GENESIS 2 - HOWTO, Tutorial, and Documentation

Lukasz Juszczyk

30. Dezember 2012

1 General Information

This is a HOWTO/Tutorial document for the *GENESIS2 (G2) testbed generator framework*. It is supposed to give an overview of how G2 can be installed, applied, and extended. Of course, this document is not complete and does not explain everything, but rather provides a good starting point for users. For getting familiar with it, I highly recommend to take a look at the source code and to play around with it in order to understand the framework's internals.

2 HOWTO Install

What you need is:

- **G2** - obviously...
- **Java 6 JDK** - Available at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- **Maven2** - Available at `http://maven.apache.org/`

It is very important that you have the Java 6 JDK installed, not just the JRE. G2 will need some libraries which are only avaible with the JDK.
The installation procedure looks as follows:

1. In the root directory run `mvn clean package`. This will download all dependencies, compile G2, package it and put the resulting jar and all dependencies into `target`.

2. Create (or copy) working directories for your G2 instances. This includes **exactly** one front-end and **at least one** back-end instance. You can take the directories from `sample/workingdirs` as templates.

3. For each instance, change into the working directory and start the front-end with '`java -jar genesis2-0.0.2.jar -fe`' and the back-end(s) with '`java -jar genesis2-0.0.2.jar`'. Use the front-end console for interactive scripting or, alternatively, execute '`java -jar genesis2-0.0.2.jar -s {script.groovy}`' for executing already existing scripts non-interactively.

For configuration, there is (for now) only one file to edit: `conf/genesis2.conf`. As G2 consists of a front-end (FE) part and an arbitrary number of back-end (BE) hosts, and sometimes users would like to have them running on the same machine, it is necessary to specify the ports in order to avoid conflicts. For that, each G2 instance (whether FE or BE) should be started in its own working directory which should contain the `conf/genesis2.conf` file, whose content is:

```
net.local.host=localhost
net.local.port=8080
```

The host is used for constructing the URI's of the generated components. The port is the instance's main port and should be distinct from the other instances. However, please note that G2 reserves this port plus the subsequent one. If you specify 8080 then it will also reserve 8081.

3 First Steps

I will not explain all details of G2 in this document, as this is not possible. For having a basic understanding of what G2 is I recommend to read the papers available at `http://www.infosys.tuwien.ac.at/prototyp/Genesis/`. However, some quick facts which are important for understanding G2:

1. G2 is a framework which is extensible via plugins. The plugins provide model types of testbed components (e.g., Web services, clients, registries) which are customizable but also the routines to translate these models into running instances.

2. G2 uses Groovy scripts for modeling and programming of testbeds.

3. A G2 testbed comprises a single FE, where scripts are executed and from where everything is steered, and a distributed BE, consisting of an arbitrary number of hosts on which the testbed instances are deployed.

4. Testbeds are modeled by instantiating model instances, customizing and programming them and deploying them on the BE hosts.

3.1 Providing Model Types via Plugins

Plugins which provide functionality to generate new testbed components must provide the corresponding types. These must be derived from the class `AbstractModelElement` and contain all necessary routines. In particular the following methods are important:

- The *constructor* which will be called, when the type is instantiated in the scripts via `create(...)`.
- `getProperty()` and `setProperty()` for assigning property values the Groovy way.
- And all the necessary public variables, getter/setter methods, and functions for accessing and customizing the model.

The plugin, which must be derived from `AbstractGenesis2Plugin`, must implement the method `deployElement(el)` which is will be called at the back-end, when a model type comes in and a running instance must be generated out of it.

3.2 Creating Model Instances

Model types which are registered at the framework, can be instantiated in two ways. 1) Either by instantiating a single object and customizing its properties or 2) by using Builders. Normal instantiations are done via the `create(...)` constructor. For instance:

```
def wsModel = webservice.create("TestService")

def clientModel = client.create()

def msgPertModel = msgperturber.create("xml")
```

Later, these instances can be customized, e.g., via:

```
wsModel.bindingStyle = "doc,lit"

def opModel = wsoperation.create("HelloWorld")
opModel.returnType = String
opModel.behavior = { return "hello world!" }

// bind operation to service
wsModel.operations = [opModel]
```

Builders, however, can help to instantiate and create nested structures of testbed models. As in the example above, first a Web service is created, later on a Web service operation is created, and finally, the operation is attached to the service. The more complex the nested structure is, the more script code would be needed to specify it. With Groovy Builders, this can be simplified as follows:

```
def wsModel = webservice.build {
  TestService(bindingStyle:"doc,lit") {
    HelloWorld(response:String) {
      return "hello world!"
    }
  }
}[0]
```

Please note the following. The Builder returns a list of models, as after specifying the `TestService` another specification could have followed. Therefore, we are accessing the created model via appending `[0]` to the call. Furthermore, Builders must be provided by the plugin which provides the model type. This is due to the necessary knowledge about which structures can be combined, how to interpret the commands, etc. In the presented listing the structure comprises a `webservice` that contains a `wsoperation` which, again, contains a `datatype` for the return type.

For more information about builders we refer to the Groovy guide[1], to the abstract class `AbstractModelBuilder`, and to the concrete implementations, such as `WebServiceBuilder` or `WsOperationBuilder`.

3.3 Customizing Model Instances

Once, model types are instantiated, they must be customized according to the requirements of the test run. What kind of customization is possible, depends heavily on the type. For instance, Web services can be assigned different binding styles, different operations with different functional behavior, etc., clients, can be assigned different behavior, and so on.

In any case, the model type must provide the corresponding means to customize it. These can be public variables, getter/setter methods, operational methods, etc. These will be accessible in the scripts, e.g., as in here:

---

[1] http://groovy.codehaus.org/Builders

```
wsModel.bindingStyle = "doc,lit"
wsModel.deleteOperation("HelloWorld")
```

3.4 Deploying Model Instances

Once customized and ready, model types can be sent to a BE host where running instance will be generated out of these. This happens via calling the `deployAt()` method.

```
def host1 = host.create("hostnameOrIP",8080)

wsModel.deployAt(host1)
```

If the model must be deployed on more than one host, this can be done via:

```
def host1 = host.create("hostnameOrIP",8080)
def host2 = host.create("hostnameOrIP",8181)

def hostList = [host1,host2]

wsModel.deployAt(host1,host2)
// or
wsModel.deployAt(hostList)
```

This causes G2 to serialize the model, to transfer it to the BE hosts, and to call the `deployElement()` method of the plugin that provides these types.

3.5 Implementing new Plugins

Well, that would require a lot of documentation to cover everything... Instead, I'm giving a very short overview.

First of all, it is necessary to distinguish between plugins which provide new model types and generate instances out of it, and these who are just doing some tasks in the background. Also between plugins which are standalone and these who must communicate with each others. Etc., etc.

For generating testbed components, plugins must provide the model type via the method `getTypesAndBuilders()` and implement the `deployElement()` routine accordingly in order to accept all provided types and to generate instances out of them. Furthermore, if the plugin wants to be accessible remotely via a Web service, it must implement this and provide it via `getPluginWsImplementor()`.

Furthermore, plugins can provide any other additional methods, macros, data objects, which are accessible via the shared runtime environment. These are registered via the methods `getAliases()` (aliases for accessing the plugin instance), `getCommands()` (register commands/macros via aliases), etc. I recommend to get familiar with the classes `AbstractGenesis2Plugin` and all its subclasses implementing specific plugins in order to understand the plugin system.

3.5.1 Shared Runtime Environment

The Shared Runtime Environment (SRE) is the binding that connects all Groovy scripts that are executed within G2. It provides access to all plugins (via their aliases), to the model types (via their names), macros, global variables, etc. G2 makes sure that SRE is the same on all hosts in the testbed in order to have a homogeneous environment, especially as it installs the same set of plugins on each host and each plugins performs the same deployment procedure on its host.

In the Java code (e.g. of the plugin implementation), the SRE can be accessed via `ExecutionEnvironment.getGlobalInstance()` in order to register variables/scripts in it. However, these changes will be only local. If they are supposed to be global (on each instance of the testbed), either make sure that each plugin instance registers the same data or use the ObjectPropagatorPlugin which automatically forwards all declarations to all known instances (`prop.variablename=value`).