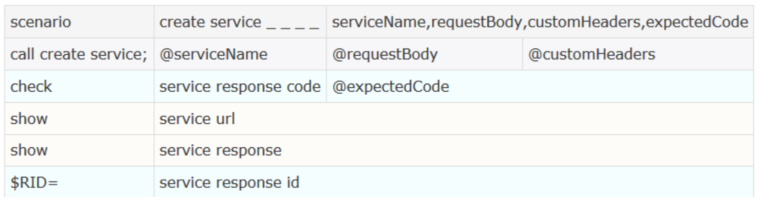
In this example there was no 'id' field in the response to the read service, so the value of $RID (MyTestDeviceType) is that of an earlier create step:



show	service response	{"id":"MyTestDeviceType"," {"mappings":"api/v0002/de /MyTestDeviceType/logicalir
$RID<-[MyTestDeviceType]	service response id	

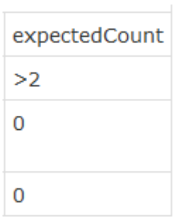
Care should be taken with the read scenario to not return too much data (via the serviceResponse () function) and potentially cause time outs or memory exhaustion. To compensate, the amount of memory allocated to the jvm can be increased when starting FitNesse (see the 'Getting Started' section).

The scenario below will invoke a POST service, passing in the additional parameter 'requestBody':



scenario	create service _ _ _ _	serviceName,requestBody,customHeaders,expectedCode	
call create service;	@serviceName	@requestBody	@customHeaders
check	service response code	@expectedCode	
show	service url		
show	service response		
$RID=	service response id		

As mentioned earlier, in a case where you are not sure how many items will be returned, you can leave the cell blank and FitNesse will skip the validation step. You can also use value comparisons for integer return values:



expectedCount
>2
0
0

Fixtures

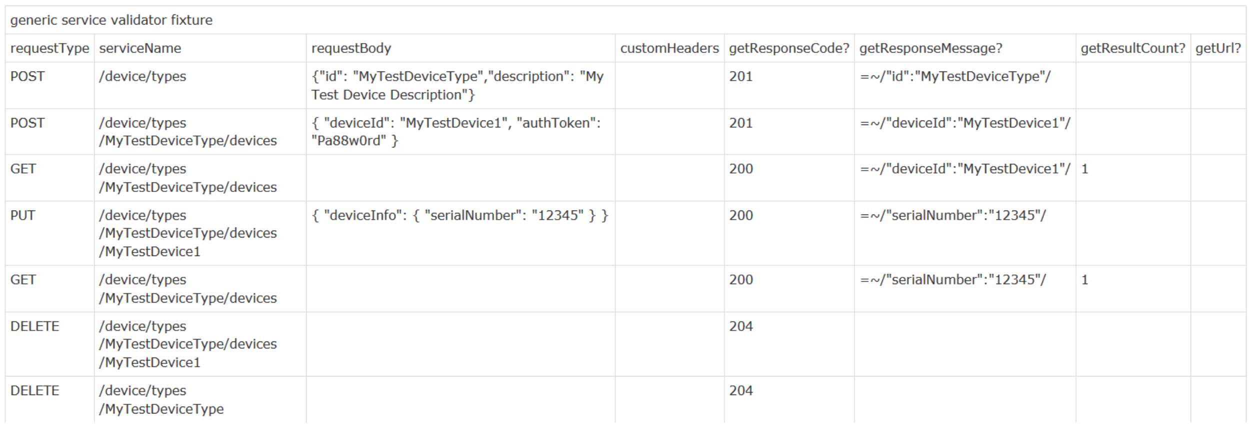
Generic Rest Service Fixtures

The generic service validator fixture is designed to handle any Rest Service.

Inputs to the generic fixture are: request type (GET, PUT, POST, DELETE, PATCH), service name, request body (can be left blank when not applicable) and custom headers (semi-colon separated list of key:value header pairs).

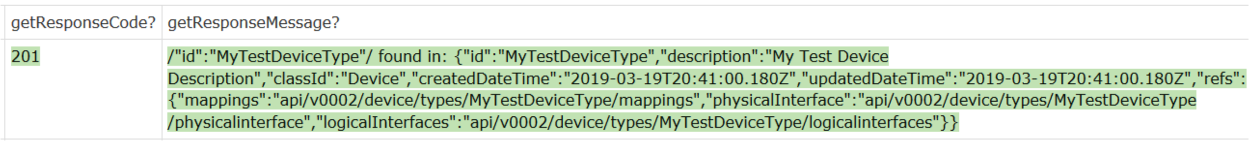
Methods that can be called for verification/display purposes: getResponseCode(), getResponseMessage(), getResultCount() and getUrl(). getResponseId() can also be invoked after POST and READ services to capture generated IDs (see the 'Capturing Generated IDs' section). getResponseMap() is available to return the fields in a read/post response as a map (see earlier section 'Full Http Response as Hash Map').

Below are a variety of 'device instance' and 'device type' services using the generic fixture:



generic service validator fixture							
requestType	serviceName	requestBody	customHeaders	getResponseCode?	getResponseMessage?	getResultCount?	getUrl?
POST	/device/types	{"id": "MyTestDeviceType","description": "My Test Device Description"}		201	=~/"id":"MyTestDeviceType"/		
POST	/device/types /MyTestDeviceType/devices	{ "deviceId": "MyTestDevice1", "authToken": "Pa88w0rd" }		201	=~/"deviceId":"MyTestDevice1"/		
GET	/device/types /MyTestDeviceType/devices			200	=~/"deviceId":"MyTestDevice1"/	1	
PUT	/device/types /MyTestDeviceType/devices /MyTestDevice1	{ "deviceInfo": { "serialNumber": "12345" } }		200	=~/"serialNumber":"12345"/		
GET	/device/types /MyTestDeviceType/devices			200	=~/"serialNumber":"12345"/	1	
DELETE	/device/types /MyTestDeviceType/devices /MyTestDevice1			204			
DELETE	/device/types /MyTestDeviceType			204			

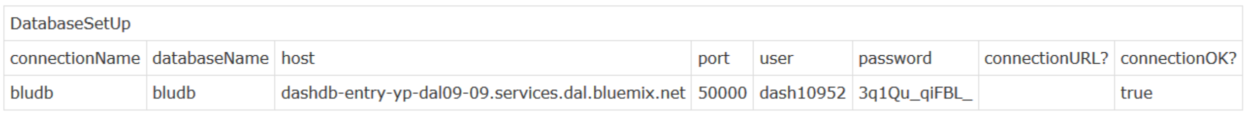
After execution, FitNesse displays the result as follows ({search for} found in: {response}):



getResponseCode?	getResponseMessage?
201	/"id":"MyTestDeviceType"/ found in: {"id":"MyTestDeviceType","description":"My Test Device Description","classId":"Device","createdDateTime":"2019-03-19T20:41:00.180Z","updatedDateTime":"2019-03-19T20:41:00.180Z","refs":{"mappings":"api/v0002/device/types/MyTestDeviceType/mappings","physicalInterface":"api/v0002/device/types/MyTestDeviceType /physicalinterface","logicalInterfaces":"api/v0002/device/types/MyTestDeviceType/logicalinterfaces"}}

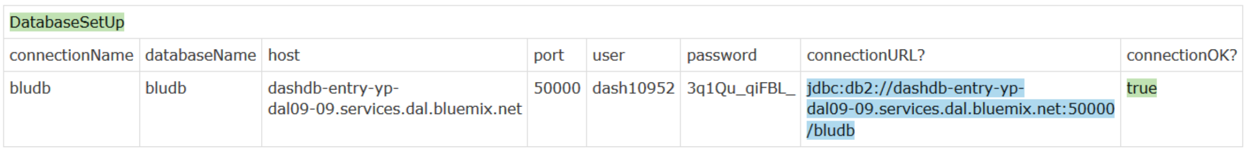
Generic Database Fixture

This fixture can be used to interact with any of the application databases. SuiteSetUp below creates a connection to BLUDB:



DatabaseSetUp							
connectionName	databaseName	host	port	user	password	connectionURL?	connectionOK?
bludb	bludb	dashdb-entry-yp-dal09-09.services.dal.bluemix.net	50000	dash10952	3q1Qu_qiFBL_		true

If the connection is created successfully the test output will look as follows:



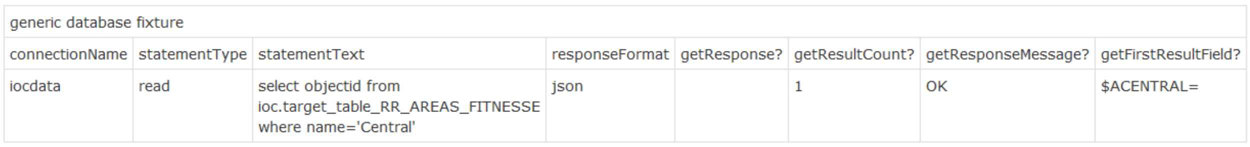
DatabaseSetUp							
connectionName	databaseName	host	port	user	password	connectionURL?	connectionOK?
bludb	bludb	dashdb-entry-yp-dal09-09.services.dal.bluemix.net	50000	dash10952	3q1Qu_qiFBL_	jdbc:db2://dashdb-entry-yp-dal09-09.services.dal.bluemix.net:50000/bludb	true

The connection name can then be passed to the fixture.

Inputs to the generic database fixture are: connection name (bludb), statement type (create, read, update, delete), statement text and response format for queries (currently only JSON is supported).

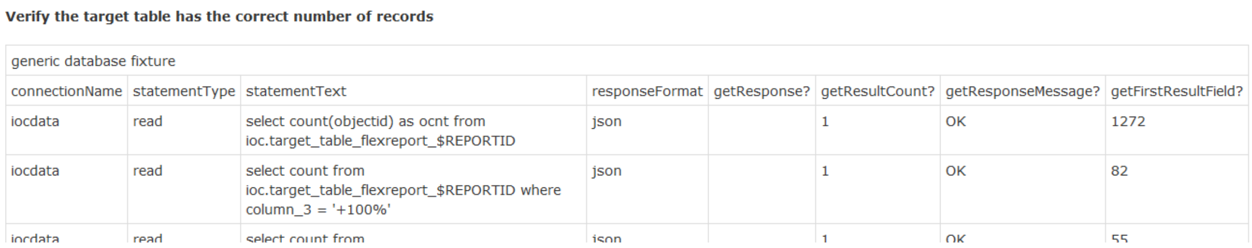
Methods that can be called for verification/display purposes: getResponse(), getResultCount(), getResponseMessage(). The latter returns “OK” for all successful executions. In the case of an SQL error the full error message is returned (useful for negative testing).

The method getFirstResultField() can be used to capture the value of the first field in the response. The example below captures the value of objectid for the ‘Central’ area in the Round Rock areas datasource and saves it in the Fitnesse variable $ACENTRAL:



generic database fixture							
connectionName	statementType	statementText	responseFormat	getResponse?	getResultCount?	getResponseMessage?	getFirstResultField?
iocdata	read	select objectid from ioc.target_table_RR_AREAS_FITNESSE where name='Central'	json		1	OK	$ACENTRAL=

Like other methods it can also be used to verify that the value returned is the expected one (by placing the expected value in the cell):

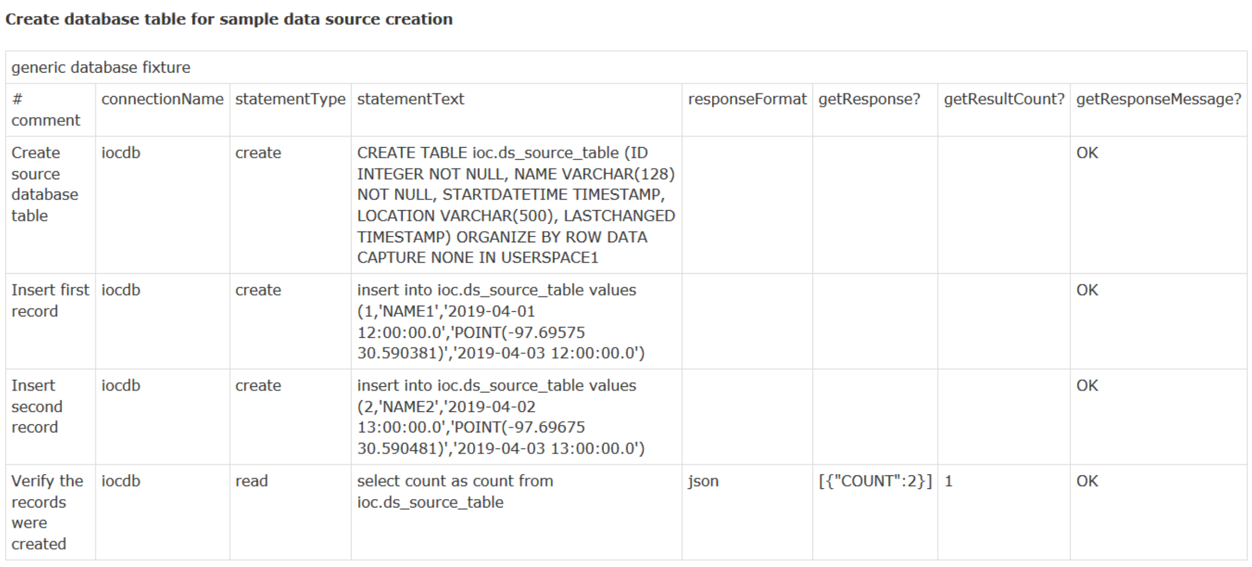


Verify the target table has the correct number of records

generic database fixture							
connectionName	statementType	statementText	responseFormat	getResponse?	getResultCount?	getResponseMessage?	getFirstResultField?
iocdata	read	select count(objectid) as ocnt from ioc.target_table_flexreport_$REPORTID	json		1	OK	1272
iocdata	read	select count from ioc.target_table_flexreport_$REPORTID where column_3 = '+100%'	json		1	OK	82
iocdata	read	select count from	json		1	OK	55

The method getResponseMap() is available to return the fields in a database read operation as a map (see earlier section ‘Full Http Response as Hash Map’).

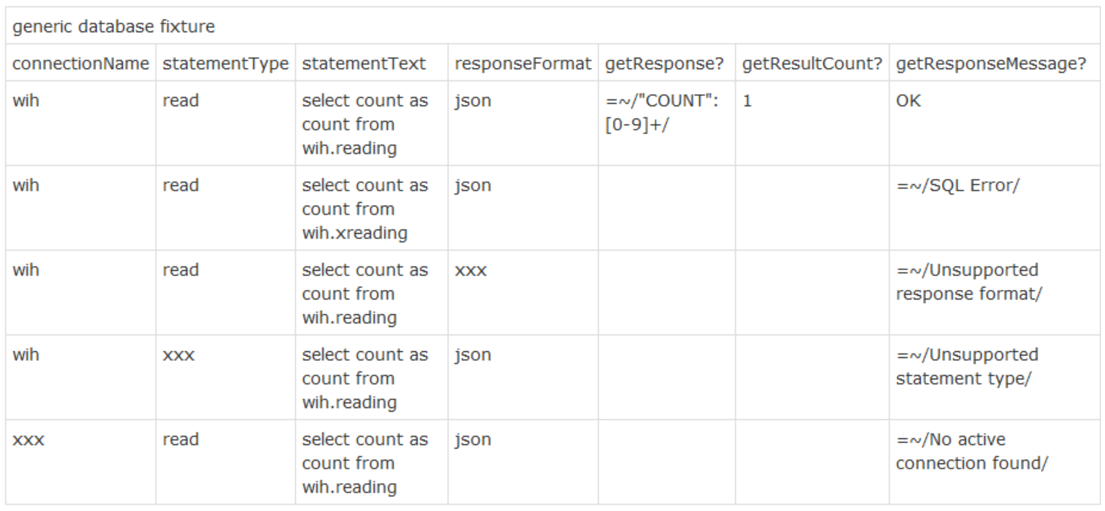
The sample decision table below is used to create a source table for a database type IOC data source. The last step verifies that two records are present in the target table:



Create database table for sample data source creation

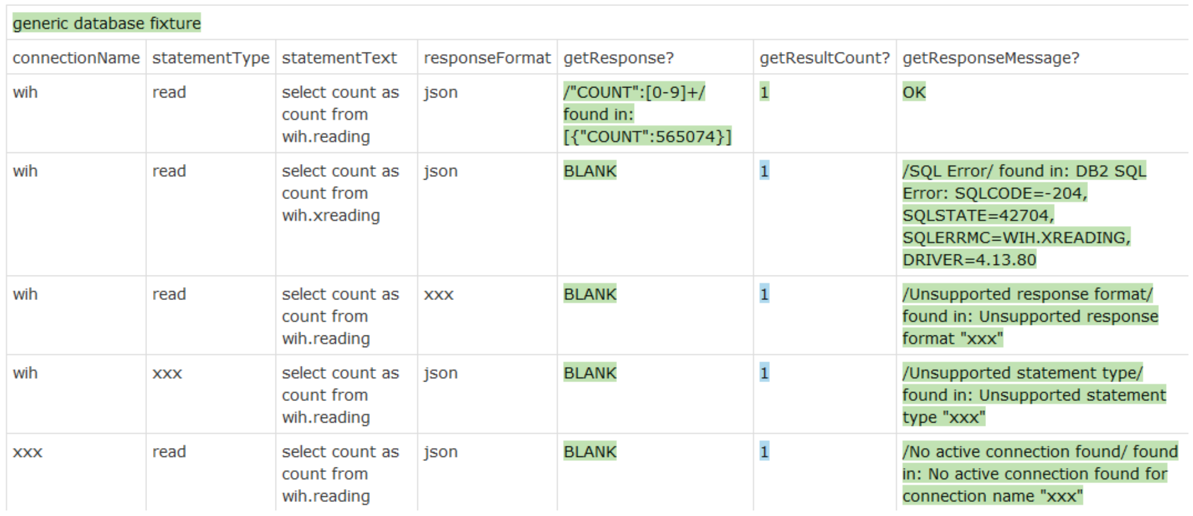
generic database fixture							
# comment	connectionName	statementType	statementText	responseFormat	getResponse?	getResultCount?	getResponseMessage?
Create source database table	iocdb	create	CREATE TABLE ioc.ds_source_table (ID INTEGER NOT NULL, NAME VARCHAR(128) NOT NULL, STARTDATETIME TIMESTAMP, LOCATION VARCHAR(500), LASTCHANGED TIMESTAMP) ORGANIZE BY ROW DATA CAPTURE NONE IN USERSPACE1				OK
Insert first record	iocdb	create	insert into ioc.ds_source_table values (1,'NAME1','2019-04-01 12:00:00.0','POINT(-97.69575 30.590381)','2019-04-03 12:00:00.0')				OK
Insert second record	iocdb	create	insert into ioc.ds_source_table values (2,'NAME2','2019-04-02 13:00:00.0','POINT(-97.69675 30.590481)','2019-04-03 13:00:00.0')				OK
Verify the records were created	iocdb	read	select count as count from ioc.ds_source_table	json	[{"COUNT":2}]	1	OK

Below is a trivial decision table that uses the fixture to perform some hypothetical negative tests:



generic database fixture						
connectionName	statementType	statementText	responseFormat	getResponse?	getResultCount?	getResponseMessage?
wih	read	select count as count from wih.reading	json	=~/"COUNT":[0-9]+/	1	OK
wih	read	select count as count from wih.xreading	json			=~/SQL Error/
wih	read	select count as count from wih.reading	xxx			=~/Unsupported response format/
wih	xxx	select count as count from wih.reading	json			=~/Unsupported statement type/
xxx	read	select count as count from wih.reading	json			=~/No active connection found/

After execution FitNesse produces the following:



generic database fixture						
connectionName	statementType	statementText	responseFormat	getResponse?	getResultCount?	getResponseMessage?
wih	read	select count as count from wih.reading	json	/"COUNT":[0-9]+/ found in: [{"COUNT":565074}]	1	OK
wih	read	select count as count from wih.xreading	json	BLANK	1	/SQL Error/ found in: DB2 SQL Error: SQLCODE=-204, SQLSTATE=42704, SQLERRMC=WIH.XREADING, DRIVER=4.13.80
wih	read	select count as count from wih.reading	xxx	BLANK	1	/Unsupported response format/ found in: Unsupported response format "xxx"
wih	xxx	select count as count from wih.reading	json	BLANK	1	/Unsupported statement type/ found in: Unsupported statement type "xxx"
xxx	read	select count as count from wih.reading	json	BLANK	1	/No active connection found/ found in: No active connection found for connection name "xxx"

Scenarios

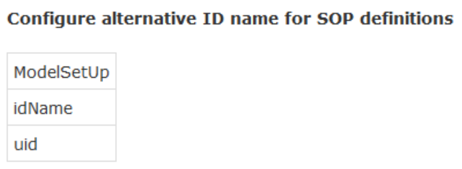
Scenarios are stored in the special ScenarioLibrary page. There appears to be a restriction in FitNesse whereby you cannot use the same method name in different fixtures; otherwise FitNesse may get confused and potentially call a requested method on the wrong object. To avoid this possibility we make sure that every method name is unique. We also avoid using hierarchical Java classes to implement scenarios as this has caused problems that are difficult to debug; try to keep all related scenarios in a single Java class.

Generic Rest Service Scenarios

These can be used to build test cases for arbitrary rest services. Currently, we have nine scenarios defined:

- read service (inputs: serviceName, customHeaders, expectedCode, searchFor, expectedCount). If there is an 'id' field in the response it is saved in the variable $RID
- create service (inputs: serviceName, requestBody, customHeaders, expectedCode). If a POST service generates an ID value (returned as an 'id' field in the response) it is saved in the variable $RID.
- update service (inputs: serviceName, requestBody, customHeaders, expectedCode)
- delete service (inputs: serviceName, customheaders, expectedCode, searchFor)
- create multipart form service (inputs: serviceName, filepathKey, filepathValue, additionalFields (semi-colon separated list of key:value pairs), expectedCode. This service will call POST with a file parameter (the path defined by 'filepathValue' must exist on the file system where FitNesse resides). If the service generates an ID value (returned as an 'id' field in the response) it is saved in the variable $RID.

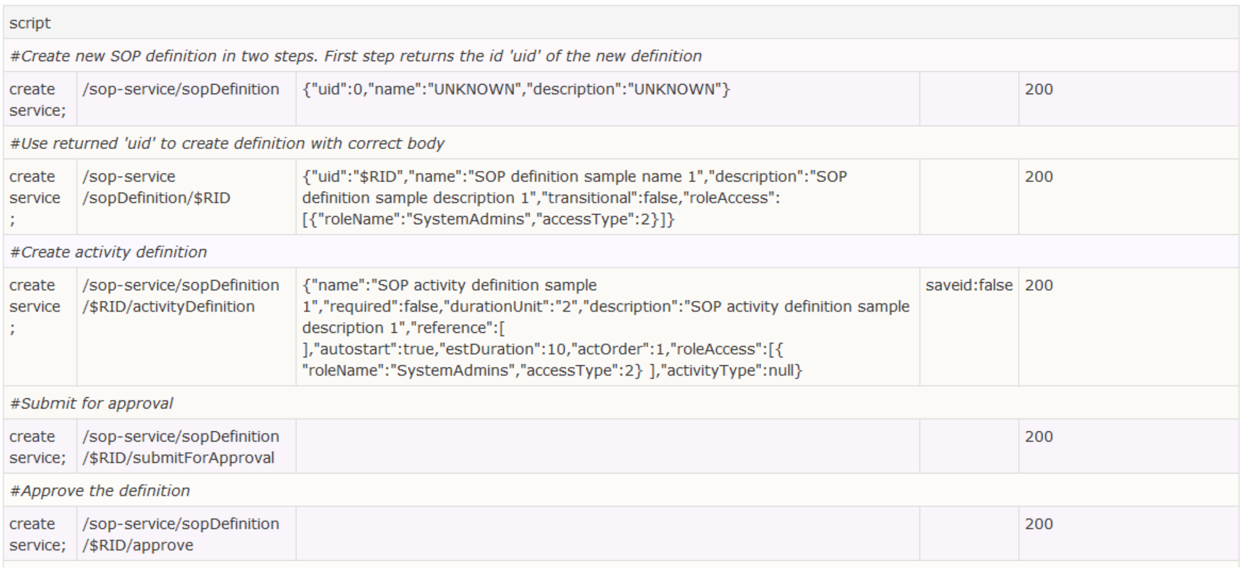
Below is a lifecycle scenario for SOP definitions that demonstrates most of the features of script tables and scenario libraries:



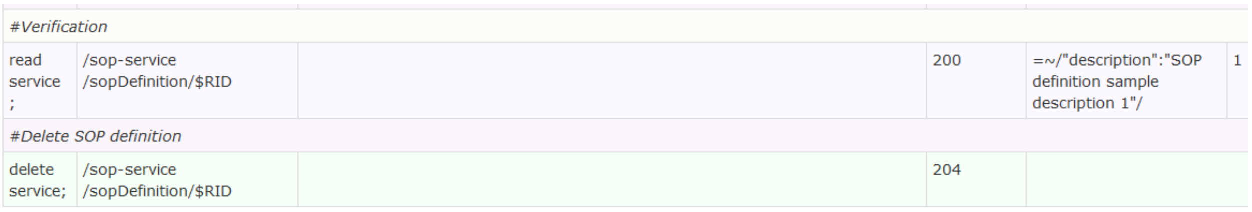
Configure alternative ID name for SOP definitions

ModelSetUp
idName
uid

Create/Read/Update/Read/Delete/Clean up SOP Definition

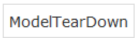


script				
#Create new SOP definition in two steps. First step returns the id 'uid' of the new definition				
create service;	/sop-service/sopDefinition	{"uid":0,"name":"UNKNOWN","description":"UNKNOWN"}		200
#Use returned 'uid' to create definition with correct body				
create service ;	/sop-service /sopDefinition/$RID	{"uid":"$RID","name":"SOP definition sample name 1","description":"SOP definition sample description 1","transitional":false,"roleAccess": [{"roleName":"SystemAdmins","accessType":2}]}		200
#Create activity definition				
create service ;	/sop-service/sopDefinition /$RID/activityDefinition	{"name":"SOP activity definition sample 1","required":false,"durationUnit":"2","description":"SOP activity definition sample description 1","reference":[ ],"autostart":true,"estDuration":10,"actOrder":1,"roleAccess":[{ "roleName":"SystemAdmins","accessType":2}],"activityType":null}	saveid:false	200
#Submit for approval				
create service;	/sop-service/sopDefinition /$RID/submitForApproval			200
#Approve the definition				
create service;	/sop-service/sopDefinition /$RID/approve			200



#Verification					
read service ;	/sop-service /sopDefinition/$RID		200	=~/"description":"SOP definition sample description 1"/	1
#Delete SOP definition					
delete service;	/sop-service /sopDefinition/$RID		204		

Restore defaults



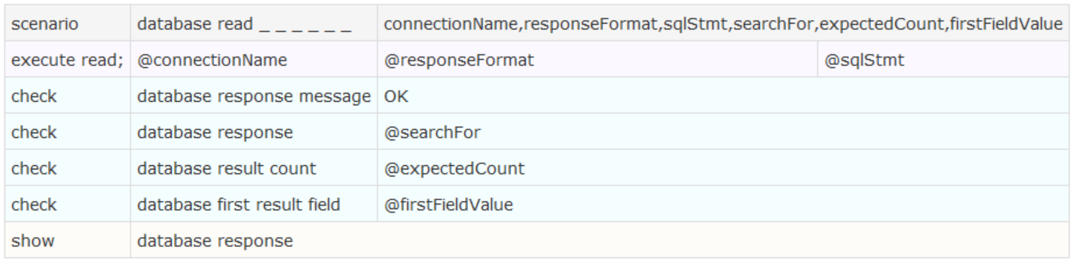
ModelTearDown

Database Scenarios

This is a set of scenarios that can be used to interact with the three main databases using the three connection names created in the Suite set up section. The defined scenarios are as follows:

- database read (inputs: connectionName, responseFormat, sqlStmt, searchFor, expectedCount,firstFieldValue)
- database create (inputs: connectionName, sqlStmt)
- database update (inputs: connectionName, sqlStmt)
- database delete (inputs: connectionName, sqlStmt)

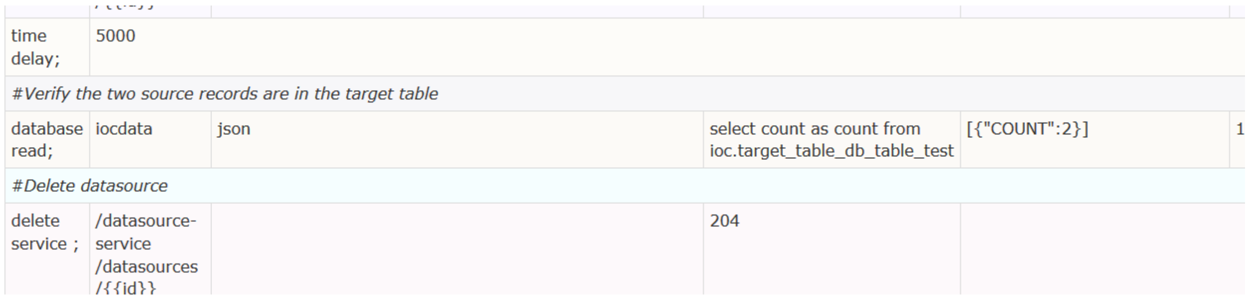
The database read scenario, for example, is defined as follows:



scenario	database read _ _ _ _ _ _	connectionName,responseFormat,sqlStmt,searchFor,expectedCount,firstFieldValue	
execute read;	@connectionName	@responseFormat	@sqlStmt
check	database response message	OK	
check	database response	@searchFor	
check	database result count	@expectedCount	
check	database first result field	@firstFieldValue	
show	database response		

This verifies that the response contains @searchFor, the number of records returned equals @expectedCount and the value of the first field equals @firstFieldValue.

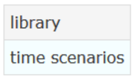
Database scenarios can be mixed with rest service scenarios in script tables to, for example, validate side effects. For example, in an IOC data source scenario we can verify that records have reached the target table:



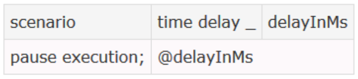
time delay;	5000				
#Verify the two source records are in the target table					
database read;	iocdata	json	select count as count from ioc.target_table_db_table_test	[{"COUNT":2}]	1
#Delete datasource					
delete service ;	/datasource-service /datasources /{{id}}		204		

Time Functions

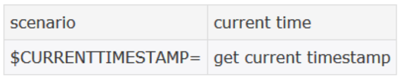
Currently the following scenarios/functions are defined:



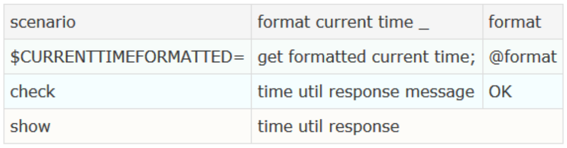
library
time scenarios



scenario	time delay _	delayInMs
pause execution;	@delayInMs	

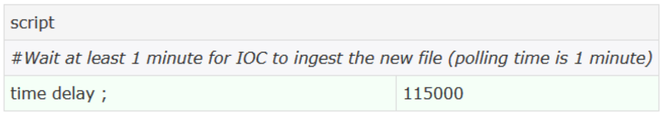


scenario	current time
$CURRENTTIMESTAMP=	get current timestamp



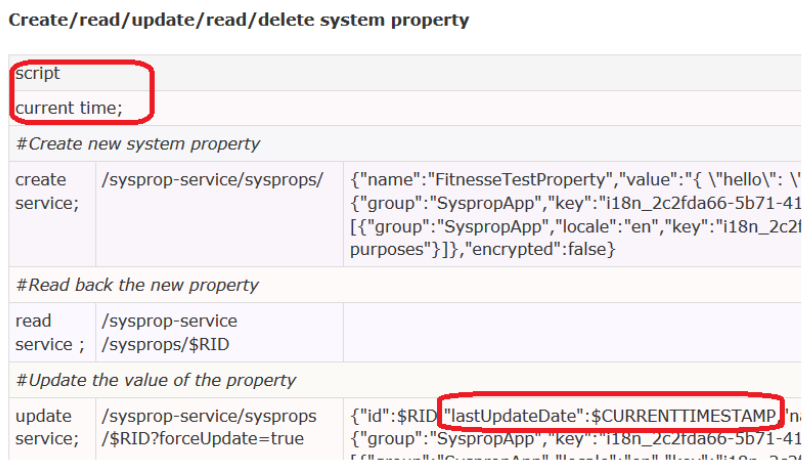
scenario	format current time _	format
$CURRENTTIMEFORMATTED=	get formatted current time;	@format
check	time util response message	OK
show	time util response	

The first function can be used to pause execution of a test script. For example, pausing while waiting for an IOC data receiver to poll for an update:



script	
#Wait at least 1 minute for IOC to ingest the new file (polling time is 1 minute)	
time delay ;	115000

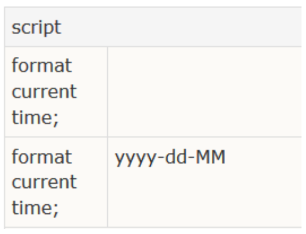
The second scenario will return the current timestamp in milliseconds. The scenario saves the value in a variable called $CURRENTTIMESTAMP, which can then be used in other scenarios. For example, in this example we pass the current timestamp to an IOC service that updates a system property:



Create/read/update/read/delete system property

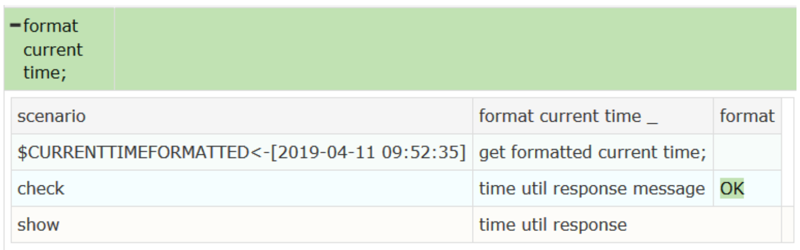
script		
current time;		
#Create new system property		
create service;	/sysprop-service/sysprops/	{"name":"FitnesseTestProperty","value":"{ \"hello\": \' {"group":"SyspropApp","key":"i18n_2c2fda66-5b71-41 [{"group":"SyspropApp","locale":"en","key":"i18n_2c2f purposes"}]},"encrypted":false}
#Read back the new property		
read service ;	/sysprop-service /sysprops/$RID	
#Update the value of the property		
update service;	/sysprop-service/sysprops /$RID?forceUpdate=true	{"id":$RID "lastUpdateDate":$CURRENTTIMESTAMP 'n {"group":"SyspropApp","key":"i18n_2c2fda66-5b71-41

The final scenario will return the current time as a formatted string. It takes the format string as parameter (if left blank it defaults to '`yyyy-MM-dd HH:mm:ss`'). The formatted time string is saved in the variable $CURRENTTIMEFORMATTED. This can then be used in subsequent scenarios. In the first example we use the default format, in the second we use a custom format:

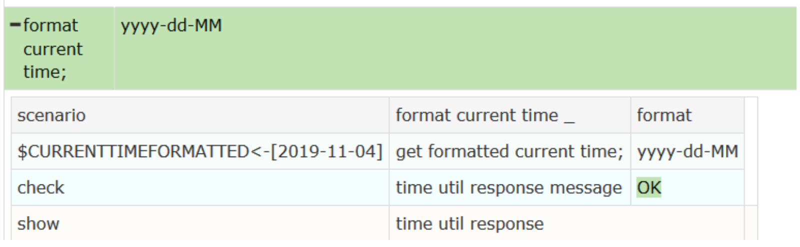


script	
format current time;	
format current time;	yyyy-dd-MM

The output in each case is shown below:



-format current time;		
scenario	format current time _	format
$CURRENTTIMEFORMATTED<-[2019-04-11 09:52:35]	get formatted current time;	
check	time util response message	OK
show	time util response	



-format current time;	yyyy-dd-MM	
scenario	format current time _	format
$CURRENTTIMEFORMATTED<-[2019-11-04]	get formatted current time;	yyyy-dd-MM
check	time util response message	OK
show	time util response	

File Functions

The following scenarios/functions are currently defined: local and remote copying of files, deleting local files and search/replace in files.

library
file scenarios

scenario	local file copy _ _	sourceFile,targetFile
copy file;	@sourceFile	@targetFile
check	file util response message	OK
show	file util response	

scenario	remote file copy _ _ _ _ _ _	sourceFile,targetFile,host,user,keyPath,keyPassword				
copy file remote;	@sourceFile	@targetFile	@host	@user	@keyPath	@keyPassword
check	file util response message	OK				
show	file util response					

scenario	local file delete _	sourceFile
delete file;	@sourceFile	
check	file util response message	OK
show	file util response	

scenario	file search replace _ _ _ _	sourceFile,searchReplacePairs,separator,pairSeparator		
replace strings in file;	@sourceFile	@searchReplacePairs	@separator	@pairSeparator
check	file util response message	OK		
show	file util response			

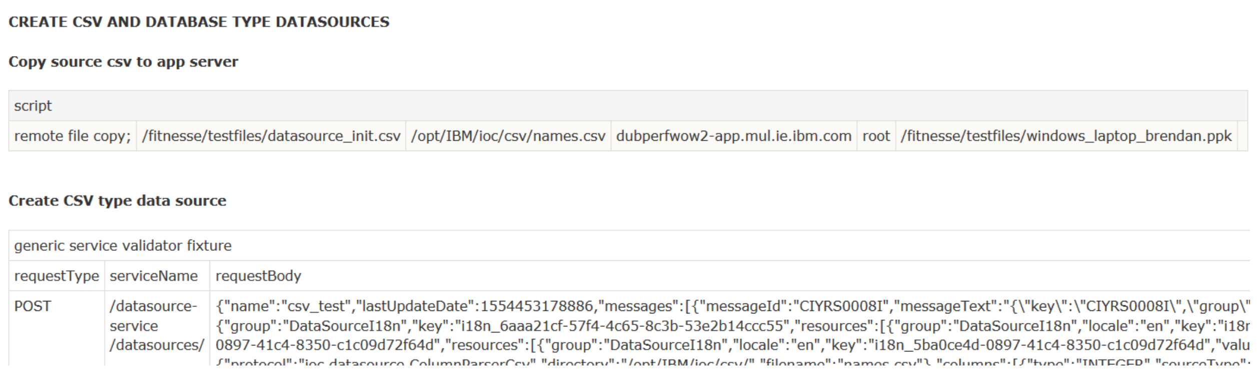
The local copy function takes two parmeters:

1. Source file path
2. Target file path

The remote copy function takes 6 parameters:

1. Source file path
2. Target File path
3. Target host/IP address
4. User name for remote host
5. Key file for source host with public and private keys
6. Password for key file (blank if none)

Remote copying can be used, for example, to set up a CSV type data source by copying source CSVs file to the default csv input folder on the app server:


CREATE CSV AND DATABASE TYPE DATASOURCES

Copy source csv to app server

script					
remote file copy;	/fitnesse/testfiles/datasource_init.csv	/opt/IBM/ioc/csv/names.csv	dubperfwow2-app.mul.ie.ibm.com	root	/fitnesse/testfiles/windows_laptop_brendan.ppk

Create CSV type data source

generic service validator fixture		
requestType	serviceName	requestBody
POST	/datasource-service /datasources/	{"name":"csv_test","lastUpdateDate":1554453178886,"messages":[{"messageId":"CIYRS0008I","messageText":"{\"key\":\"CIYRS0008I\",\"group\" {"group":"DataSourceI18n","key":"i18n_6aaa21cf-57f4-4c65-8c3b-53e2b14ccc55","resources":[{"group":"DataSourceI18n","locale":"en","key":"i18r 0897-41c4-8350-c1c09d72f64d","resources":[{"group":"DataSourceI18n","locale":"en","key":"i18n_5ba0ce4d-0897-41c4-8350-c1c09d72f64d","valu {"protocol":"ioc.datasource.ColumnParserCsv","directory":"/opt/IBM/ioc/csv/","filename":"names.csv"},"columns":[{"type":"INTEGER","sourceType":

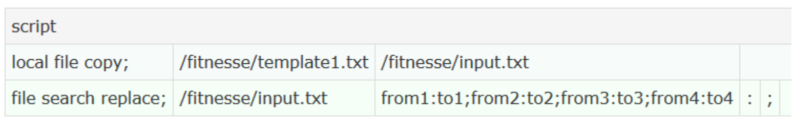
The local delete function takes one parameter:

1. Source file path

The search/replace scenario takes four parameters:

1. Input file
2. List of search/replace pairs
3. Separator between search and replace strings – if left blank this defaults to ‘:’
4. Separator between search/replace pairs – if left blank this defaults to ‘;’

This function could be used to prepare a template file for a variety of tests. E.g., you might copy the template file first, then replace a number of search strings:



script				
local file copy;	/fitnesse/template1.txt	/fitnesse/input.txt		
file search replace;	/fitnesse/input.txt	from1:to1;from2:to2;from3:to3;from4:to4	:	;

Before:
foo
from1...from2...
from3...from4...
bar

After:
foo
to1...to2...
to3...to4...
bar