Forge Reference Guide

Authors & Contributors

Lincoln Baxter III

Introduction

How many times have you wanted to start a new project in Java EE, but struggled to put all the pieces together?

Has the Maven archetype syntax left you scratching your head? Everyone else is talking about cool new tools in other languages or frameworks, and you're left thinking, "I wish it were that easy for me." Well, there's good news: You don't have to leave Java EE just to find a developer tool that makes starting out simple. JBoss Forge is heating up Java EE, and is ready to work it into a full-fledged project.

In addition to being a rapid-application generation tool, Forge is also an incremental enhancement tool that lets you to take an existing Java EE projects and safely work-in new functionality. Forge comprehends your entire project, including the abstract structure of the files, and can make intelligent decisions of how and what to change.

Whether you want to get your startup going today, or make your big customers happy tomorrow, Forge is a tool you should be looking at.

Installation

Installing Forge is a relatively short process, and this guide will take you through the fundamentals (providing links to external materials if required;) however, if you encounter any issues with this process, please ask in the *Forge Users* [https://lists.jboss.org/mailman/listinfo/forge-users] mailing list, or if you think something is wrong with this guide, *report a defect* [http://bit.ly/forgeissues] under "Documentation".

1.1. Installing a distribution download

Follow these steps to install a Forge distribution:

1. Ensure that you have already installed a *Java 6+ JDK* [http://www.oracle.com/technetwork/java/javase/downloads/index.html] and *Apache Maven 3.0+* [http://maven.apache.org/download.html]
2. *Download* [http://sourceforge.net/projects/jboss/files/Forge/] and Un-zip Forge (or a recent *snapshot build* [http://repository.jboss.org/nexus/content/groups/public/org/jboss/seam/forge/forge-distribution/]) into a folder on your hard-disk, this folder will be your `FORGE_HOME`
3. Add '`$FORGE_HOME/bin`' to your path (*windows* [http://www.google.com/search?q=windows+edit+path], *linux* [http://www.google.com/search?q=linux+set+path], *mac osx* [http://www.google.com/search?q=mac+osx+edit+path])
4. Open a command prompt and run 'forge'

That's it, you've now got Forge installed, but what to do next?

There are a few things you should probably check-out. If you are confused at any time, try pressing <TAB>. For instance, if you have not yet seen the Forge built-in commands, you may either press <TAB> to see a list of the currently available commands, or get a more descriptive list by typing:

```
$ list-commands --all
```

You may also use the 'help' command for more detailed information about available Forge, a plugin, or a command.

```
$ help {plugin-name} {command-name}
```

Generating a basic Java EE web-application

For the most part, people interested in Forge are likely interested in creating web-applications. Thusly, this chapter will overview the basic steps to generate such an application using Forge.

2.1. First steps with Scaffolding

Assuming you have already completed the steps to *install Forge*, the first thing you'll need to do is download and install *JBoss Application Server 6.0* [http://www.jboss.org/jbossas/downloads.html]. This server will host your application once is is built.

Next, follow these steps to create your skeleton web-application; be sure to replace any {ARGS} with your own personal values. Also keep in mind that while typing commands, you may press <TAB> at any time to see command completion options:

1. Execute `$ forge` from a command prompt.
2. Create a new project:

```
$ new-project --named {name} --topLevelPackage {com.package} --projectFolder {/directory/path}
```

3. Install the web-scaffold facet, and press ENTER to confirm installation of required facet dependencies and/or packaging types:

```
$ scaffold setup
```

4. That's it! Now in a separate command shell, build your project using Maven, and deploy it onto your JBoss Application Server instance:

```
$ mvn clean package
$ mvn jboss:hard-deploy
$ mvn jboss:start
```

5. Access your application at: `http://localhost:8080/{name}-1.0.0-SNAPSHOT/`

Developing a Plugin

Part of Forge's architecture is to allow extensions to be created with extreme ease. This is done using the same programming model that you would use for any CDI or Java EE application, and you should quickly recognize the annotation-driven patterns and practices applied.

A Forge plugin could be as simple as a tool to print files to the console, or as complex as deploying an application to a server, 'tweet'ing the status of your latest source-code commit, or even sending commands to a home-automation system; the sky is the limit!

3.1. Referencing the Forge APIs

Because Forge is based on Maven, the easiest way to get started quickly writing a plugin is to create a new maven Java project. This can be done by hand, or using Forge's build in plugin project facet.

3.1.1. Using Forge

In two short steps, you can have a new plugin-project up and running; this can be done using Forge itself!

1. Execute `$ forge` from a command prompt.
2. Create a new project:

```
$ new-project --named {name} --topLevelPackage {com.package} --projectFolder {/directory/path}
```

3. Install the Forge API facet, select the API version you wish to use, and press ENTER to confirm installation of required facet dependencies:

```
$ install forge.api
```

That's it! Now your project is ready to be compiled and installed in Forge, but you may still want to *add some commands*.

3.1.2. With Maven)

If you do not wish to create a new plugin project using Forge itself, you will need to manually include the Forge-API dependencies. For purposes of simplicity, we have pasted a sample Maven POM file which can be used as a starting point for a new plugin:

Tip

`'org.jboss.seam.forge : forge-shell-api : {version}'` is the only dependency you must include in your project.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd"
   xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.plugin</groupId>
  <artifactId>example</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <properties>
    <forge.api.version>[1.0.0-SNAPSHOT,)</forge.api.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.jboss.seam.forge</groupId>
      <artifactId>forge-shell-api</artifactId>
      <version>${forge.api.version}</version>
    </dependency>
  </dependencies>

  <repositories>
    <repository>
      <id>jboss</id>
      <url>https://repository.jboss.org/nexus/content/groups/public/</url>
    </repository>
  </repositories>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
```

```
      </plugin>
    </plugins>
  </build>
</project>
```

3.2. Implementing the Plugin interface

Your class must implement the `org.jboss.seam.forge.shell.plugins.Plugin` interface.

```
import org.jboss.seam.forge.shell.plugins.Plugin;

public class ExamplePlugin implements Plugin {
}
```

3.3. Naming your plugin

Each plugin should be given a name. This is done by adding the `@org.jboss.seam.forge.shell.plugins.Alias` annotation to your plugin class. By default, if no `@Alias` annotation is found, the lower-case Class name will be used; for instance, our `ExamplePlugin`, above, would be executed by typing:

```
$ exampleplugin
```

Now we will add a name to our plugin.

```
@Alias("example")
public class ExamplePlugin implements Plugin {
   // commands
}
```

Our named `@Alias("example") ExamplePlugin` would be executed by typing:

```
$ example
```

3.4. Ensure all required classes are on the CLASSPATH

All imports must be available on the `CLASSPATH`. If your Plugin depends on classes that are not provided by Forge, then you must either package those classes in the JAR file containing your Plugin (for instance, using the maven *shade plugin* [http://maven.apache.org/plugins/maven-

shade-plugin/]), or you must ensure that the required dependencies are also placed on the `CLASSPATH` (typically in the `$FORGE_HOME/lib` folder,) otherwise your plugin will *not* be loaded.

3.5. Make your Plugin available to Forge

After following all of the steps in *this section*, you should now be ready to install your Plugin into the Forge environment. This is accomplished simply by packaging your Plugin in a JAR file with a CDI activator, otherwise referred to as a `/META-INF/beans.xml` file.

Tip

You must include a /META-INF/beans.xml file in your JAR, or none of the classes in your archive will be discovered; therefore, your Plugin will not be made available to Forge.

3.6. Add commands to your plugin

Now that you have implemented the `Plugin` interface, it's time to add some functionality. This is done by adding "Commands" to your plugin class. Commands are plain Java methods in your plugin Class. Plugin methods must be annotated as either a `@DefaultCommand`, the method to be invoked if the plugin is called by name (with no additional commands), or `@Command(name="...")`, in which case a the plugin name and command name must both be used to invoke the method.

Commands also accept `@Options` parameters as arguments. These are *described in detail* later in this section.

3.6.1. Default commands

Default commands must be annotated with `@DefaultCommand`, and are not named; you may still provide help text or command metadata. Each plugin may have only one `@DefaultCommand`.

The following default command would be executed by executing the plugin by its name:

```
public class ExamplePlugin implements Plugin {
  @DefaultCommand
  public void exampleDefaultCommand( @Option String opt ) {
    // this method will be invoked, and 'opt' will be passed from the command line
  }
}
```

```
$ exampleplugin some-input
```

In this case, the value of '`opt`' will be "some-input". `@Options` are *described in detail* later in this section.

3.6.2. Named commands

Named commands must, to little surprise, be given a name with which they are invoked. This is done by placing the `@Command(name="...")` annotation on a public Java method in your `Plugin` class.

The following command would be executed by executing the plugin by its name, followed by the name of the command:

```
public class ExamplePlugin implements Plugin {
   @Command(name="perform")
   public void exampleCommand( @Option(required=false) String opt, PipeOut out) {
      out.println(">> the command \"perform\" was invoked with the value: " + opt);
   }
}
```

```
$ exampleplugin perform
>> the command "perform" was invoked with the value: null
```

Notice that our command method has a parameter called "`PipeOut`," in addition to our 'opt' parameter. `PipeOut` is a special parameter, which can be placed in any order. It provides access to a variety of shell output functions, including enabling color and controlling piping between plugins.

Along with `PipeOut`, there is also a `@PipeIn InputStream stream` annotation, which is used to inject a piped input stream (output from another Plugin's `PipeOut`.) These concepts will be described more in the section on *piping*, but for now, you should just know that `PipeOut` is used to write output to the Forge console.

3.7. Understanding command @Options

Once we have a command or two in our Plugin, it's time to give our users some control over what it does; to do this, we use `@Option` params; options enable users to pass information of various types into our commands.

Options can be named, in which case they are set by passing the `--name` followed immediately by the value, or if the option is a boolean flag, simply passing the flag will signal a `true` value. Named parameters may be passed into a command in any order, while unnamed parameters must be passed into the command in the order with which they were defined.

3.7.1. --named options

As mentioned above, options can be given both a long-name and/or a short-name. in which case, they would be defined like this:

```
@Option(name="one", shortName="o")
```

Short named parameters are called using a single dash '-' followed by the letter assigned '-o'; whereas long-named parameters are called using a double dash '--' immediately followed by the name '--one'.)

For example, the following command accepts several options, named 'one', and 'two':

```
public class ExamplePlugin implements Plugin {
  @Command(name="perform")
  public void exampleCommand(
            @Option(name="one", shortName="o") String one,
            @Option(name="two") String two,
            PipeOut out) {
    out.println(">> option one equals: " + one);
    out.println(">> option two equals: " + two);
  }
}
```

The above command, when executed, would produce the following output:

```
$ example-plugin perform --one cat --two dog
>> option one equals: cat
>> option two equals: dog
```

Tip

Named parameters can be called in any order. Notice that we could have also called the command with options 'one' and 'two' in reverse order, or by using their short names. These commands are equivalent:

```
$ example-plugin perform --one cat --two dog
$ example-plugin perform --two dog --one cat
```

```
$ example-plugin perform --two dog -o cat
```

3.7.2. Ordered options

In addition to `--named` option parameters, as described *above*, parameters may also be passed on the command line by the order in which they are entered. These are called "ordered option parameters", and do not require any parameters other than help or description information.

```
@Option String value
```

The order of the options in the method signature controls how values are assigned from parsed Forge shell command statements.

For example, the following command accepts several options, named 'one', and 'two':

```
public class ExamplePlugin implements Plugin {
  @Command(name="perform")
  public void exampleCommand(
              @Option String one,
              @Option String two,
              PipeOut out) {
     out.println(">> first option equals: " + one);
     out.println(">> second option equals: " + two);
  }
}
```

The above command, when executed, would produce the following output:

```
$ example-plugin cat dog
>> option one equals: cat
>> option two equals: dog
```

3.7.3. Combining --named and ordered options

Both *`--named`* and *ordered* option parameters can be mixed in the same command; there are some constraints on how commands must be typed, but there is a great deal of flexibility as well.

```
@Option String value,
```

```
@Option(name="num") int number
```

The order of ordered options in the method signature controls how values are assigned from the command line shell, whereas the named options have no bearing on the order in which inputs are provided on the command line.

For example, the following command accepts several options, named 'one', 'two', and several more options that are not named:

```
public class ExamplePlugin implements Plugin {
  @Command(name="perform")
  public void exampleCommand(
              @Option(name="one") String one,
              @Option(name="two") String two,
              @Option String three,
              @Option String four,
              PipeOut out) {
    out.println(">> first option equals: " + one);
    out.println(">> second option equals: " + two);
    out.println(">> third option equals: " + three);
    out.println(">> fourth option equals: " + four);
  }
}
```

The above command, when executed, would produce the following output:

```
$ example-plugin --one cat --two dog bird lizard
>> option one equals: cat
>> option two equals: dog
>> option one equals: bird
>> option two equals: lizard
```

However, we could also achieve the same result by re-arranging parameters, and as long as the name-value pairs remain together, and the ordered values are passed in the correct order, interpretation will remain the same:

```
$ example-plugin --two dog bird --one cat lizard
>> option one equals: cat
>> option two equals: dog
>> option one equals: bird
```

```
>> option two equals: lizard
```

3.7.4. Option attributes and configuration

This table describes all of the available `@Option(...)` attributes and their usage.

Attribute	Type	Description	Default
`name`	`String`	If specified, defines the *--name* with which this option may be passed on the command line. If left blank, this option will be *ordered* (not named,) and will be passed in the order with which it was written in the Method signature. `@Option(name="exampleName")`	none
`required`	`boolean`	Options may be declared as required when they must be supplied in order for proper command execution. The shell will enforce this requirement by prompting the user for valid input if the option is omitted when the command is executed. `@Option(required="false")`	false
`shortName`	`String`	If specified, defines the short name by which this option may be called on the command line. Short names must be one character in length. *short name* with which this option may be passed on the command line.	none

3.8. Piping output between plugins

Much like a standard UNIX-style shell, the Forge shell supports piping IO between executables; however in the case of forge, piping actually occurs between plugins, commands, for example:

```
$ cat /home/username/.forge/config | grep automatic
@/* Automatically generated config file */;
```

This might look like a typical BASH command, but if you run forge and try it, you may be surprised to find that the results are the same as on your system command prompt, and in this example, we are demonstrating the pipe: '|'

In order to enable piping in your plugins, you must use one or both of the `@PipeIn InputStream stream` or `PipeOut out` command arguments. Notice that `PipeOut` is a java type that must be used as a Method parameter, whereas `@PipeIn` is an annotation that must be placed on a Java `InputStream` Method parameter.

`` `PipeOut out` `` - by default - is used to print output to the shell console; however, if the plugin on the left-hand-side is piped to a secondary plugin on the command line, the output will be written to the `` `@PipeIn InputStream stream` `` of the plugin on the right-hand-side:

```
$ left | right
```

Or in terms of pipes, this could be thought of as a flow of data from left to right:

```
$ PipeOut out -> @PipeIn InputStream stream
```

Notice that you can pipe output between any number of plugins as long as each uses both a `@PipeIn InputStream` and `PipeOut`:

```
$ first command | second command | third command
```

Take the '`grep`' command itself, for example, which supports two methods of invocation: Invocation on one or more `Resource<?>` objects, or invocation on a piped `InputStream`.

Tip

If no piping is invoked (e.g: via standalone execution of the plugin), a piped `InputStream` will be null. In addition, piped `InputStreams` do not need to be closed; Forge will handle cleanup of these streams.

```
@Alias("grep")
@Topic("File & Resources")
@Help("print lines matching a pattern")
public class GrepPlugin implements Plugin
{
   @DefaultCommand
   public void run(
```

```
      @PipeIn final InputStream pipeIn,
      @Option(name = "ignore-case", shortName = "i", flagOnly = true) boolean ignoreCase,
      @Option(name = "regexp", shortName = "e") String regExp,
      @Option(description = "PATTERN") String pattern,
      @Option(description = "FILE ...") Resource<?>[] resources,
      final PipeOut pipeOut
  ) throws IOException
  {
    Pattern matchPattern = /* determine pattern (omitted for space) */;

    if (resources != null) {

      /* User passed file(s) on the command line; grep those. */

      for (Resource<?> r : resources) {
        InputStream inputStream = r.getResourceInputStream();
        try {
          match(inputStream, matchPattern, pipeOut, ignoreCase);
        }
        finally {
          inputStream.close();
        }
      }
    }
    else if (pipeIn != null) {

      /* No files were passed on the command line; check for a
       * piped InputStream and use that.
       */

      match(pipeIn, matchPattern, pipeOut, ignoreCase);
    }
    else {

      /* No input was passed to the plugin. */

      throw new RuntimeException("Error: arguments required");
    }
  }

     private void match(InputStream instream, Pattern pattern, PipeOut pipeOut, boolean
caseInsensitive) throws IOException {
    StringAppender buf = new StringAppender();
```

```
      int c;
      while ((c = instream.read()) != -1) { /* Read from the given stream. */
        switch (c) {
        case '\r':
        case '\n':
          String s = caseInsensitive ? buf.toString().toLowerCase() : buf.toString();

          if (pattern.matcher(s).matches()) {
            pipeOut.println(s); /* Write to the output pipe. */
          }
          buf.reset();
          break;
        default:
          buf.append((char) c);
          break;
        }
      }
    }
}
```

Appendix A. Affiliation

This reference guide was written for Forge: a tool from JBoss by Red Hat, Inc.