Spring Shell Reference Documentation

Eric Bottard, Janne Valkealahti, Jay Bryant, Corneil du Plessis

Table of Contents

1. What is Spring Shell? . . . 2
2. Getting Started . . . 3
    2.1. Writing a Simple Boot Application . . . 3
    2.2. Adding a Dependency on Spring Shell . . . 3
    2.3. Your First Command . . . 4
    2.4. Trying the Application . . . 4
3. Basics . . . 6
4. Commands . . . 7
    4.1. Annotation Model . . . 7
    4.2. Programmatic Model . . . 8
    4.3. Organizing Commands . . . 8
    4.4. Dynamic Command Availability . . . 9
    4.5. Exit Code . . . 12
5. Options . . . 14
    5.1. Definition . . . 14
    5.2. Short Format . . . 15
    5.3. Arity . . . 16
    5.4. Positional . . . 17
    5.5. Optional Value . . . 17
    5.6. Default Value . . . 18
    5.7. Validation . . . 18
    5.8. Label . . . 19
6. Completion . . . 20
    6.1. Interactive . . . 20
    6.2. Command-Line . . . 21
7. Building . . . 22
    7.1. Native Support . . . 22
8. Components . . . 23
    8.1. Built-In Commands . . . 23
        8.1.1. Help . . . 23
        8.1.2. Clear . . . 26
        8.1.3. Exit . . . 26
        8.1.4. Stacktrace . . . 26
        8.1.5. Script . . . 26
        8.1.6. History . . . 26
        8.1.7. Completion . . . 27
        8.1.8. Version . . . 27
    8.2. Flow . . . 28

8.3. Flow Components . . . 30
8.3.1. Component Render . . . 30
8.3.2. String Input . . . 32
8.3.3. Path Input . . . 33
8.3.4. Confirmation . . . 34
8.3.5. Single Select . . . 35
8.3.6. Multi Select . . . 37
9. Customization . . . 39
9.1. Theming . . . 39
10. Execution . . . 42
10.1. Interaction Mode . . . 42
Appendix A: Appendix: Techical Introduction . . . 43
A.1. Command Registration . . . 43
A.1.1. Commands . . . 43
A.1.2. Interaction Mode . . . 43
A.1.3. Options . . . 43
A.1.4. Target . . . 44
Method . . . 44
Function . . . 45
Consumer . . . 45
A.2. Command Parser . . . 46
A.3. Command Execution . . . 46
A.4. Command Context . . . 46
A.5. Command Catalog . . . 46
A.5.1. Command Resolver . . . 47
A.5.2. Command Catalog Customizer . . . 47
A.6. Theming . . . 48

2.1.5

Chapter 1. What is Spring Shell?

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

Spring Shell lets you create such a runnable application, where the user enters textual commands that are run until the program terminates. The Spring Shell project provides the infrastructure to create such a REPL (Read, Eval, Print Loop) application, letting you concentrate on implementing commands by using the familiar Spring programming model.

Spring Shell includes advanced features (such as parsing, tab completion, colorization of output, fancy ASCII-art table display, input conversion, and validation), freeing you to focus on core command logic.

Spring Shell 2.1.x is a major rework to bring the codebase up to date with existing Spring Boot versions, adding new features and, especially, making it work with GraalVM which makes command-line applications much more relevant in a Java space. Moving to a new major version also lets us clean up the codebase and make some needed breaking changes.

Chapter 2. Getting Started

To see what Spring Shell has to offer, we can write a trivial shell application that has a simple command to add two numbers.

2.1. Writing a Simple Boot Application

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.

For the purpose of this tutorial, we create a simple Boot application by using start.spring.io. This minimal application depends only on `spring-boot-starter` and configures the `spring-boot-maven-plugin` to generate an executable über-jar:

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

2.2. Adding a Dependency on Spring Shell

The easiest way to get going with Spring Shell is to depend on the `{starter-artifactId}` artifact. This comes with everything you need 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>2.1.5</version>
</dependency>
```

Given that Spring Shell starts the REPL (Read-Eval-Print-Loop) because this dependency is present, you need to either skip tests when you build (`-DskipTests`) throughout this tutorial or remove the sample integration test that was generated by start.spring.io. If you do not remove it, the integration test creates the Spring `ApplicationContext` and, depending on your build tool, stays stuck in the eval loop or crashes with a NPE.

2.3. Your First Command

Now we can add our first command. To do so, create a new class (named whatever 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 we can create an `add` method that takes two ints (`a` and `b`) and returns their sum. We need to 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;
    }
}
```

2.4. Trying the Application

To build the application and run the generated jar, run the following command:

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

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

```
shell:>
```

A yellow `shell:>` prompt invites you to type commands. Type `add 1 2`, press `ENTER`, and admire the magic:

```
shell:>add --a 1 --b 2
3
```

You should play with the shell (hint: there is a `help` command). When you are done, type `exit` and press `ENTER`.

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

Chapter 3. Basics

This section covers the basics of Spring Shell. Before going on to define actual commands and options, we need to go through some of the fundamental concepts of Spring Shell.

Essentially, a few things needs to happen before you have a working Spring Shell application:

- Create a Spring Boot application.
- Define commands and options.
- Package the application.
- Run the application, either interactively or non-interactively.

You can get a full working Spring Shell application without defining any user-level commands as some basic built-in commands (such as `help` and `history`) are provided.

Throughout this documentation, we make references to configuring something by using annotations (mostly relates to use of `@ShellMethod` and `@ShellOption`) and to the programmatic way (which uses `CommandRegistration`).

The programmatic model is how things are actually registered, even if you use annotations. The `@ShellMethod` and `@ShellOption` annotations are a legacy feature that we do not yet want to remove. `CommandRegistration` is the new development model where new features are added. We are most likely going to replace existing annotations with something better, to support new features in a `CommandRegistration` model.

Chapter 4. Commands

In this section, we go through an actual command registration and leave command options and execution for later in a documentation. You can find more detailed info in Command Registration.

There are two different ways to define a command: through an annotation model and through a programmatic model. In the annotation model, you define your methods in a class and annotate the class and the methods with specific annotations. In the programmatic model, you use a more low level approach, defining command registrations (either as beans or by dynamically registering with a command catalog).

4.1. Annotation Model

When you use the standard API, methods on beans are 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 that is itself meta-annotated with `@Component`. As a result, you can use it in addition to the filtering mechanism to declare beans (for example, by using `@ComponentScan`).

You can customize the name of the created bean by using the `value` attribute of the annotation.

```
@ShellComponent
static class MyCommands {

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

The only required attribute of the `@ShellMethod` annotation is its `value` attribute, which should have a short, one-sentence, description of what the command does. This lets your users get consistent help about your commands without having to leave the shell (see Help).

The description of your command should be short — no more than one or two sentences. For better consistency, it should start with a capital letter and end with a period.

By default, you need not specify the key for your command (that is, the word(s) that should be used to invoke it in the shell). The name of the method is used as the command key, turning camelCase

names into dashed, gnu-style, names (for example, `sayHello()` becomes `say-hello`).

You can, however, explicitly set the command key, by using the `key` attribute of the annotation:

```
@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, the command is registered with 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. That is, you should avoid mixing dashed-names with spaced names and other inconsistencies.

4.2. Programmatic Model

In the programmatic model, `CommandRegistration` is defined as a `@Bean`, and it is automatically registered:

```
@Bean
CommandRegistration commandRegistration() {
    return CommandRegistration.builder()
        .command("mycommand")
        .build();
}
```

4.3. Organizing Commands

When your shell starts to provide a lot of functionality, you may end up with a lot of commands, which could be confusing for your users. By typing `help`, they would see a daunting list of commands, organized in alphabetical order, which may not always be the best way to show the available commands.

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

By default, commands are grouped according to the class they are implemented in, turning the camelCase class name into separate words (so `URLRelatedCommands` becomes `URL Related Commands`).

This is a sensible default, as related commands are often already in the class anyway, because 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:

1. Specify a `group()` in the `@ShellMethod` annotation.
2. Place a `@ShellCommandGroup` on the class in which the command is defined. This applies the group for all commands defined in that class (unless overridden, as explained earlier).
3. Place a `@ShellCommandGroup` on the package (through `package-info.java`) in which the command is defined. This applies to all the commands defined in the package (unless overridden at the method or class level, as explained earlier).

The following listing shows an example:

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

    @ShellMethod(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() {}
}
```

4.4. Dynamic Command Availability

Registered commands do not always make sense, due to the internal state of the application. For example, there may be 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 explain that the command exists but that it is not available at the time. Spring Shell lets you do that, even letting you provide 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 use a no-arg method that returns an instance of `Availability`. Consider the following 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");
    }
}
```

The `connect` method is used to connect to the server (details omitted), altering the state of the command through the `connected` boolean when done. The `download` command as marked as unavailable until 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. If the command is not 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 used in the integrated help. See Help.

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

You should not start the sentence with a capital or add a final period

If naming the availability method after the name of the command method does not suit you, you can provide an explicit name by using the `@ShellMethodAvailability` annotation:

```
@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

Finally, 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 as a group. Instead of having to stick the `@ShellMethodAvailability` on all command methods, Spring Shell lets you 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 `*`. This special wildcard matches all command names. This makes it easy to turn all commands of a single class on or off with a single availability method:

```
@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 many constraints on how to write commands and how to organize classes. However, it is often good practice to put related commands in the same class, and the availability indicators can benefit from that.

4.5. Exit Code

Many command line applications when applicable return an *exit code* which running environment can use to differentiate if command has been executed successfully or not. In a `spring-shell` this mostly relates when a command is run on a non-interactive mode meaning one command is always executed once with an instance of a `spring-shell`.

Default behaviour of an exit codes is as:

- Errors from a command option parsing will result code of `2`
- Any generic error will result result code of `1`
- Obviously in any other case result code is `0`

Every `CommandRegistration` can define its own mappings between *Exception* and *exit code*. Essentially we're bound to functionality in `Spring Boot` regarding *exit code* and simply integrate into that.

Assuming there is an exception show below which would be thrown from a command:

```
static class MyException extends RuntimeException {

    private final int code;

    MyException(String msg, int code) {
        super(msg);
        this.code = code;
    }

    public int getCode() {
        return code;
    }
}
```

It is possible to define a mapping function between `Throwable` and exit code. You can also just configure a *class* to *exit code* which is just a syntactic sugar within configurations.

```
CommandRegistration.builder()
    .withExitCode()
        .map(MyException.class, 3)
        .map(t -> {
            if (t instanceof MyException) {
                return ((MyException) t).getCode();
            }
            return 0;
        })
        .and()
    .build();
```

Exit codes cannot be customized with annotation based configuration

Chapter 5. Options

Command line arguments can be separated into options and positional parameters. Following sections describes features how options are defined and used.

5.1. Definition

Options can be defined within a target method as annotations in a method arguments or with programmatically with `CommandRegistration`.

Having a target method with argument is automatically registered with a matching argument name.

```
public String example(String arg1) {
    return "Hello " + arg1;
}
```

`@ShellOption` annotation can be used to define an option name if you don't want it to be same as argument name.

```
public String example(@ShellOption(value = { "--argx" }) String arg1) {
    return "Hello " + arg1;
}
```

If option name is defined without prefix, either `-` or `--`, it is discovered from *ShellMethod#prefix*.

```
public String example(@ShellOption(value = { "argx" }) String arg1) {
    return "Hello " + arg1;
}
```

Programmatic way with `CommandRegistration` is to use method adding a long name.

```
CommandRegistration.builder()
    .withOption()
        .longNames("arg1")
        .and()
    .build();
```

5.2. Short Format

Short style *POSIX* option in most is just a synonym to long format but adds additional feature to combine those options together. Having short options *a*, *b*, *c* can be used as `-abc`.

Programmatically short option is defined by using short name function.

```
CommandRegistration.builder()
    .withOption()
        .shortNames('a')
        .and()
    .withOption()
        .shortNames('b')
        .and()
    .withOption()
        .shortNames('c')
        .and()
    .build();
```

Short option with combined format is powerful if type is defined as a flag which means type is a *boolean*. That way you can define a presense of a flags as `-abc`, `-abc true` or `-abc false`.

```
CommandRegistration.builder()
    .withOption()
        .shortNames('a')
        .type(boolean.class)
        .and()
    .withOption()
        .shortNames('b')
        .type(boolean.class)
        .and()
    .withOption()
        .shortNames('c')
        .type(boolean.class)
        .and()
    .build();
```

With annotation model you can define short argument directly.

```
public String example(
    @ShellOption(value = { "-a" }) String arg1,
    @ShellOption(value = { "-b" }) String arg2,
    @ShellOption(value = { "-c" }) String arg3
) {
    return "Hello " + arg1;
}
```

5.3. Arity

Sometimes, you want to have more fine control of how many parameters with an option are processed when parsing operations happen. Arity is defined as min and max values, where min must be a positive integer and max has to be more or equal to min.

```
CommandRegistration.builder()
    .withOption()
        .longNames("arg1")
        .arity(0, 1)
        .and()
    .build();
```

Arity can also be defined as an `OptionArity` enum, which are shortcuts within the following table:

```
CommandRegistration.builder()
    .withOption()
        .longNames("arg1")
        .arity(OptionArity.EXACTLY_ONE)
        .and()
    .build();
```

Table 1. OptionArity

Value	min/max
ZERO	0 / 0
ZERO_OR_ONE	0 / 1
EXACTLY_ONE	1 / 1
ZERO_OR_MORE	0 / Integer MAX
ONE_OR_MORE	1 / Integer MAX

The annotation model supports defining only the max value of an arity.

```
public String example(@ShellOption(arity = 1) String arg1) {
    return "Hello " + arg1;
}
```

5.4. Positional

Positional information is mostly related to a command target method:

```
CommandRegistration.builder()
    .withOption()
        .longNames("arg1")
        .position(0)
        .and()
    .build();
```

5.5. Optional Value

An option is either required or not and, generally speaking, how it behaves depends on a command target:

```
CommandRegistration.builder()
    .withOption()
        .longNames("arg1")
        .required()
        .and()
    .build();
```

In the annotation model, there is no direct way to define if argument is optional. Instead, it is instructed to be `NULL`:

```
public String example(
    @ShellOption(defaultValue = ShellOption.NULL) String arg1
) {
    return "Hello " + arg1;
}
```

5.6. Default Value

Having a default value for an option is somewhat related to Optional Value, as there are cases where you may want to know if the user defined an option and change behavior based on a default value:

```
CommandRegistration.builder()
    .withOption()
        .longNames("arg1")
        .defaultValue("defaultValue")
        .and()
    .build();
```

The annotation model also supports defining default values:

```
public String example(
    @ShellOption(defaultValue = "defaultValue") String arg1
) {
    return "Hello " + arg1;
}
```

5.7. Validation

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

Annotations found on command parameters and annotations at the method level are honored and trigger validation prior to the command executing. Consider the following command:

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

From the preceding example, you get the following 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')
```

5.8. Label

Option Label has no functional behaviour within a shell itself other than what a default `help` command outputs. Within a command documentation a type of an option is documented but this is not always super useful. Thus you may want to give better descriptive word for an option.

```
CommandRegistration.builder()
    .withOption()
        .longNames("arg1")
        .and()
    .withOption()
        .longNames("arg2")
        .label("MYLABEL")
        .and()
    .build();
```

Defining label is then shown in `help`.

```
my-shell:>help mycommand
NAME
       mycommand -

SYNOPSIS
       mycommand --arg1 String --arg2 MYLABEL

OPTIONS
       --arg1 String
       [Optional]

       --arg2 MYLABEL
       [Optional]
```

Chapter 6. Completion

Spring Shell can provide completion proposals for both interactive shell and a command-line. There are differences however as when shell is in interactive mode we have an active instance of a shell meaning it's easier to provide more programmatic ways to provide completion hints. When shell is purely run as a command-line tool a completion can only be accomplished with integration into OS level shell's like *bash*.

6.1. Interactive

Hints for completions are calculated with *function* or *interface* style methods which takes `CompletionContext` and returns a list of `CompletionProposal` instances. `CompletionContext` gives you various information about a current context like command registration and option.

Generic resolvers can be registered as a beans if those are useful for all commands and scenarious. For example existing completion implementation `RegistrationOptionsCompletionResolver` handles completions for a option names.

```
static class MyValuesCompletionResolver implements CompletionResolver {

    @Override
    public List<CompletionProposal> apply(CompletionContext t) {
        return Arrays.asList("val1", "val2").stream()
            .map(CompletionProposal::new)
            .collect(Collectors.toList());
    }
}
```

Option values with builder based command registration can be defined per option.

```
void dump1() {
    CommandRegistration.builder()
        .withOption()
            .longNames("arg1")
            .completion(ctx -> {
                return Arrays.asList("val1", "val2").stream()
                    .map(CompletionProposal::new)
                    .collect(Collectors.toList());
            })
            .and()
        .build();
}
```

Option values with annotation based command registration are handled via `ValueProvider` interface which can be defined with `@ShellOption` annotation.

```
static class MyValuesProvider implements ValueProvider {

    @Override
    public List<CompletionProposal> complete(CompletionContext completionContext)
{
        return Arrays.asList("val1", "val2").stream()
            .map(CompletionProposal::new)
            .collect(Collectors.toList());
    }
}
```

Actual `ValueProvider` with annotation based command needs to be registered as a *Bean*.

```
@ShellMethod(value = "complete", key = "complete")
public String complete(
    @ShellOption(valueProvider = MyValuesProvider.class) String arg1)
{
    return "You said " + arg1;
}
```

6.2. Command-Line

Command-line completion currently only support *bash* and is documented in a built-in `completion` command Completion.

Chapter 7. Building

This section covers how to build a Spring Shell application.

7.1. Native Support

Version 2.1.x includes experimental support for compiling Spring Shell applications into native applications with GraalVM and Spring Native. Because the underlying JLine library works with GraalVM, most things should just work.

You can compile the project with a native profile to get a native application:

```
$ ./mvnw clean package -Pnative
```

You can then run the application in either interactive or non-interactive mode:

```
$ ./spring-shell-samples/target/spring-shell-samples help
AVAILABLE COMMANDS

Built-In Commands
        completion bash: Generate bash completion script
        help: Display help about available commands.
        history: Display or save the history of previously run commands
        script: Read and execute commands from a file.
...
```

Chapter 8. Components

Components are a set of features which are either build-in or something you can re-use or extend for your own needs. Components in question are either built-in *commands* or UI side components providing higher level features within commands itself.

8.1. Built-In Commands

8.1.1. Help

Running a shell application often implies that the user is in a graphically limited environment. Also, while we are nearly always connected in the era of mobile phones, 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` lists all the commands known to the shell (including unavailable commands) and a short description of what they do, similar to the following:

```
my-shell:>help
AVAILABLE COMMANDS

Built-In Commands
       exit: Exit the shell.
       help: Display help about available commands
       stacktrace: Display the full stacktrace of the last error.
       clear: Clear the shell screen.
       quit: Exit the shell.
       history: Display or save the history of previously run commands
       completion bash: Generate bash completion script
       version: Show version info
       script: Read and execute commands from a file.
```

Typing `help <command>` shows more detailed information about a command, including the available parameters, their type, whether they are mandatory or not, and other details.

The following listing shows the `help` command applied to itself:

```
my-shell:>help help
NAME
       help - Display help about available commands

SYNOPSIS
       help --command String

OPTIONS
       --command or -C String
       The command to obtain help for.
       [Optional]
```

Help is templated and can be customized if needed. Settings are under `spring.shell.command.help` where you can use `enabled` to disable command, `grouping-mode` taking `group` or `flat` if you want to hide groups by flattening a structure, `command-template` to define your template for output of a command help, `commands-template` to define output of a command list.

If `spring.shell.command.help.grouping-mode=flat` is set, then help would show:

```
my-shell:>help help
AVAILABLE COMMANDS

exit: Exit the shell.
help: Display help about available commands
stacktrace: Display the full stacktrace of the last error.
clear: Clear the shell screen.
quit: Exit the shell.
history: Display or save the history of previously run commands
completion bash: Generate bash completion script
version: Show version info
script: Read and execute commands from a file.
```

Output from `help` and `help <commmand>` are both templated with a default implementation which can be changed.

Option `spring.shell.command.help.commands-template` defaults to `classpath:template/help-commands-default.stg` and is passed `GroupsInfoModel` as a model.

Option `spring.shell.command.help.command-template` defaults to `classpath:template/help-command-default.stg` and is passed `CommandInfoModel` as a model.

Table 2. GroupsInfoModel Variables

Key	Description
`showGroups`	`true` if showing groups is enabled. Otherwise, false.
`groups`	The commands variables (see GroupCommandInfoModel Variables).
`commands`	The commands variables (see CommandInfoModel Variables).
`hasUnavailableCommands`	`true` if there is unavailable commands. Otherwise, false.

Table 3. GroupCommandInfoModel Variables

Key	Description
`group`	The name of a group, if set. Otherwise, empty.
`commands`	The commands, if set. Otherwise, empty. Type is a multi value, see CommandInfoModel Variables.

Table 4. CommandInfoModel Variables

Key	Description
`name`	The name of a command, if set. Otherwise, null. Type is string and contains full command.
`names`	The names of a command, if set. Otherwise, null. Type is multi value essentially `name` splitted.
`aliases`	The possible aliases, if set. Type is multi value with strings.
`description`	The description of a command, if set. Otherwise, null.
`parameters`	The parameters variables, if set. Otherwise empty. Type is a multi value, see CommandParameterInfoModel Variables.
`availability`	The availability variables (see CommandAvailabilityInfoModel Variables).

Table 5. CommandParameterInfoModel Variables

Key	Description
`type`	The type of a parameter if set. Otherwise, null.
`arguments`	The arguments, if set. Otherwise, null. Type is multi value with strings.
`required`	`true` if required. Otherwise, false.

Key	Description
`description`	The description of a parameter, if set. Otherwise, null.
`defaultValue`	The default value of a parameter, if set. Otherwise, null.
`hasDefaultValue`	`true` if defaultValue exists. Otherwise, false.

Table 6. CommandAvailabilityInfoModel Variables

Key	Description
`available`	`true` if available. Otherwise, false.
`reason`	The reason if not available if set. Otherwise, null.

8.1.2. Clear

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

8.1.3. Exit

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

8.1.4. Stacktrace

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

8.1.5. Script

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

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

8.1.6. History

The `history` command shows the history of commands that has been executed.

There are a few configuration options that you can use to configure behavior of a history. History is kept in a log file, which is enabled by default and can be turned off by setting `spring.shell.history.enabled`. The name of a log file is resolved from `spring.application.name` and defaults to `spring-shell.log`, which you can change by setting `spring.shell.history.name`.

By default, a log file is generated to a current working directory, which you can dictate by setting `spring.shell.config.location`. This property can contain a placeholder (`{userconfig}`), which resolves to a common shared config directory.

Run the Spring Shell application to see how the sample application works as it uses these options.

8.1.7. Completion

The `completion` command set lets you create script files that can be used with am OS shell implementations to provide completion. This is very useful when working with non-interactive mode.

Currently, the only implementation is for bash, which works with `bash` sub-command.

8.1.8. Version

The `version` command shows existing build and git info by integrating into Boot's `BuildProperties` and `GitProperties` if those exist in the shell application. By default, only version information is shown, and you can enable other information through configuration options.

The relevant settings are under `spring.shell.command.version`, where you can use `enabled` to disable a command and, optionally, define your own template with `template`. You can use the `show-build-artifact`, `show-build-group`, `show-build-name`, `show-build-time`, `show-build-version`, `show-git-branch`, `show-git-commit-id`, `show-git-short-commit-id` and `show-git-commit-time` commands to control fields in a default template.

The template defaults to `classpath:template/version-default.st`, and you can define your own, as the following example shows:

```
<buildVersion>
```

This setting would output something like the following:

```
X.X.X
```

You can add the following attributes to the default template rendering: `buildVersion`, `buildGroup`, `buildGroup`, `buildName`, `buildTime`, `gitShortCommitId`, `gitCommitId`, `gitBranch`, and `gitCommitTime`.

8.2. Flow

When you use Flow Components to build something that involves use of a multiple components, your implementation may become a bit cluttered. To ease these use cases, we added a `ComponentFlow` that can hook multiple component executions together as a “flow”.

The following listings show examples of flows and their output in a shell:

```
static class FlowSampleComplex {

    @Autowired
    private ComponentFlow.Builder componentFlowBuilder;

    public void runFlow() {
        Map<String, String> single1SelectItems = new HashMap<>();
        single1SelectItems.put("key1", "value1");
        single1SelectItems.put("key2", "value2");
        List<SelectItem> multi1SelectItems = Arrays.asList(SelectItem.of("key1",
"value1"),
                SelectItem.of("key2", "value2"), SelectItem.of("key3", "value3"));
        ComponentFlow flow = componentFlowBuilder.clone().reset()
                .withStringInput("field1")
                    .name("Field1")
                    .defaultValue("defaultField1Value")
                    .and()
                .withStringInput("field2")
                    .name("Field2")
                    .and()
                .withConfirmationInput("confirmation1")
                    .name("Confirmation1")
                    .and()
                .withPathInput("path1")
                    .name("Path1")
                    .and()
                .withSingleItemSelector("single1")
                    .name("Single1")
                    .selectItems(single1SelectItems)
                    .and()
                .withMultiItemSelector("multi1")
                    .name("Multi1")
                    .selectItems(multi1SelectItems)
                    .and()
                .build();
        flow.run();
    }

}
```

Normal execution order of a components is same as defined with a builder. It's possible to conditionally choose where to jump in a flow by using a `next` function and returning target *component id*. If this returned id is aither *null* or doesn't exist flow is essentially stopped right there.

```
static class FlowSampleConditional {

    @Autowired
    private ComponentFlow.Builder componentFlowBuilder;

    public void runFlow() {
        Map<String, String> single1SelectItems = new HashMap<>();
        single1SelectItems.put("Field1", "field1");
        single1SelectItems.put("Field2", "field2");
        ComponentFlow flow = componentFlowBuilder.clone().reset()
                .withSingleItemSelector("single1")
                    .name("Single1")
                    .selectItems(single1SelectItems)
                    .next(ctx -> ctx.getResultItem().get().getItem())
                    .and()
                .withStringInput("field1")
                    .name("Field1")
                    .defaultValue("defaultField1Value")
                    .next(ctx -> null)
                    .and()
                .withStringInput("field2")
                    .name("Field2")
                    .defaultValue("defaultField2Value")
                    .next(ctx -> null)
                    .and()
                .build();
        flow.run();
    }

}
```

The result from running a flow returns `ComponentFlowResult`, which you can use to do further actions.

8.3. Flow Components

Starting from version 2.1.x, a new component model provides an easier way to create higher-level user interaction for the usual use cases, such as asking for input in various forms. These usually are just plain text input or choosing something from a list.

Templates for built-in components are in the `org/springframework/shell/component` classpath.

Built-in components generally follow this logic:

1. Enter a run loop for user input.
2. Generate component-related context.
3. Render the runtime status of a component state.
4. Exit.
5. Render the final status of a component state.

Flow gives better interface for defining the flow of components that are better suited for defining interactive command flows.

8.3.1. Component Render

You can implement component rendering in either of two ways: fully programmatically or by using a *ANTLR Stringtemplate*. Strictly speaking, there is a simple `Function` renderer interface that takes `Context` as an input and outputs a list of `AttributedString`. This lets you choose between templating and code.

Templating is a good choice if you do not need to do anything complex or you just want to slightly modify existing component layouts. Rendering through code then gives you flexibility to do whatever you need.

The programmatic way to render is to create a `Function`:

```
class StringInputCustomRenderer implements Function<StringInputContext,
List<AttributedString>> {
    @Override
    public List<AttributedString> apply(StringInputContext context) {
        AttributedStringBuilder builder = new AttributedStringBuilder();
        builder.append(context.getName());
        builder.append(" ");
        if (context.getResultValue() != null) {
            builder.append(context.getResultValue());
        }
        else  {
            String input = context.getInput();
            if (StringUtils.hasText(input)) {
                builder.append(input);
            }
            else {
                builder.append("[Default " + context.getDefaultValue() + "]");
            }
        }
        return Arrays.asList(builder.toAttributedString());
    }
}
```

Then you can hook it to a component:

```
@ShellMethod(key = "component stringcustom", value = "String input", group =
"Components")
public String stringInputCustom(boolean mask) {
    StringInput component = new StringInput(getTerminal(), "Enter value",
"myvalue",
            new StringInputCustomRenderer());
    component.setResourceLoader(getResourceLoader());
    component.setTemplateExecutor(getTemplateExecutor());
    if (mask) {
        component.setMaskCharater('*');
    }
    StringInputContext context = component.run(StringInputContext.empty());
    return "Got value " + context.getResultValue();
}
```

Components have their own context but usually share some functionality from a parent component types. The following tables show those context variables:

Table 7. TextComponentContext Template Variables

Key	Description
`resultValue`	The value after a component renders its result.
`name`	The name of a component — that is, its title.
`message`	The possible message set for a component.
`messageLevel`	The level of a message — one of `INFO`, `WARN`, or `ERROR`.
`hasMessageLevelInfo`	Return `true` if level is `INFO`. Otherwise, false.
`hasMessageLevelWarn`	Return `true` if level is `WARN`. Otherwise, false.
`hasMessageLevelError`	Return `true` if level is `ERROR`. Otherwise, false.
`input`	The raw user input.

Table 8. SelectorComponentContext Template Variables

Key	Description
`name`	The name of a component — that is, its title.
`input`	The raw user input — mostly used for filtering.
`itemStates`	The full list of item states.
`itemStateView`	The visible list of item states.
`isResult`	Return `true` if the context is in a result mode.
`cursorRow`	The current cursor row in a selector.

8.3.2. String Input

The string input component asks a user for simple text input, optionally masking values if the content contains something sensitive. The following listing shows an example:

```
@ShellComponent
public class ComponentCommands extends AbstractShellComponent {

    @ShellMethod(key = "component string", value = "String input", group =
"Components")
    public String stringInput(boolean mask) {
        StringInput component = new StringInput(getTerminal(), "Enter value",
"myvalue");
        component.setResourceLoader(getResourceLoader());
        component.setTemplateExecutor(getTemplateExecutor());
        if (mask) {
            component.setMaskCharater('*');
        }
        StringInputContext context = component.run(StringInputContext.empty());
        return "Got value " + context.getResultValue();
    }
}
```

The following image shows typical output from a string input component:

The context object is `StringInputContext`. The following table lists its context variables:

Table 9. StringInputContext Template Variables

Key	Description
`defaultValue`	The default value, if set. Otherwise, null.
`maskedInput`	The masked input value
`maskedResultValue`	The masked result value
`maskCharacter`	The mask character, if set. Otherwise, null.
`hasMaskCharacter`	`true` if a mask character is set. Otherwise, false.
`model`	The parent context variables (see TextComponentContext Template Variables).

8.3.3. Path Input

The path input component asks a user for a `Path` and gives additional information about a path itself.

```
@ShellComponent
public class ComponentCommands extends AbstractShellComponent {

    @ShellMethod(key = "component path", value = "Path input", group =
"Components")
    public String pathInput() {
        PathInput component = new PathInput(getTerminal(), "Enter value");
        component.setResourceLoader(getResourceLoader());
        component.setTemplateExecutor(getTemplateExecutor());
        PathInputContext context = component.run(PathInputContext.empty());
        return "Got value " + context.getResultValue();
    }
}
```

The following image shows typical output from a path input component:

The context object is `PathInputContext`. The following table describes its context variables:

Table 10. PathInputContext Template Variables

Key	Description
`model`	The parent context variables (see TextComponentContext Template Variables).

8.3.4. Confirmation

The confirmation component asks a user for a simple confirmation. It is essentially a yes-or-no question.

```
@ShellComponent
public class ComponentCommands extends AbstractShellComponent {

    @ShellMethod(key = "component confirmation", value = "Confirmation input",
group = "Components")
    public String confirmationInput(boolean no) {
        ConfirmationInput component = new ConfirmationInput(getTerminal(), "Enter
value", !no);
        component.setResourceLoader(getResourceLoader());
        component.setTemplateExecutor(getTemplateExecutor());
        ConfirmationInputContext context =
component.run(ConfirmationInputContext.empty());
        return "Got value " + context.getResultValue();
    }
}
```

The following image shows the typical output from a confirmation component:

The context object is `ConfirmationInputContext`. The following table describes its context variables:

Table 11. ConfirmationInputContext Template Variables

Key	Description
`defaultValue`	The default value — either `true` or `false`.
`model`	The parent context variables (see TextComponentContext Template Variables).

8.3.5. Single Select

A single select component asks a user to choose one item from a list. It is similar to a simple dropbox implementation. The following listing shows an example:

```
@ShellComponent
public class ComponentCommands extends AbstractShellComponent {

    @ShellMethod(key = "component single", value = "Single selector", group =
"Components")
    public String singleSelector() {
        SelectorItem<String> i1 = SelectorItem.of("key1", "value1");
        SelectorItem<String> i2 = SelectorItem.of("key2", "value2");
        List<SelectorItem<String>> items = Arrays.asList(i1, i2);
        SingleItemSelector<String, SelectorItem<String>> component = new
SingleItemSelector<>(getTerminal(),
                items, "testSimple", null);
        component.setResourceLoader(getResourceLoader());
        component.setTemplateExecutor(getTemplateExecutor());
        SingleItemSelectorContext<String, SelectorItem<String>> context =
component
                .run(SingleItemSelectorContext.empty());
        String result = context.getResultItem().flatMap(si ->
Optional.ofNullable(si.getItem())).get();
        return "Got value " + result;
    }
}
```

The following image shows typical output for a single select component:

The context object is `SingleItemSelectorContext`. The following table describes its context variables:

Table 12. SingleItemSelectorContext Template Variables

Key	Description
`value`	The returned value when the component exists.
`rows`	The visible items, where rows contains maps of name and selected items.
`model`	The parent context variables (see SelectorComponentContext Template Variables).

You can pre-select an item by defining it to get exposed. This is useful if you know the default and lets the user merely press `Enter` to make a choice. The following listing sets a default:

```
SelectorItem<String> i1 = SelectorItem.of("key1", "value1");
SelectorItem<String> i2 = SelectorItem.of("key2", "value2");
List<SelectorItem<String>> items = Arrays.asList(i1, i2);
SingleItemSelector<String, SelectorItem<String>> component = new
SingleItemSelector<>(getTerminal(),
        items, "testSimple", null);
component.setDefaultExpose(i2);
```

8.3.6. Multi Select

The multi select component asks a user to select multiple items from a list. The following listing shows an example:

```
@ShellComponent
public class ComponentCommands extends AbstractShellComponent {

    @ShellMethod(key = "component multi", value = "Multi selector", group =
"Components")
    public String multiSelector() {
        List<SelectorItem<String>> items = new ArrayList<>();
        items.add(SelectorItem.of("key1", "value1"));
        items.add(SelectorItem.of("key2", "value2", false, true));
        items.add(SelectorItem.of("key3", "value3"));
        MultiItemSelector<String, SelectorItem<String>> component = new
MultiItemSelector<>(getTerminal(),
                items, "testSimple", null);
        component.setResourceLoader(getResourceLoader());
        component.setTemplateExecutor(getTemplateExecutor());
        MultiItemSelectorContext<String, SelectorItem<String>> context = component
                .run(MultiItemSelectorContext.empty());
        String result = context.getResultItems().stream()
                .map(si -> si.getItem())
                .collect(Collectors.joining(","));
        return "Got value " + result;
    }
}
```

The following image shows a typical multi-select component:

The context object is `MultiItemSelectorContext`. The following table describes its context variables:

Table 13. MultiItemSelectorContext Template Variables

Key	Description
`values`	The values returned when the component exists.
`rows`	The visible items, where rows contain maps of name, selected, on-row, and enabled items.
`model`	The parent context variables (see SelectorComponentContext Template Variables).

Chapter 9. Customization

This section describes how you can customize the shell.

9.1. Theming

Current terminal implementations are rich in features and can usually show something else that just plain text. For example a text can be styled to be *bold* or have different colors. It's also common for terminals to be able to show various characters from an unicode table like emoji's which are usually used to make shell output more pretty.

Spring Shell supports these via it's theming framework which contains two parts, firstly *styling* can be used to change text type and secondly *figures* how some characters are shown. These two are then combined together as a *theme*.

More about *theming* internals, see Theming.

Default theme is named `default` but can be change using property `spring.shell.theme.name`. Other built-in theme named `dump` uses no styling for colors and tries to not use any special figures.

Modify existing style by overriding settings.

```
static class MyStyleSettings extends StyleSettings {

    @Override
    public String highlight() {
        return super.highlight();
    }
}
```

Modify existing figures by overriding settings.

```
static class MyFigureSettings extends FigureSettings {

    @Override
    public String error() {
        return super.error();
    }
}
```

To create a new theme, create a `ThemeSettings` and provide your own *style* and *figure* implementations.

```
static class MyThemeSettings extends ThemeSettings {

    @Override
    public StyleSettings styles() {
        return new MyStyleSettings();
    }

    @Override
    public FigureSettings figures() {
        return new MyFigureSettings();
    }
}
```

Register a new bean `Theme` where you can return your custom `ThemeSettings` and a *theme* name.

```
@Configuration
static class CustomThemeConfig {

    @Bean
    Theme myTheme() {
        return new Theme() {
            @Override
            public String getName() {
                return "mytheme";
            }

            @Override
            public ThemeSettings getSettings() {
                return new MyThemeSettings();
            }
        };
    }
}
```

You can use `ThemeResolver` to resolve *styles* if you want to create JLine-styled strings programmatically and *figures* if you want to theme characters for being more pretty.

```
@Autowired
private ThemeResolver resolver;

void resolve() {
    String resolvedStyle = resolver.resolveStyleTag(StyleSettings.TAG_TITLE);
    // bold,fg:bright-white

    AttributedStyle style = resolver.resolveStyle(resolvedStyle);
    // jline attributed style from expression above

    String resolvedFigure = resolver.resolveFigureTag(FigureSettings.TAG_ERROR);
    // character i.e. U+2716 Heavy Multiplication X Emoji, cross
}
```

Chapter 10. Execution

This section describes how to set up a Spring Shell to work in interactive mode.

10.1. Interaction Mode

Version 2.1.x introduced built-in support to distinguish between interactive and non-interactive modes. This makes it easier to use the shell as a simple command-line tool without requiring customization.

Currently, interactive mode is entered if any command line options are passed when starting or running a shell from a command line. This works especially well when a shell application is compiled with Native Support.

Some commands may not have any useful meanings when they run in interactive mode or (conversely) in non-interactive mode. For example, a built-in `exit` command would have no meaning in non-interactive mode, because it is used to exit interactive mode.

The `@ShellMethod` annotation has a field called `interactionMode` that you can use to inform shell about when a particular command is available.

Appendix A: Appendix: Techical Introduction

This appendix contains information for developers and others who would like to know more about how Spring Shell works internally and what its design decisions are.

A.1. Command Registration

Defining a command registration is a first step to introducing the structure of a command and its options and parameters. This is loosely decoupled from what happens later, such as parsing command-line input and running actual target code. Essentially, it is the definition of a command API that is shown to a user.

A.1.1. Commands

A command in a `spring-shell` structure is defined as an array of commands. This yields a structure similar to the following example:

```
command1 sub1
command2 sub1 subsub1
command2 sub2 subsub1
command2 sub2 subsub2
```

We do not currently support mapping commands to an explicit parent if sub-commands are defined. For example, `command1 sub1` and `command1 sub1 subsub1` cannot both be registered.

A.1.2. Interaction Mode

Spring Shell has been designed to work on two modes: interactive (which essentially is a `REPL` where you have an active shell instance throughout a series of commands) and non-interactive (where commands are executed one by one from a command line).

Differentation between these modes is mostly around limitations about what can be done in each mode. For example, it would not be feasible to show what was a previous stacktrace of a command if the shell is no longer active. Generally, whether the shell is still active dictates the available information.

Also, being on an active `REPL` session may provide more information about what the user has been doing within an active session.

A.1.3. Options

Options can be defined as long and short, where the prefixing is `--` and `-`, respectively. The

following examples show long and short options:

```
CommandRegistration.builder()
    .withOption()
        .longNames("myopt")
        .and()
    .build();
```

```
CommandRegistration.builder()
    .withOption()
        .shortNames('s')
        .and()
    .build();
```

A.1.4. Target

The target defines the execution target of a command. It can be a method in a POJO, a `Consumer`, or a `Function`.

Method

Using a `Method` in an existing POJO is one way to define a target. Consider the following class:

```
public static class CommandPojo {

    String command(String arg) {
        return arg;
    }
}
```

Given the existing class shown in the preceding listing, you can then register its method:

```
CommandPojo pojo = new CommandPojo();
CommandRegistration.builder()
    .command("command")
    .withTarget()
        .method(pojo, "command")
        .and()
    .withOption()
        .longNames("arg")
        .and()
    .build();
```

Function

Using a `Function` as a target gives a lot of flexibility to handle what happens in a command execution, because you can handle many things manually by using a `CommandContext` given to a `Function`. The return type from a `Function` is then what gets printed into the shell as a result. Consider the following example:

```
CommandRegistration.builder()
    .command("command")
    .withTarget()
        .function(ctx -> {
            String arg = ctx.getOptionValue("arg");
            return String.format("hi, arg value is '%s'", arg);
        })
        .and()
    .withOption()
        .longNames("arg")
        .and()
    .build();
```

Consumer

Using a `Consumer` is basically the same as using a `Function`, with the difference being that there is no return type. If you need to print something into a shell, you can get a reference to a `Terminal` from a context and print something through it. Consider the following example:

```
CommandRegistration.builder()
    .command("command")
    .withTarget()
        .consumer(ctx -> {
            String arg = ctx.getOptionValue("arg");
            ctx.getTerminal().writer()
                .println(String.format("hi, arg value is '%s'", arg));
        })
    .and()
    .withOption()
        .longNames("arg")
        .and()
    .build();
```

A.2. Command Parser

Before a command can be executed, we need to parse the command and whatever options the user may have provided. Parsing comes between command registration and command execution.

A.3. Command Execution

When command parsing has done its job and command registration has been resolved, command execution does the hard work of running the code.

A.4. Command Context

The `CommandContext` interface gives access to a currently running context. You can use it to get access to options:

```
String arg = ctx.getOptionValue("arg");
```

If you need to print something into a shell, you can get a `Terminal` and use its writer to print something:

```
ctx.getTerminal().writer().println("hi");
```

A.5. Command Catalog

The `CommandCatalog` interface defines how command registrations exist in a shell application. It is

possible to dynamically register and de-register commands, which gives flexibility for use cases where possible commands come and go, depending on a shell's state. Consider the following example:

```
CommandRegistration registration = CommandRegistration.builder().build();
catalog.register(registration);
```

A.5.1. Command Resolver

You can implement the `CommandResolver` interface and define a bean to dynamically resolve mappings from a command's name to its `CommandRegistration` instances. Consider the following example:

```
static class CustomCommandResolver implements CommandResolver {
    List<CommandRegistration> registrations = new ArrayList<>();

    CustomCommandResolver() {
        CommandRegistration resolved = CommandRegistration.builder()
            .command("resolve command")
            .build();
        registrations.add(resolved);
    }

    @Override
    public List<CommandRegistration> resolve() {
        return registrations;
    }
}
```

A current limitation of a `CommandResolver` is that it is used every time commands are resolved. Thus, we advise not using it if a command resolution call takes a long time, as it would make the shell feel sluggish.

A.5.2. Command Catalog Customizer

You can use the `CommandCatalogCustomizer` interface to customize a `CommandCatalog`. Its main use is to modify a catalog. Also, within `spring-shell` auto-configuration, this interface is used to register existing `CommandRegistration` beans into a catalog. Consider the following example:

```
static class CustomCommandCatalogCustomizer implements CommandCatalogCustomizer {

    @Override
    public void customize(CommandCatalog commandCatalog) {
        CommandRegistration registration = CommandRegistration.builder()
            .command("resolve command")
            .build();
        commandCatalog.register(registration);
    }
}
```

You can create a `CommandCatalogCustomizer` as a bean, and Spring Shell handles the rest.

A.6. Theming

Styling in a theming is provided by a use of a *AttributedString* from `JLine`. Unfortunately styling in `JLine` is mostly undocumented but we try to go through some of its features here.

In `JLine` a style spec is a string having a special format. Spec can be given multiple times if separated by a comma. A spec will either define a color for foreground, background or its mode. Special format `<spec>:=<spec>` allows to define a default within latter spec if former for some reason is invalid.

If spec contains a colon its former part indicates either foreground or background and possible values are `foreground`, `fg`, `f`, `background`, `bg`, `b`, `foreground-rgb`, `fg-rgb`, `f-rgb`, `background-rgb`, `bg-rgb` or `b-rgb`. Without rbg a color value is name from an allowable colors `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` or `white`. Colors have their short formats `k`, `r`, `g`, `y`, `b`, `m`, `c` and `w` respectively. If color is prefixed with either `!` or `bright-`, bright mode is automatically applied. Prefixing with `~` will resolve from JLine internal bsd color table.

If rgb format is expected and prefixed with either `x` or `#` a normal hex format is used.

```
fg-red
fg-r
fg-rgb:red
fg-rgb:xff3333
fg-rgb:#ff3333
```

If spec contains special names `default`, `bold`, `faint`, `italic`, `underline`, `blink`, `inverse`, `inverse-neg`, `inverseneg`, `conceal`, `crossed-out`, `crossedout` or `hidden` a style is changed accordingly with an existing color.

```
bold
bold,fg:red
```

If spec is a number or numbers separated with semicolon, format is a plain part of an ansi ascii codes.

```
31
31;1
```

JLine special mapping format which would resolve spec starting with dot can't be used as we don't yet map those into Spring Shell styling names.