Spring Shell Documentation

Mark Pollack
Costin Leau
Jarred Li

Spring Shell Documentation

by Mark Pollack, Costin Leau, and Jarred Li

1.2.0.M1-mapr-1607

Table of Contents

Preface

The Spring Shell provides an interactive shell that allows you to plugin your own custom commands using a Spring based programming model.

The shell has been extracted from the Spring Roo project, giving it a strong foundation and rich feature set. One significant change from Spring Roo is that the plugin model is no longer based on OSGi but instead uses the Spring IoC container to discover commands through classpath scanning. There is currently no classloader isolation between plugins, however that maybe added in future versions.

Spring Shell's features include

- A simple, annotation driven, programming model to contribute custom commands
- Use of Spring's classpath scanning functionality as the basis for a command plugin strategy and command development
- Inheritance of the Roo Shell features, most notably tab completion, colorization, and script execution.
- Customization of command prompt, banner, shell history file name.

This document assumes that the reader already has a basic familiarity with the Spring Framework.

While every effort has been made to ensure that this documentation is comprehensive and there are no errors, nevertheless some topics might require more explanation and some typos might have crept in. If you do spot any mistakes or even more serious errors and you can spare a few cycles during lunch, please do bring the error to the attention of the Spring Shell team by raising an issue.

Part I. Introduction

The Spring Shell provides an interactive shell that lets you contribute commands using a simple Spring based programming model.

This document is the reference guide for the Spring Shell and covers the key classes that are part of the Shell infrastructure, the plugin model, how to create commands for the shell as well as discussion of the sample application.

1. Requirements

The Spring Shell requires JDK level 6.0 and above as well as the Spring Framework 3.0 (3.1 recommended) and above.

Part II. Reference Documentation

This part of the reference documentation explains the core components of the Spring Shell.

2. Spring Shell

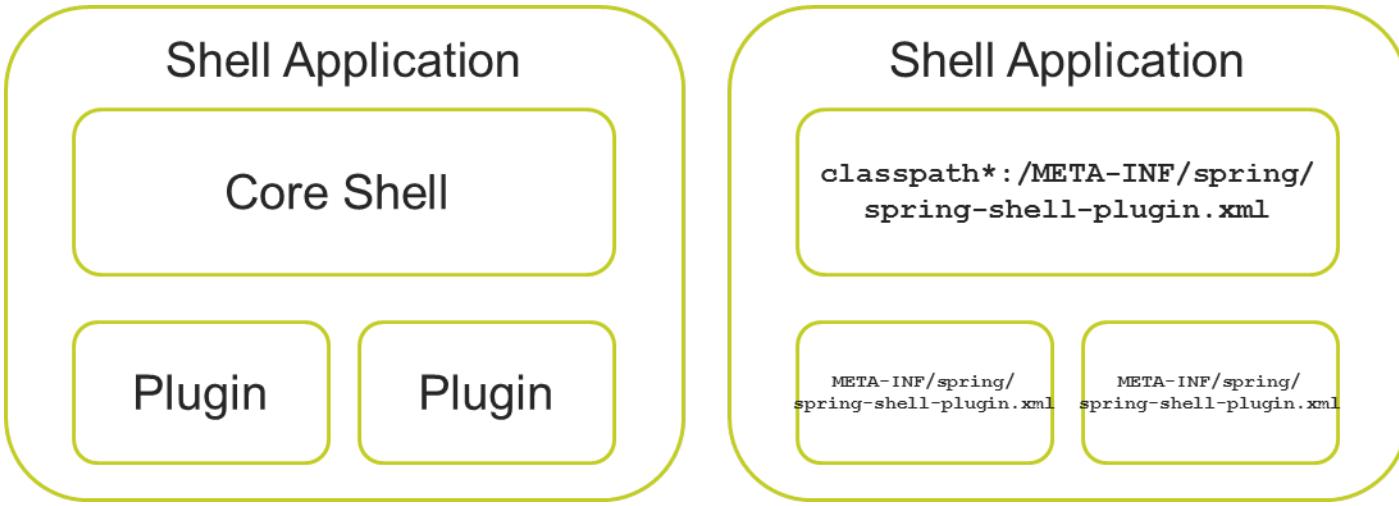
The core components of the shell are its plugin model, built-in commands, and converters.

2.1 Plugin Model

The plugin model is based on Spring. Each plugin jar is required to contain the file `META-INF/spring/spring-shell-plugin.xml`. These files will be loaded to bootstrap a Spring `ApplicationContext` when the shell is started. The essential boostrapping code that looks for your contributions looks like this:

```
new ClassPathXmlApplicationContext("classpath*:/META-INF/spring/spring-shell-plugin.xml");
```

In the `spring-shell-plugin.xml` file you should define the command classes and any other collaborating objects that support the command's actions. The plugin model is depicted in the following diagram



Note that the current plugin model loads all plugins under the same class loader. An open JIRA issue suggests providing a classloader per plugin to provide isolation.

Commands

An easy way to declare the commands is to use Spring's component scanning functionality. Here is an example `spring-shell-plugin.xml` from the sample application:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.1.xsd">

  <context:component-scan
          base-package="org.springframework.shell.samples.helloworld.commands" />

</beans>
```

The commands are Spring components, demarcated as such using the `@Component` annotation. For example, the shell of the `HelloWorldCommands` class from the sample application looks like this

```
@Component
public class HelloWorldCommands implements CommandMarker {

  // use any Spring annotations for Dependency Injection or other Spring
  // interfaces as required.

  // methods with @Cli annotations go here

}
```

Once the commands are registered and instantiated by the Spring container, they are registered with the core shell parser so that the `@Cli` annotations can be processed. The way the commands are identified is through the use of the `CommandMarker` interface.

Converters

The `org.springframework.shell.core.Converter` interface provides the contract to convert the strings that are entered on the command line to rich Java types passed into the arguments of `@Cli`-annotated methods.

By default converters for common types are registered. These cover primitive types (boolean, int, float...) as well as Date, Character, and File.

If you need to register any additional `Converter` instances, register them with the Spring container in the `spring-shell-plugin.xml` file and they will be picked up automatically.

2.2 Built in commands

There are a few built in commands. Here is a listing of their class name and functionality

- `ConsoleCommands` - `clr` and `clear` - to clear the console.
- `DateCommands` - `date` - show the current date and time.
- `ExitCommands` - `exit` and `quit` - to exit the shell.
- `HelpCommands` - `help` - list all commands and their usage
- `InlineCommentCommands` - `//` and `;` shows the valid characters to use for inline comments
- `OsCommands` - the keyword for this command is the exclamation point, `!`. After the exclamation point you can pass in a unix/windows command string to be executed.
- `SystemPropertyCommands` - `system properties`- shows the shell's system properties
- `VersionCommands` - `version`- shows the shell's version

There are two commands in provided by the `AbstractShell` class related to useage of block comments

- `/*` and `*/`- The begin and end characters for block comments

2.3 Customizing the shell

There are a few extension points that allow you to customize the shell. The extension points are the interfaces

- `BannerProvider` - Specifies the banner text, welcome message, and version number that will be displayed when the shell is started
- `PromptProvider` - Specifies the command prompt text, eg. "`shell>`" or "#" or "`$`". This will be called after every command execution so it does not need to be a static string.
- `HistoryFileNameProvider` - Specifies the name of the command history file

There is a default implementation for these interfaces but you should create your own implementations for your own shell application. All of these interfaces extend from `NamedProvider`. Use Spring's `@Order` annotation to set the priority of the provider. This allows your provider implementations to take precedence over any other implementations that maybe present on the classpath from other plugins.

To make cool "ASCII art" banners the website http://patorjk.com/software/taag is quite neat!

2.4 Communicating between plugins

As this is a standard Spring application you can use Spring's `ApplicationContext` event infrastructure to communicate across plugins.

2.5 Command method interception

It has shown to be useful to provide a simple form of interception around the invocation of a command method. This enables the command class to check for updates to state, such as configuration information modified by other plugins, before the command method is executed. The interface `ExecutionProcessor` should be implemented instead of `CommandMarker` to access this functionality. The `ExecutionProcessor` interface is shown below

```
public interface ExecutionProcessor extends CommandMarker {

 /**
  * Method called before invoking the target command (described by {@link ParseResult}).
  * Additionally, for advanced cases, the parse result itself effectively changing the
  * invocation calling site.
  *
  * @param invocationContext target command context
  * @return the invocation target
  */
 ParseResult beforeInvocation(ParseResult invocationContext);

 /**
  * Method called after successfully invoking the target command (described by
  * {@link ParseResult}).
  *
```

```
   * @param invocationContext target command context
   * @param result the invocation result
   */
  void afterReturningInvocation(ParseResult invocationContext, Object result);

  /**
   * Method called after invoking the target command (described by {@link ParseResult})
   * had thrown an exception .
   *
   * @param invocationContext target command context
   * @param thrown the thrown object
   */
  void afterThrowingInvocation(ParseResult invocationContext, Throwable thrown);

}
```

2.6 Command line options

There are a few command line options that can be specified when starting the shell. They are

- `--profiles` - Specifies values for the system property spring.profiles.active so that Spring 3.1 and greater profile support is enabled.
- `--cmdfile` - Specifies a file to read that contains shell commands
- `--histsize` - Specifies the maximum number of lines to store in the command history file. Default value is 3000.
- `--disableInternalCommands` - Flag that disables all commands that would be pre-registered with the shell. There is no argument to this option. You can selectively add back any internal commands by referencing them in your shell plugin file. Look at the Spring Shell javadocs for specific commands located in the `org.springframework.shell.commands` package as well as the section in this documentation of Built in commands.

2.7 Scripts and comments

Scripts can be executed either by passing in the `--cmdfile` argument at startup or by executing the `script` command inside the shell. When using scripts it helps to add comments and this can be done using block comments that start and end with `/*` and `*/` or an inline one line comment using the `//` or `;` characters.

Part III. Developing Spring Shell Applications

This section provides some guidance on how one can create commands for the Spring Shell.

3. Developing Spring Shell Applications

Contributing commands to the shell is very easy. There are only a few annotations you need to learn. The implementation style of the command is the same as developing classes for an application that uses dependency injection. You can leverage all the features of the Spring container to implement your command classes.

3.1 Marker Interface

The first step to creating a command is to implement the marker interface `CommandMarker` and to annotate your class with Spring's `@Component` annotation. (Note there is an open JIRA issue to provide a `@CliCommand` meta-annotation to avoid having to use a marker interface). Using the code from the helloworld sample application, the code of a `HelloWorldCommands` class is shown below:

```
@Component
public class HelloWorldCommands implements CommandMarker {

  // use any Spring annotations for Dependency Injection or other Spring interfaces
  // as required.

  // methods with @Cli annotations go here

}
```

3.2 Logging

Logging is currently done using JDK logging. Due to the intricacies of console, JLine and Ansi handling, it is generally advised to display messages as return values to the method commands. However, when logging is required, the typical JDK logger declaration should suffice.

```
@Component
public class HelloWorldCommands implements CommandMarker {

  protected final Logger LOG = Logger.getLogger(getClass().getName());

  // methods with @Cli annotations go here

}
```

Warning

Note: it is the responsibility of the packager/developer to handle logging for third-party libraries. Typically one wants to reduce the logging level so the console/shell does not get affected by logging messages.

3.3 CLI Annotations

There are three annotations used on methods and method arguments that define the main contract for interacting with the shell. These are:

- `CliAvailabilityIndicator` - Placed on a method that returns a boolean value and indicates if a particular command can be presented in the shell. This decision is usually based on the history of commands that have been executed previously. It prevents extraneous commands being presented until some preconditions are met, for example the execution of a 'configuration' command.

- `CliCommand` - Placed on a method that provides a command to the shell. Its value provides one or more strings that serve as the start of a particular command name. These must be unique within the entire application, across all plugins.

- `CliOption` - Placed on the arguments of a command method, allowing it to declare the argument value as mandatory or optional with a default value.

Here is a simple use of these annotations in a command class

```
@Component
public class HelloWorldCommands implements CommandMarker {

  @CliAvailabilityIndicator({"hw simple"})
  public boolean isCommandAvailable() {
    return true;
  }

  @CliCommand(value = "hw simple", help = "Print a simple hello world message")
  public String simple(
    @CliOption(key = { "message" }, mandatory = true, help = "The hello world message")
    final String message,

    @CliOption(key = { "location" }, mandatory = false,
               help = "Where you are saying hello", specifiedDefaultValue="At work")
    final String location) {

    return "Message = [" + message + "] Location = [" + location + "]";

  }
}
```

The method annotated with `@CliAvailabilityIndicator` is returning true so that the one and only command in this class is exposed to the shell to be invoked. If there were more commands in the class, you would list them as comma separated value.

The `@CliCommand` annotation is creating the command '`hw simple`' in the shell. The help message is what will be printed if you use the build in `help` command. The method name is '`simple`' but it could just have been any other name.

The `@CliOption` annotation on each of the command arguments is where you will spend most of your time authoring commands. You need to decide which arguments are required, which are optional, and if they are optional is there a default value. In this case there are two arguments or options to the command: message and location. The message option is required and a help message is provided to give guidance to the user when tabbing to get completion for the command.

The implementation of the '`simple`' method is trivial, just a log statement, but this is where you would typically call other collaborating objects that were injected into the class via Spring.

The method argument types in this example are `String`, which doesn't present any issue with type conversion. You can specify methods with any rich object type as well as basic primitive types such as int, float etc. For all types other than those handled by the shell by default (basic types, `Date`, `File`) you will need to register your own implementation of the `org.springframework.shell.core.Converter` interface with the container in your plugin.

Note that the method return argument can be non-void - in our example, it is the actual message we want to display. Whenever an object is returned, the shell will display its `toString()` representation.

3.4 Testing shell commands

To perform a test of the shell commands you can instantiate the shell inside a test case, execute the command and then perform assertions on the return value CommandResult. A simple base class to set this up is shown below.

```
public abstract class AbstractShellIntegrationTest {

 private static JLineShellComponent shell;

 @BeforeClass
 public static void startUp() throws InterruptedException {
  Bootstrap bootstrap = new Bootstrap();
  shell = bootstrap.getJLineShellComponent();
 }

 @AfterClass
 public static void shutdown() {
  shell.stop();
 }

 public static JLineShellComponent getShell() {
  return shell;
 }

}
```

Here is an example testing the Date command

```
public class BuiltInCommandTests extends AbstractShellIntegrationTest {

 @Test
 public void dateTest() throws ParseException {

  //Execute command
  CommandResult cr = getShell().executeCommand("date");

  //Get result
  DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL,Locale.US);
  Date result = df.parse(cr.getResult().toString());

  //Make assertions - DateMaters is an external dependency not shown here.
  Date now = new Date();
  MatcherAssert.assertThat(now, DateMatchers.within(5, TimeUnit.SECONDS, result));
 }
}
```

The java.lang.Class of CommandResult's getResult method will match that of the return value of the method annotated with @CliCommand. You should cast to the appropriate type to help perform your assertions.

3.5 Building and running the shell

In our opinion, the easiest way to build and execute the shell is to cut-n-paste the gradle script in the example application. This uses the application plugin from gradle to create a bin directory with a startup script for windows and Unix and places all dependent jars in a lib directory. Maven has a similar plugin - the AppAssembler plugin.

The main class of the shell is `org.springframework.shell.Bootstrap`. As long as you place other plugins, perhaps developed independently, on the classpath, the Bootstrap class will incorporate them into the shell.

Part IV. Spring Shell Sample application

This part of the reference documentation covers the sample applications included with Spring Shell that demonstrate the features in a code centric manner.

Chapter 4, *Simple sample application using the Spring Shell*Describes a simple Spring Shell application that echo's the command parameters to the console.

4. Simple sample application using the Spring Shell

4.1 Introduction

The sample application named 'helloworld' contains three '`hw`' commands, they are '`hw simple`', '`hw complex`' and '`hw enum`' and demonstrate simple to intermediate level usage of the `@Cli` annotation classes for creating commands.

The example code is located in the distribution directory `<spring-shell-install-dir>/samples/helloworld`.

To build the example cd to the helloworld directory and execute `gradlew installApp` (you first need to add the `gradlew` gradle wrapper to your path). To run the application cd to `build\install\helloworld\bin` and execute the helloworld script.

4.2 HelloWorldCommands

The `HelloWorldCommands` class is show below

```
package org.springframework.shell.samples.helloworld.commands;

import org.springframework.shell.core.CommandMarker;
import org.springframework.shell.core.annotation.CliAvailabilityIndicator;
import org.springframework.shell.core.annotation.CliCommand;
import org.springframework.shell.core.annotation.CliOption;
import org.springframework.stereotype.Component;

@Component
public class HelloWorldCommands implements CommandMarker {

 private boolean simpleCommandExecuted = false;

 @CliAvailabilityIndicator({"hw simple"})
 public boolean isSimpleAvailable() {
  //always available
  return true;
 }

 @CliAvailabilityIndicator({"hw complex", "hw enum"})
 public boolean isComplexAvailable() {
  if (simpleCommandExecuted) {
   return true;
  } else {
   return false;
  }
 }

 @CliCommand(value = "hw simple", help = "Print a simple hello world message")
 public String simple(
  @CliOption(key = { "message" }, mandatory = true, help = "The hello world message") final String message,
  @CliOption(key = { "location" }, mandatory = false, help = "Where you are saying hello", specifiedDefaultV
  simpleCommandExecuted = true;
```

```
  return "Message = [" + message + "] Location = [" + location + "]";
 }

 @CliCommand(value = "hw complex", help = "Print a complex hello world message")
 public String hello(
  @CliOption(key = { "message" }, mandatory = true, help = "The hello world message") final String message,
  @CliOption(key = { "name1"}, mandatory = true, help = "Say hello to the first name") final String name1,
  @CliOption(key = { "name2" }, mandatory = true, help = "Say hello to a second name") final String name2,
  @CliOption(key = { "time" }, mandatory = false, specifiedDefaultValue="now", help = "When you are saying
  @CliOption(key = { "location" }, mandatory = false, help = "Where you are saying hello") final String loca
  return "Hello " + name1 + " and " + name2 + ". Your special message is "  + message + ". time=[" + time +
 }

 @CliCommand(value = "hw enum", help = "Print a simple hello world message from an enumerated value")
 public String eenum(
  @CliOption(key = { "message" }, mandatory = true, help = "The hello world message") final MessageType mess
  return "Hello.  Your special enumerated message is " + message;
 }

 enum MessageType {
  Type1("type1"),
  Type2("type2"),
  Type3("type3");

  private String type;

  private MessageType(String type){
   this.type = type;
  }

  public String getType(){
   return type;
  }
 }
}
```

The use of the `@CliAvailabilityIndicator` annotation on two methods, `isSimpleAvailable` and `isComplexAvailable` shows how you can enable the presence of the '`hw complex`' and '`hw enum`' commands only if the '`hw simple`' command was executed.

Here is an example session showing the behavior.

```
X:\projects\spring-shell\samples\helloworld\build\install\
=========================================
*                                       *
*            HelloWorld                 *
*                                       *
=========================================
Verson:1.0.1
Welcome to HelloWorld CLI
hw-shell>

!          */         /*         //         date       exi

hw-shell>hw simple --message
hw simple --message
required --message: The hello world message; no default va
hw-shell>hw simple --message yo!
Message = [yo!] Location = [null]
hw-shell>hw

hw complex    hw enum       hw simple

hw-shell>hw complex --

hw complex --message    hw complex --name1      hw complex

hw-shell>hw complex --message yo! --name

hw complex --message yo! --name1    hw complex --message y

hw-shell>hw complex --message yo! --name1 Joe
hw complex --message yo! --name1 Joe
required --name1: Say hello to the first name; no default
hw-shell>hw complex --message yo! --name1 Joe --name2 Mary
Hello Joe and Mary. Your special message is yo!. time=[nul
hw-shell>hw complex --message yo! --name1 Joe --name2 Mary

hw complex --message yo! --name1 Joe --name2 Mary --locati

hw-shell>hw complex --message yo! --name1 Joe --name2 Mary
Hello Joe and Mary. Your special message is yo!. time=[nul
hw-shell>hw complex --message yo! --name1 Joe --name2 Mary
Hello Joe and Mary. Your special message is yo!. time=[12:
hw-shell>
```

The '`hw enum`' command shows how the shell supports the use of an enumeration as command method arguments.

```
hw-shell>hw enum --message
hw enum --message
required --message: The hello world message; no default va
hw-shell>hw enum --message Type

Type1   Type2   Type3

hw-shell>hw enum --message Type2
Hello.  You special enumerated message is Type2
hw-shell>_
```