Seam 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".

If you would like to get involved, stop by the *Forge Dev* [https://lists.jboss.org/mailman/listinfo/forge-dev] mailing list, *file an issue* [http://bit.ly/forgeissues], or come find us on irc.freenode.net, in channel #seam-dev.

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].
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. Consider installing *Git* [http://git-scm.com/] and *Maven 3.0+* [http://maven.apache.org/] (both optional)
5. Open a command prompt and run 'forge'

```
localhost:~ $ forge
[no project] ~ $
```

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

Tip

Having problems? *Tell us* [https://issues.jboss.org/secure/CreateIssue.jspa?pid=12311102&issuetype=1].

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 or 7.0* [http://www.jboss.org/jbossas/downloads.html]. This server will host your application once it 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. Set up scaffolding, and press ENTER to confirm installation of any required facet dependencies and/or packaging types:

```
$ scaffold setup
```

   Then set up default index files, a default template and home-page. This creates the page that users will first see when accessing your application.

```
$ scaffold create-indexes --overwrite
```

4. Set up persistence (JPA), and press ENTER to confirm installation of any required facet dependencies and/or packaging types, and remember to press TAB if you are not sure what comes next.

```
$ persistence setup --provider {your JPA implementation} --container {your container}
```

If you do not wish to use a Java EE container default data-source, you can also specify additional connection parameters such as JNDI data-source names, JDBC connection information, and data-source types. Note, however, that this means you will probably need configure your application server to provide this new data-source and/or database connection.

5. Create some JPA entities:

```
$ entity --named Customer
```

At which point Forge should automatically pick-up the newly created entity, and will be ready to add fields.

```
Customer.java $ field string --named firstName
Customer.java $ field string --named lastName
```

You may inspect the entity using 'ls'.

```
Customer.java $ ls

[fields]
private::String::firstName;       private::String::lastName;
private::int::version;            private::long::id;

[methods]
public::getFirstName()::String                                  public::getId()::long
public::getLastName()::String                                   public::getVersion()::int
public::setFirstName(final String firstName)::void              public::setId(final long id)::void
public::setLastName(final String lastName)::void                public::setVersion(final int
 version)::void
public::toString()::String

Customer.java $
```

6. Once you have created fields in the entity, it's time to generate some scaffolding. For now, use the Forge default scaffold; you'll also need to confirm installation of a few libraries/ dependencies.

```
Customer.java $ scaffold from-entity

No scaffold type was provided, use Forge default? [Y/n]
```

```
Install which version of Metawidget?

  1 - [org.metawidget:metawidget:1.0.5]
  2 - [org.metawidget:metawidget:1.10]

Choose an option by typing the number of the selection: 2
Install which version of Seam Persistence?

  1 - [org.jboss.seam.persistence:seam-persistence:3.0.0-SNAPSHOT]
  2 - [org.jboss.seam.persistence:seam-persistence:3.0.0.Final]

Choose an option by typing the number of the selection: 2

Wrote /src/main/webapp/resources/forge-template.xhtml
Wrote /src/main/webapp/resources/forge.css
Wrote /src/main/webapp/resources/favicon.ico
Wrote /src/main/java/com/scaffold/domain/Customer.java
Wrote /src/main/java/com/scaffold/view/CustomerBean.java
Wrote /src/main/webapp/scaffold/customer/view.xhtml
Wrote /src/main/webapp/scaffold/customer/create.xhtml
Wrote /src/main/webapp/scaffold/customer/list.xhtml
***SUCCESS*** Generated UI for [com.scaffold.domain.Customer]

Customer.java $
```

7. That's it! Now build your project and deploy it onto your JBoss Application Server instance:

```
$ build
```

Or, if you have installed *Apache Maven 3.0+* [http://maven.apache.org/download.html], and set your JBOSS_HOME environment variable to the location of your JBoss AS server installation folder, you can also build and deploy your application with the following commands.

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

Your project can correspondingly be undeployed using:

```
$ mvn jboss:hard-undeploy
```

```
$ mvn jboss:stop
```

8. Access your application at: `http://localhost:8080/{name}-1.0.0-SNAPSHOT/`
9. Access scaffolded entities at: `http://localhost:8080/{projectname}/faces/scaffold/{entity}/view.xhtml`

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

```
$ project install-facet forge.api
```

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

Example 3.1. Creating a new Forge Plugin Project

```
[no project] Desktop $
[no project] Desktop $ new-project --named example-plugin --topLevelPackage com.example.forge.plugin
Use [~/Desktop/example-plugin] as project directory? [Y/n] Y
```

```
Wrote ~/Desktop/example-plugin/src/main/resources/META-INF/forge.xml
***SUCCESS*** Created project [example-plugin] in new working directory [~/Desktop/example-plugin]
[example-plugin] example-plugin $
[example-plugin] example-plugin $
[example-plugin] example-plugin $ project install-facet forge.api
The [forge.api] facet depends on the following missing facet(s): [forge.spec.cdi]. Install as well? [Y/n] Y
Wrote ~/Desktop/example-plugin/src/main/resources/META-INF/beans.xml
***SUCCESS*** Installed [forge.spec.cdi] successfully.
Install which version of the Forge API?

  1 - [org.jboss.seam.forge:forge-shell-api:1.0.0-SNAPSHOT]
  2 - [org.jboss.seam.forge:forge-shell-api:1.0.0.Alpha2]

Choose an option by typing the number of the selection: 1
***SUCCESS*** Installed [forge.api] successfully.
[example-plugin] example-plugin $
```

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:

NOTE: You must also create a `beans.xml` file in the `src/main/resources/META-INF/` directory of your project, or your plugin will not be detected by Forge.

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

The first thing you must do in order to create a forge plugin, is create a new class and implement the `org.jboss.seam.forge.shell.plugins.Plugin` interface. Notice that the interface has no methods, this is because you will be adding your own custom commands later.

```
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. 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("...")`, in which case 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.

3.4.1. 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 ) {
    out.println(">> invoked default command with option value: " + opt);
    // this method will be invoked, and 'opt' will be passed from the command line
    // 'out' is your handle to this plugin's output stream.
  }
}
```

```
$ exampleplugin some-input
```

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

3.4.2. Named commands

Named commands must, to little surprise, be given a name with which they are invoked. This is done by placing the `@Command("...")` 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("perform")
  public void exampleCommand( @Option 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.

3.5. 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.

3.5.1. --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("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:

```
$ exampleplugin 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:

```
$ exampleplugin perform --one cat --two dog
$ exampleplugin perform --two dog --one cat
$ exampleplugin perform --two dog -o cat
```

3.5.2. 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("perform")
  public void exampleCommand(
              @Option String one,
              @Option 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:

```
$ exampleplugin perform cat dog
>> option one equals: cat
>> option two equals: dog
```

3.5.3. 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("perform")
  public void exampleCommand(
              @Option(name="one") String one,
              @Option(name="two") String two,
              @Option String three,
              @Option String four,
              PipeOut out) {
    out.println(">> option one equals: " + one);
    out.println(">> option two equals: " + two);
    out.println(">> option three equals: " + three);
    out.println(">> option four equals: " + four);
  }
}
```

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

```
$ exampleplugin perform --one cat --two dog bird lizard
>> option one equals: cat
>> option two equals: dog
>> option three equals: bird
>> option four 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:

```
$ exampleplugin --two dog bird --one cat lizard
>> option one equals: cat
>> option two equals: dog
>> option three equals: bird
>> option four equals: lizard
```

3.6. 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 `@PipeIn InputStream stream` or `@PipeIn String in` 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
```

Example 3.2. @PipeIn InputStream stream

```
@Command("example-command")
public void exampleCommand(
        @PipeIn final InputStream in,
        @Option(required = false) final boolean option,
        PipeOut out)
{ ... }
```

Example 3.3. @PipeIn String in

```
@Command("example-command")
public void exampleCommand(
        @PipeIn final String in,
        @Option(required = false) final boolean option,
        PipeOut out)
{ ... }
```

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 `InputStream`s 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;
        }
      }
    }
}
```

3.7. Ensure all referenced libraries are on the CLASSPATH

If your Plugin depends on classes or libraries 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.

Warning

NOTE: This is required even if dependencies are specified directly in your pom.xml file. Transitive dependencies WILL NOT BE INSTALLED with your plugin; they must be packaged via shade.

Example 3.4. Using Forge to set up Shading

```
[example-plugin] example-plugin $ shade setup
***SUCCESS*** Shade plugin is installed.
[example-plugin] example-plugin $
[example-plugin] example-plugin $ shade include commons-collections:commons-collections:3.2.1
```

Notice that the pom.xml file has been modified and now includes a shade configuration including commons-collections.

```
<plugin>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <artifactSet>
          <includes>
            <include>commons-collections:commons-collections</include>
          </includes>
        </artifactSet>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Warning

Do NOT include Forge provided libraries with shade, or you will very likely create a non-functional plugin.

It it also recommended, however, to relocate shaded class files to a new package. Your code will access the bundled code at this new location, and will prevent version conflicts if another plugin bundles a different version of the same library. For this, we use the following command.

```
$ shade relocate --pattern {ORIGINAL PGK} --shadedPattern {NEW PKG} --excludes {EXCLUDED PKGS...}
```

For the purposes of this example, let us assume that our Plugin depends on the Apache Commons Collections library (`org.apache.commons.collections`), and we want to make sure that no conflicts occur.

```
[example-plugin] example-plugin $ shade relocate --pattern org.apache.commons.collections --shadedPattern ~.shaded.apache.collections
***SUCCESS*** Relocating [org.apache.commons.collections] to [com.example.forge.plugin.shaded.apache.collections]
```

```
[example-plugin] example-plugin $
```

This should be repeated for each dependency as necessary. Notice that our POM has been updated with the configuration:

```
<plugin>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <artifactSet>
          <includes>
            <include>commons-collections:commons-collections</include>
          </includes>
        </artifactSet>
        <relocations>
          <relocation>
            <pattern>org.apache.commons.collections</pattern>
          <shadedPattern>com.example.forge.plugin.shaded.apache.collections</shadedPattern>
          </relocation>
        </relocations>
      </configuration>
    </execution>
  </executions>
</plugin>
```

When you build your plugin, you should see confirmation output from the Maven Shade Plugin looking something like this:

```
[INFO] --- maven-shade-plugin:1.4:shade (default) @ example-plugin ---
[INFO] Excluding org.jboss.seam.forge:forge-shell-api:jar:{version} from the shaded jar.
[INFO] Excluding org.jboss.seam.forge:forge-parser-java-api:jar:{version} from the shaded jar.
[INFO] Excluding org.jboss.seam.forge:forge-parser-xml:jar:{version} from the shaded jar.
[INFO] Excluding org.jboss.shrinkwrap.descriptors:shrinkwrap-descriptors-api:jar:{version} from
 the shaded jar.
[INFO] Excluding javax.enterprise:cdi-api:jar:{version} from the shaded jar.
```

```
[INFO] Excluding org.jboss.spec.javax.interceptor:jboss-interceptors-api:jar:{version} from the shaded jar.
[INFO] Excluding javax.annotation:jsr250-api:jar:{version} from the shaded jar.
[INFO] Excluding javax.inject:javax.inject:jar:{version} from the shaded jar.

[INFO] Including commons-collections:commons-collections:jar:3.2.1 in the shaded jar.

[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing ~/Desktop/example-plugin/target/example-plugin-1.0.0-SNAPSHOT.jar
  with ~/Desktop/example-plugin/target/example-plugin-1.0.0-SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

3.8. 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. There are several methods for installing and distributing your plugin; these methods are described in this section.

All plugin installation should take place using the '`$ forge`' meta-command. For more information on this command, type:

```
$ help forge
```

Tip

After installation, Forge automatically hot-loads plugin classes and makes them available for use. There is no need to restart Forge when installing or updating plugins.

3.8.1. As local source files (for development)

Perhaps the simplest form of plugin installation is when the plugin source files are stored locally in a project on the local computer. If a plugin project is already checked out locally, it may be built and installed using the following command:

```
$ forge source-plugin {PATH}
```

Example 3.5. Installing a local plugin project

```
[no project] Desktop $ forge source-plugin ~/Desktop/example-plugin/
***INFO*** Invoking build with underlying build system.
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building example-plugin 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] Compiling 1 source file to ~/Desktop/example-plugin/target/classes
[INFO]

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
There are no tests to run.

[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ example-plugin ---
[INFO] Building jar: /home/lb3/Desktop/example-plugin/target/example-plugin-1.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-shade-plugin:1.4:shade (default) @ example-plugin ---
[INFO] Excluding org.jboss.seam.forge:forge-shell-api:jar:1.0.0-SNAPSHOT from the shaded jar.
[INFO] Excluding org.jboss.seam.forge:forge-parser-java-api:jar:1.0.0-SNAPSHOT from the shaded jar.
[INFO] Excluding org.jboss.seam.forge:forge-parser-xml:jar:1.0.0-SNAPSHOT from the shaded jar.
[INFO] Excluding org.jboss.shrinkwrap.descriptors:shrinkwrap-descriptors-api:jar:0.1.4 from the shaded jar.
[INFO] Excluding javax.enterprise:cdi-api:jar:1.0-SP4 from the shaded jar.
[INFO] Excluding org.jboss.spec.javax.interceptor:jboss-interceptors-api_1.1_spec:jar:1.0.0.Beta1 from the shaded jar.
[INFO] Excluding javax.annotation:jsr250-api:jar:1.0 from the shaded jar.
[INFO] Excluding javax.inject:javax.inject:jar:1 from the shaded jar.
[INFO] Including commons-collections:commons-collections:jar:3.2.1 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing /home/lb3/Desktop/example-plugin/target/example-plugin-1.0.0-SNAPSHOT.jar with /home/lb3/Desktop/example-plugin/target/example-plugin-1.0.0-SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

```
***SUCCESS*** Build successful.
```

If you are installing an updated version of an existing plugin, Forge will prompt for confirmation to continue installation.

```
An existing version of this plugin was found. Replace it? [Y/n] y
***INFO*** Installing plugin artifact.
Wrote ~/.forge/plugins/com.example.forge.plugin$example-plugin$1$1.0.0-SNAPSHOT.jar
***SUCCESS*** Installed from [example-plugin] successfully.
[no project] Desktop $
```

Your plugin is now installed and ready to use:

```
$ exampleplugin perform --one cat --two dog bird lizard
>> option one equals: cat
>> option two equals: dog
>> option three equals: bird
>> option four equals: lizard
```

3.8.2. As a git repository

Plugins may be installed directly from a Git repository. This feature simply automates the source code checkout and installation, much like the *$ forge source-plugin* command. Your plugin will be built and installed while Forge runs.

```
$ forge git-plugin git://github.com/username/repository.git --ref 1.0.0.Alpha1
```

Notice that a --ref may be specified if a specific branch or tag should be built. This is recommended, since snapshots are more likelty to contain unstable code.

3.8.3. As a Maven artifact

One of the most convenient methods for installing plugins is via Maven.

```
$ forge mvn-plugin com.example:example-plugin:{version}
```

Be sure to check the identifier thoroughly before installing. If a {version} is not specified, Forge will allow the user to choose from a list of available plugin versions. The version selected will be installed while Forge runs.

3.8.4. As a local distributable JAR file

In addition to automated installation of plugin projects on the local disk, Forge is also able to install a JAR file directly as a plugin; this JAR is installed while Forge runs.

```
$ forge jar-plugin ~/Downloads/plugin.jar --id com.example:example-plugin
```

Notice that you must specify a Dependency ID for the plugin. Forge uses these identifiers to track installed versions of plugins. This means that if you use an incorrect identifier and attempt to ugprade the plugin, Forge may not detect the duplicate version. If this occurs, it could prevent Forge from running, in which case, a manual cleanup of plugin files is required.

It is not recommended to install plugins directly as JAR files unless you are certain that the content of the JAR file may be trusted.

Tip

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

3.8.5. As a remote distributable JAR file

In addition to automated installation of plugin projects on the local disk, Forge is also able to install a JAR file from a URL; this JAR is installed while Forge runs.

```
$ forge url-plugin http://example.com/forge/plugin.jar --id com.example:example-plugin
```

Notice that you must specify a Dependency ID for the plugin. Forge uses these identifiers to track installed versions of plugins. This means that if you use an incorrect identifier and attempt to ugprade the plugin, Forge may not detect the duplicate version. If this occurs, it could prevent Forge from running, in which case, a manual cleanup of plugin files is required.

It is not recommended to install plugins directly as JAR files unless you are certain that the content of the JAR file may be trusted.

Tip

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

3.9. Removing a plugin from Forge

In order to remove plugins, you must shut down Forge and delete the plugin files from the forge plugin directory (usually `~/.forge/plugins/`). To see a list of which plugins are installed, and where they are stored, type the following command:

```
[no project] Desktop $ forge list-plugins --all

[installed plugins]
org.jboss.errai.forge : forge-errai : 1.0.0-SNAPSHOT
com.ocpsoft.forge.prettyfaces : prettyfaces-forge-plugin : 1.0.0.Alpha2
com.example.forge.plugin : example-plugin : 1.0.0-SNAPSHOT

[no project] Desktop $
```

Appendix A. Affiliation

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