Spring Shell Reference Documentation

Eric Bottard

Table of Contents

Version 2.0.0.RELEASE

What is Spring Shell?

Not all applications need a fancy web user interface! Sometimes, interacting with an application using an interactive terminal is the most appropriate way to get things done.

Spring Shell allows one to easily create such a runnable application, where the user will enter textual commands that will get executed until the program terminates. The Spring Shell project provides the infrastructure to create such a REPL (Read, Eval, Print Loop), allowing the developer to concentrate on the commands implementation, using the familiar Spring programming model.

Advanced features such as parsing, `TAB` completion, colorization of output, fancy ascii-art table display, input conversion and validation all come for free, with the developer only having to focus on core command logic.

Using Spring Shell

Getting Started

To see what Spring Shell has to offer, let's write a trivial shell application that has a simple command to add two numbers together.

Let's Write a Simple Boot App

Starting with version 2, Spring Shell has been rewritten from the ground up with various enhancements in mind, one of which is easy integration with Spring Boot, although it is not a strong requirement. For the purpose of this tutorial, let's create a simple Boot application, for example using start.spring.io. This minimal application only depends on `spring-boot-starter` and configures the `spring-boot-maven-plugin`, generating an executable über-jar:

```
...
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    ...
```

Adding a Dependency on Spring Shell

The easiest way to get going with Spring Shell is to depend on the `spring-shell-starter` artifact. This comes with everything one needs to use Spring Shell and plays nicely with Boot, configuring only the necessary beans as needed:

```
...
<dependency>
    <groupId>org.springframework.shell</groupId>
    <artifactId>spring-shell-starter</artifactId>
    <version>2.0.0.RELEASE</version>
</dependency>
...
```

Given that Spring Shell will kick in and start the REPL by virtue of this dependency being present, you'll need to either build skipping tests (`-DskipTests`) throughout this tutorial or remove the sample integration test that was generated by start.spring.io. If you don't do so, the integration test will create the Spring `ApplicationContext` and, depending on your build tool, will stay stuck in the eval loop or crash with a NPE.

Your first command

It's time to add our first command. Create a new class (name it however you want) and annotate it with `@ShellComponent` (a variation of `@Component` that is used to restrict the set of classes that are scanned for candidate commands).

Then, create an `add` method that takes two ints (`a` and `b`) and returns their sum. Annotate it with `@ShellMethod` and provide a description of the command in the annotation (the only piece of information that is required):

```
package com.example.demo;

import org.springframework.shell.standard.ShellMethod;
import org.springframework.shell.standard.ShellComponent;

@ShellComponent
public class MyCommands {

    @ShellMethod("Add two integers together.")
    public int add(int a, int b) {
        return a + b;
    }
}
```

Let's Give It a Ride!

Build the application and run the generated jar, like so;

```
./mvnw clean install -DskipTests
[...]

java -jar target/demo-0.0.1-SNAPSHOT.jar
```

You'll be greeted by the following screen (the banner comes from Spring Boot, and can be customized as usual):

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.6.RELEASE)

shell:>
```

Below is a yellow `shell:>` prompt that invites you to type commands. Type `add 1 2` then `ENTER` and

admire the magic!

```
shell:>add 1 2
3
```

Try to play with the shell (hint: there is a `help` command) and when you're done, type `exit ENTER`.

The rest of this document delves deeper into the whole Spring Shell programming model.

Writing your own Commands

The way Spring Shell decides to turn a method into an actual shell command is entirely pluggable (see Extending Spring Shell), but as of Spring Shell 2.x, the recommended way to write commands is to use the new API described in this section (the so-called *standard* API).

Using the *standard* API, methods on beans will be turned into executable commands provided that

- the bean class bears the `@ShellComponent` annotation. This is used to restrict the set of beans that are considered.
- the method bears the `@ShellMethod` annotation.

The `@ShellComponent` is a stereotype annotation itself meta-annotated with `@Component`. As such, it can be used in addition to the filtering mechanism to also *declare* beans (*e.g.* using `@ComponentScan`).

The name of the created bean can be customized using the `value` attribute of the annotation.

It's all about Documentation!

The only required attribute of the `@ShellMethod` annotation is its `value` attribute, which should be used to write a short, one-sentence, description of what the command does. This is important so that your users can get consistent help about your commands without having to leave the shell (see Integrated Documentation with the `help` Command).

The description of your command should be short, one or two sentences only. For better consistency, it is recommended that it starts with a capital letter and ends with a dot.

Customizing the Command Name(s)

By default, there is no need to specify the *key* for your command (*i.e.* the word(s) that should be used to invoke it in the shell). The name of the method will be used as the command key, turning camelCase names into dashed, gnu-style, names (that is, `sayHello()` will become `say-hello`).

It is possible, however, to explicitly set the command key, using the `key` attribute of the annotation, like so:

```
@ShellMethod(value = "Add numbers.", key = "sum")
public int add(int a, int b) {
    return a + b;
}
```

The `key` attribute accepts multiple values. If you set multiple keys for a single method, then the command will be registered using those different aliases.

The command key can contain pretty much any character, including spaces. When coming up with names though, keep in mind that consistency is often appreciated by users (*i.e.* avoid mixing dashed-names with spaced names, *etc.*)

Invoking your Commands

By Name *vs.* Positional Parameters

As seen above, decorating a method with `@ShellMethod` is the sole requirement for creating a command. When doing so, the user can set the value of all method parameters in two possible ways:

- using a parameter key (*e.g.* `--arg value`). This approach is called "by name" parameters
- or without a key, simply setting parameter values in the same order they appear in the method signature ("positional" parameters).

These two approaches can be mixed and matched, with named parameters always taking precedence (as they are less prone to ambiguity). As such, given the following command

```
@ShellMethod("Display stuff.")
public String echo(int a, int b, int c) {
    return String.format("You said a=%d, b=%d, c=%d", a, b, c);
}
```

then the following invocations are all equivalent, as witnessed by the output:

```
shell:>echo 1 2 3              ①
You said a=1, b=2, c=3

shell:>echo --a 1 --b 2 --c 3  ②
You said a=1, b=2, c=3

shell:>echo --b 2 --c 3 --a 1  ③
You said a=1, b=2, c=3

shell:>echo --a 1 2 3          ④
You said a=1, b=2, c=3

shell:>echo 1 --c 3 2          ⑤
You said a=1, b=2, c=3
```

① This uses positional parameters

② This is an example of full by-name parameters

③ By-name parameters can be reordered as desired

④ You can use a mix of the two approaches

⑤ The non by-name parameters are resolved in the order they appear

Customizing the Named Parameter Key(s)

As seen above, the default strategy for deriving the key for a named parameter is to use the java name of the method signature and prefixing it with two dashes (`--`). This can be customized in two ways:

1. to change the default prefix for the whole method, use the `prefix()` attribute of the `@ShellMethod` annotation
2. to override the *whole* key on a per-parameter fashion, annotate the parameter with the `@ShellOption` annotation.

Have a look at the following example:

```
    @ShellMethod(value = "Display stuff.", prefix="-")
    public String echo(int a, int b, @ShellOption("--third") int c) {
        return String.format("You said a=%d, b=%d, c=%d", a, b, c);
    }
```

For such a setup, the possible parameter keys will be `-a`, `-b` and `--third`.

It is possible to specify several keys for a single parameter. If so, these will be mutually exclusive ways to specify the same parameter (so only one of them can be used). As an example, here is the signature of the built-in `help` command:

```
    @ShellMethod("Describe a command.")
    public String help(@ShellOption({"-C", "--command"}) String
command) {
        ...
    }
```

Optional Parameters and Default Values

Spring Shell provides the ability to give parameters default values, which will allow the user to omit those parameters:

```
    @ShellMethod("Say hello.")
    public String greet(@ShellOption(defaultValue="World") String who) {
        return "Hello " + who;
    }
```

Now, the `greet` command can still be invoked as `greet Mother` (or `greet --who Mother`), but the following is also possible:

```
shell:>greet
Hello World
```

Parameter Arity

Up to now, it has always been assumed that each parameter mapped to a single word entered by the user. Situations may arise though, when a parameter value should be *multi valued*. This is driven by the `arity()` attribute of the `@ShellOption` annotation. Simply use a collection or array for the parameter type, and specify how many values are expected:

```
    @ShellMethod("Add Numbers.")
    public float add(@ShellOption(arity=3) float[] numbers) {
        return numbers[0] + numbers[1] + numbers[2];
    }
```

The command may then be invoked using any of the following syntax:

```
shell:>add 1 2 3.3
6.3
shell:>add --numbers 1 2 3.3
6.3
```

When using the *by-name* parameter approach, the key should **not** be repeated. The following does **not** work:

```
shell:>add --numbers 1 --numbers 2 --numbers 3.3
```

Infinite Arity

TO BE IMPLEMENTED

Special Handling of Boolean Parameters

When it comes to parameter arity, there is a kind of parameters that receives a special treatment by default, as is often the case in command-line utilities. Boolean (that is, `boolean` as well as `java.lang.Boolean`) parameters behave like they have an `arity()` of `0` by default, allowing users to set their values using a "flag" approach. Take a look at the following:

```
    @ShellMethod("Terminate the system.")
    public String shutdown(boolean force) {
        return "You said " + force;
    }
```

This allows the following invocations:

```
shell:>shutdown
You said false
shell:>shutdown --force
You said true
```

This special treatment plays well with the default value specification. Although the default for boolean parameters is to have their default value be `false`, you can specify otherwise (*i.e.* `@ShellOption(defaultValue="true")`) and the behavior will be inverted (that is, not specifying the parameter will result in the value being `true`, and specifying the flag will result in the value being `false`)

Having this behavior of implicit `arity()=0` prevents the user from specifying a value (*e.g.* `shutdown --force true`). If you would like to allow this behavior (and forego the flag approach), then force an arity of `1` using the annotation:

```
    @ShellMethod("Terminate the system.")
    public String shutdown(@ShellOption(arity=1, defaultValue="false")
boolean force) {
        return "You said " + force;
    }
```

Quotes Handling

Spring Shell takes user input and tokenizes it in *words*, splitting on space characters. If the user wants to provide a parameter value that contains spaces, that value needs to be quoted. Both single (') and double (") quotes are supported, and those quotes will not be part of the value:

```
    @ShellMethod("Prints what has been entered.")
    public String echo(String what) {
        return "You said " + what;
    }
```

```
shell:>echo Hello
You said Hello
shell:>echo 'Hello'
You said Hello
shell:>echo 'Hello World'
You said Hello World
shell:>echo "Hello World"
You said Hello World
```

Supporting both single and double quotes allows the user to easily embed one type of quotes into a value:

```
shell:>echo "I'm here!"
You said I'm here!
shell:>echo 'He said "Hi!"'
You said He said "Hi!"
```

Should the user need to embed the same kind of quote that was used to quote the whole parameter, the escape sequence uses the backslash (\) character:

```
shell:>echo 'I\'m here!'
You said I'm here!
shell:>echo "He said \"Hi!\""
You said He said "Hi!"
shell:>echo I\'m here!
You said I'm here!
```

It is also possible to escape space characters when not using enclosing quotes, as such:

```
shell:>echo This\ is\ a\ single\ value
You said This is a single value
```

Interacting with the Shell

The Spring Shell project builds on top of the JLine library, and as such brings a lot of nice interactive features, some of which are detailed in this section.

First and foremost, Spring Shell supports `TAB` completion almost everywhere possible. So if there is an `echo` command and the user presses `e`, `c`, `TAB` then `echo` will appear. Should there be several commands that start with `ec`, then the user will be prompted to choose (using `TAB` or `Shift+TAB` to navigate, and `ENTER` for selection.)

But completion does not stop at command keys. It also works for parameter keys (`--arg`) and even parameter values, if the application developer registered the appropriate beans (see Providing TAB Completion Proposals).

Another nice feature of Spring Shell apps is support for line continuation. If a command and its parameters is too long and does not fit nicely on screen, a user may chunk it and terminate a line with a backslash (`\`) character then hit `ENTER` and continue on the next line. Uppon submission of the whole command, this will be parsed as if the user entered a single space on line breaks.

```
shell:>register module --type source --name foo  \ ①
> --uri file:///tmp/bar
Successfully registered module 'source:foo'
```

① command continues on next line

Line continuation also automatically triggers if the user has opened a quote (see Quotes Handling) and hits `ENTER` while still in the quotes:

```
shell:>echo "Hello ①
dquote> World"
You said Hello World
```

① user presses `ENTER` here

Lastly, Spring Shell apps benefit from a lot of keyboard shortcuts you may already be familiar with when working with your regular OS Shell, borrowed from Emacs. Notable shortcuts include `Ctrl+r` to perform a reverse search, `Ctrl+a` and `Ctrl+e` to move to beginning and end of line respectively or `Esc f` and `Esc b` to move forward (*resp.* backward) one word at a time.

Providing TAB Completion Proposals

TBD

Validating Command Arguments

Spring Shell integrates with the Bean Validation API to support automatic and self documenting constraints on command parameters.

Annotations found on command parameters as well as annotations at the method level will be honored and trigger validation prior to the command executing. Given the following command:

```
    @ShellMethod("Change password.")
    public String changePassword(@Size(min = 8, max = 40) String password) {
        return "Password successfully set to " + password;
    }
```

You'll get this behavior, for free:

```
shell:>change-password hello
The following constraints were not met:
    --password string : size must be between 8 and 40 (You passed 'hello')
```

Applies to All Command Implementations

It is important to note that bean validation applies to all command implementations, whether they use the "standard" API or any other API, through the use of an adapter (see Supporting Other APIs)

Dynamic Command Availability

There may be times when registered commands don't make sense, due to internal state of the application. For example, maybe there is a `download` command, but it only works once the user has used `connect` on a remote server. Now, if the user tries to use the `download` command, the shell should gracefully explain that the command *does* exist, but that it is not available at the time. Spring Shell lets the developer do that, even providing a short explanation of the reason for the command not being available.

There are three possible ways for a command to indicate availability. They all leverage a no-arg method that returns an instance of `Availability`. Let's start with a simple example:

```
@ShellComponent
public class MyCommands {

    private boolean connected;

    @ShellMethod("Connect to the server.")
    public void connect(String user, String password) {
        [...]
        connected = true;
    }

    @ShellMethod("Download the nuclear codes.")
    public void download() {
        [...]
    }

    public Availability downloadAvailability() {
        return connected
            ? Availability.available()
            : Availability.unavailable("you are not connected");
    }
}
```

Here you see the `connect` method is used to connect to the server (details omitted), altering state of the command through the `connected` boolean when done. The `download` command will be marked as *unavailable* till the user has connected, thanks to the presence of a method named exactly as the `download` command method with the `Availability` suffix in its name. The method returns an instance of `Availability`, constructed with one of the two factory methods. In case of the command not being available, an explanation has to be provided. Now, if the user tries to invoke the command while not being connected, here is what happens:

```
shell:>download
Command 'download' exists but is not currently available because you are not
connected.
Details of the error have been omitted. You can use the stacktrace command to print
the full stacktrace.
```

Information about currently unavailable commands is also leveraged in the integrated help. See Integrated Documentation with the `help` Command.

The reason provided when the command is not available should read nicely if appended after "Because ..."

It's best not to start the sentence with a capital and not add a final dot.

If for some reason naming the availability method after the name of the command method does not suit you, you can provide an explicit name using the `@ShellMethodAvailability`, like so:

```
@ShellMethod("Download the nuclear codes.")
@ShellMethodAvailability("availabilityCheck") ①
public void download() {
    [...]
}

public Availability availabilityCheck() { ①
    return connected
        ? Availability.available()
        : Availability.unavailable("you are not connected");
}
```

① the names have to match

Lastly, it is often the case that several commands in the same class share the same internal state and thus should all be available or unavailable all at one. Instead of having to stick the `@ShellMethodAvailability` on all command methods, Spring Shell allows the user to flip things around and put the `@ShellMethodAvailabilty` annotation on the availability method, specifying the names of the commands that it controls:

```
@ShellMethod("Download the nuclear codes.")
public void download() {
    [...]
}

@ShellMethod("Disconnect from the server.")
public void disconnect() {
    [...]
}

@ShellMethodAvailability({"download", "disconnect"})
public Availability availabilityCheck() {
    return connected
        ? Availability.available()
        : Availability.unavailable("you are not connected");
}
```

The default value for the `@ShellMethodAvailability.value()` attribute is `"*"` and this serves as a special wildcard that matches all command names. It's thus easy to turn all commands of a single class on or off with a single availability method. Here is an example below:

```
@ShellComponent
public class Toggles {
  @ShellMethodAvailability
  public Availability availabilityOnWeekdays() {
    return Calendar.getInstance().get(DAY_OF_WEEK) == SUNDAY
      ? Availability.available()
      : Availability.unavailable("today is not Sunday");
  }

  @ShellMethod
  public void foo() {}

  @ShellMethod
  public void bar() {}
}
```

Spring Shell does not impose much constraints on how to write commands and how to organize classes. But it's often good practice to put related commands in the same class, and the availability indicators can benefit from that.

Organizing Commands

When your shell starts to provide a lot of functionality, you may en up with a lot of commands, which could be confusing for your users. Typing `help` they would see a daunting list of commands, organized by alphabetical order, which may not always make sense.

To alleviate this, Spring Shell provides the ability to group commands together, with reasonable defaults. Related commands would then end up in the same *group* (*e.g.* `User Management Commands`) and be displayed together in the help screen and other places.

By default, commands will be grouped according to the class they are implemented in, turning the camel case class name into separate words (so `URLRelatedCommands` becomes `URL Related Commands`). This is a very sensible default, as related commands are often already in the class anyway, for they need to use the same collaborating objects.

If however, this behavior does not suit you, you can override the group for a command in the following ways, in order of priority:

- specifying a `group()` in the `@ShellMethod` annotation
- placing a `@ShellCommandGroup` on the class the command is defined in. This will apply the group for all commands defined in that class (unless overridden as above)
- placing a `@ShellCommandGroup` on the package (*via* `package-info.java`) the command is defined in.

This will apply to all commands defined in the package (unless overridden at the method or class level as explained above)

Here is a short example:

```
public class UserCommands {
    @ShellCommand(value = "This command ends up in the 'User Commands' group")
    public void foo() {}

    @ShellCommand(value = "This command ends up in the 'Other Commands' group",
        group = "Other Commands")
    public void bar() {}
}

...

@ShellCommandGroup("Other Commands")
public class SomeCommands {
    @ShellMethod(value = "This one is in 'Other Commands'")
    public void wizz() {}

    @ShellMethod(value = "And this one is 'Yet Another Group'",
        group = "Yet Another Group")
    public void last() {}
}
```

Built-In Commands

Any application built using the `spring-shell-starter` artifact (or, to be more precise, the `spring-shell-standard-commands` dependency) comes with a set of built-in commands. These commands can be overridden or disabled individually (see Overriding or Disabling Built-In Commands), but if they're not, this section describes their behavior.

Integrated Documentation with the `help` Command

Running a shell application often implies that the user is in a graphically limited environment. And although, in the era of mobile phones we're always connected, accessing a web browser or any other rich UI application such as a pdf viewer may not always be possible. This is why it is important that the shell commands are correctly self documented, and this is where the `help` command comes in.

Typing `help` + `ENTER` will list all the known commands to the shell (including unavailable commands) and a short description of what they do:

```
shell:>help
AVAILABLE COMMANDS
        add: Add numbers together.
      * authenticate: Authenticate with the system.
      * blow-up: Blow Everything up.
        clear: Clear the shell screen.
        connect: Connect to the system
        disconnect: Disconnect from the system.
        exit, quit: Exit the shell.
        help: Display help about available commands.
        register module: Register a new module.
        script: Read and execute commands from a file.
        stacktrace: Display the full stacktrace of the last error.

Commands marked with (*) are currently unavailable.
Type `help <command>` to learn more.
```

Typing `help <command>` will display more detailed information about a command, including the available parameters, their type and whether they are mandatory or not, *etc.*

Here is the `help` command applied to itself:

```
shell:>help help


NAME
    help - Display help about available commands.

SYNOPSYS
    help [[-C] string]

OPTIONS
    -C or --command  string
        The command to obtain help for.  [Optional, default = <none>]
```

Clearing the Screen

The `clear` command does what you would expect and clears the screen, resetting the prompt in the top left corner.

Exitting the Shell

The `quit` command (also aliased as `exit`) simply requests the shell to quit, gracefully closing the Spring application context. If not overridden, a JLine `History` bean will write a history of all commands executed to disk, so that they are available again (see Interacting with the Shell) on next launch.

Displaying Details about an Error

When an exception occurs inside command code, it is caught by the shell and a simple, one-line message is displayed so as not to overflow the user with too much information. There are cases though when understanding what exactly happened is important (especially if the exception has a nested cause).

To this purpose, Spring Shell remembers the last exception that occurred and the user can later use the `stacktrace` command to print all the gory details on the console.

Running a Batch of Commands

The `script` command accepts a local file as an argument and will replay commands found there, one at a time.

Reading from the file behaves exactly like inside the interactive shell, so lines starting with `//` will be considered as comments and ignored, while lines ending with `\` will trigger line continuation.

Customizing the Shell

Overriding or Disabling Built-In Commands

Built-in commands are provided with Spring Shell to achieve everyday tasks that many if not all shell applications need. If you're not happy with the way they behave though, you can disable or override them, as explained in this section.

Disabling all Built-in Commands

If you don't need built-in commands at all, then there is an easy way to "disable" them: just don't include them! Either use a maven exclusion on `spring-shell-standard-commands` or, if you're selectively including Spring Shell dependencies, don't bring that one in!

```
<dependency>
    <groupId>org.springframework.shell</groupId>
    <artifactId>spring-shell-starter</artifactId>
    <version>2.0.0.RELEASE</version>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.shell</groupId>
            <artifactId>spring-shell-standard-commands</artifactId>
        </exclusion>
    </exclusion>
</dependency>
```

Disabling Specific Commands

To disable a single built-in command, simply set the `spring.shell.command.<command>.enabled`

property to `false` in the app `Environment`. One easy way to do this is to pass extra args to the Boot application in your `main()` entry point:

```
    public static void main(String[] args) throws Exception {
        String[] disabledCommands = {"--spring.shell.command.help.enabled=false"}; ①
        String[] fullArgs = StringUtils.concatenateStringArrays(args,
disabledCommands);
        SpringApplication.run(MyApp.class, fullArgs);
    }
```

① This disables the integrated `help` command

Overriding Specific Commands

If, instead of disabling a command you'd rather provide your own implementation, then you can either

- disable the command like explained above and have your implementation registered with the same name
- have your implementing class implement the `<Command>.Command` interface. As an example, here is how to override the `clear` command:

```
public class MyClear implements Clear.Command {

    @ShellCommand("Clear the screen, only better.")
    public void clear() {
        // ...
    }
}
```

Please Consider Contributing your Changes

If you feel like your implementation of a standard command could be valuable to the community, please consider opening a pull-request at github.com/spring-projects/spring-shell.

Alternatively, before making any changes on your own, you can open an issue with the project. Feedback is always welcome!

ResultHandlers

PromptProvider

After each command invocation, the shell waits for new input from the user, displaying a *prompt* in yellow:

```
shell:>
```

It is possible to customize this behavior by registering a bean of type `PromptProvider`. Such a bean may use internal state to decide what to display to the user (it may for example react to application events) and can use JLine's `AttributedCharSequence` to display fancy ANSI text.

Here is a fictional example:

```
@Component
public class CustomPromptProvider implements PromptProvider {

    private ConnectionDetails connection;

    @Override
    public AttributedString getPrompt() {
        if (connection != null) {
            return new AttributedString(connection.getHost() + ":>",
                AttributedStyle.DEFAULT.foreground(AttributedStyle.YELLOW));
        }
        else {
            return new AttributedString("server-unknown:>",
                AttributedStyle.DEFAULT.foreground(AttributedStyle.RED));
        }
    }

    @EventListener
    public void handle(ConnectionUpdatedEvent event) {
        this.connection = event.getConnectionDetails();
    }
}
```

Customizing Command Line Options Behavior

Spring Shell comes with two default Spring Boot `ApplicationRunners`:

- `InteractiveShellApplicationRunner` bootstraps the Shell REPL. It sets up the JLine infrastructure and eventually calls `Shell.run()`
- `ScriptShellApplicationRunner` looks for program arguments that start with `@`, assumes those are local file names and tries to run commands contained in those files (with the same semantics as the script command) and then exits the process (by effectively disabling the `InteractiveShellApplicationRunner`, see below).

If this behavior does not suit you, simply provide one (or more) bean of type `ApplicationRunner` and optionally disable the standard ones. You'll want to take inspiration from the `ScriptShellApplicationRunner`:

```
@Order(InteractiveShellApplicationRunner.PRECEDENCE - 100) // Runs before
InteractiveShellApplicationRunner
public class ScriptShellApplicationRunner implements ApplicationRunner {

    @Override
    public void run(ApplicationArguments args) throws Exception {
        List<File> scriptsToRun = args.getNonOptionArgs().stream()
                .filter(s -> s.startsWith("@"))
                .map(s -> new File(s.substring(1)))
                .collect(Collectors.toList());

        boolean batchEnabled = environment.getProperty(SPRING_SHELL_SCRIPT_ENABLED,
boolean.class, true);

        if (!scriptsToRun.isEmpty() && batchEnabled) {
            InteractiveShellApplicationRunner.disable(environment);
            for (File file : scriptsToRun) {
                try (Reader reader = new FileReader(file);
                        FileInputProvider inputProvider = new FileInputProvider(
reader, parser)) {
                    shell.run(inputProvider);
                }
            }
        }
    }

...
```

Customizing Arguments Conversion

Conversion from text input to actual method arguments uses the standard Spring conversion mechanism. Spring Shell installs a new `DefaultConversionService` (with built-in converters enabled) and registers to it any bean of type `Converter<S, T>`, `GenericConverter` or `ConverterFactory<S, T>` that it finds in the application context.

This means that it's really easy to customize conversion to your custom objects of type `Foo`: just install a `Converter<String, Foo>` bean in the context.

```
@ShellComponent
class ConversionCommands {

    @ShellMethod("Shows conversion using Spring converter")
    public String conversionExample(DomainObject object) {
        return object.getClass();
    }

}

class DomainObject {
    private final String value;

    DomainObject(String value) {
        this.value = value;
    }

    public String toString() {
        return value;
    }
}

@Component
class CustomDomainConverter implements Converter<String, DomainObject> {

    @Override
    public DomainObject convert(String source) {
        return new DomainObject(source);
    }
}
```

Mind your String representation

As in the example above, it's probably a good idea if you can to have your `toString()` implementations return the converse of what was used to create the object instance. This is because when a value fails validation, Spring Shell prints

```
The following constraints were not met:
    --arg <type> : <message> (You passed '<value.toString()>')
```

See Validating Command Arguments for more information.

If you want to customize the `ConversionService` further, you can either

- Have the default one injected in your code and act upon it in some way
- Override it altogether with your own (custom converters will need to be registered by hand). The ConversionService used by Spring Shell needs to be qualified as `"spring-shell"`.

Extending Spring Shell

Support for Spring Shell 1 and JCommander

Discovering Methods that Can Act as Commands

Resolving Parameter Values

Supporting TAB Completion