2

Defining Beans and Using Dependency Injection

In this chapter, we will cover the following recipes:

- Defining a bean explicitly with @Bean
- Defining a bean implicitly with @Component
- Using a bean via dependency injection with @Autowired
- Using a bean directly
- Listing all beans
- Using multiple configuration classes

Introduction

Beans are at the core of Spring. They are standard Java objects instantiated and managed by Spring.

Beans are mostly used to:

- Configure Spring in some way (database connection parameters, security, and so on)
- Avoid hardcoding dependencies using **dependency injection**, so that our classes remain self-contained and unit testable

In this chapter, you'll learn how to define beans and use them.

Defining a bean explicitly with @Bean

The simplest way to define a bean is to create, in a Spring configuration class, a method annotated with `@Bean` returning an object (the actual bean). Such beans are usually used to configure Spring in some way (database, security, view resolver, and so on). In this recipe, we'll define a bean that contains the connection details of a database.

How to do it...

In a Spring configuration class, add a `dataSource()` method annotated with `@Bean` and return a `Datasource` object. In this method, create a `DriverManagerDataSource` object initialized with the connection details of a database:

```
@Bean
public DataSource dataSource() {
        DriverManagerDataSource dataSource = new
DriverManagerDataSource();

        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/db1");
        dataSource.setUsername("root");
        dataSource.setPassword("123");

        return dataSource;
}
```

How it works...

At startup, because of `@Bean`, the `dataSource()` method is automatically executed and returns a `Datasource` object which is stored by Spring (in a Spring object called `ApplicationContext`). The bean name is `dataSource`, which is the same as its methods name. From this point, any call to `dataSource()` will return the same cached `DataSource` object; `dataSource()` won't actually be executed again. This is done using aspect-oriented programming; any call to `dataSource()` is intercepted by Spring, which directly returns the object instead of executing the method.

There's more...

To customize the bean name, use the name parameter:

```
@Bean(name="theSource")
public DataSource dataSource() {
...
```

To force `dataSource()` to be executed each time it's called (and return a different object each time), use the `@Scope` annotation with a `prototype` scope:

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public DataSource dataSource() {
...
```

It's possible to define beans using our own classes. For example, if we have a `UserService` class, we can define a `UserService` bean in a Spring configuration class:

```
@Bean
public UserService userService() {
        return new UserService();
}
```

This class should be a configuration class

However, it's usually simpler to let Spring generate this kind of beans automatically using a `@Component` annotation on the `UserService` class, as explained in the *Defining a bean implicitly with @Component* recipe.

Defining a bean implicitly with @Component

Beans don't have to be defined in a Spring configuration class. Spring will automatically generate a bean from any class annotated with `@Component`.

Getting ready

We will use the basic web application created in the *Creating a Spring web application* recipe in *Chapter 1*, *Creating a Spring Application*.

Create the `com.springcookbook.service` package and the following service class in it:

```
public class UserService {
  public int findNumberOfUsers() {
    return 10;
  }
}
```

How to do it...

Here are the steps to define a bean by adding `@Component` to an existing class:

1. In the Spring configuration file, in the `@ComponentScan` class annotation, add the `com.springcookbook.service` base package:

```
@Configuration
@EnableWebMvc
```

```
@ComponentScan(basePackages =
{"com.springcookbook.controller",
"com.springcookbook.service"})
public class AppConfig {
}
```

2. In the `UserService` class, add `@Component`:

```
@Component
public class UserService {
  public int findNumberOfUsers() {
    return 10;
  }
}
```

How it works...

At startup, the `com.springcookbook.service` package will be scanned by Spring. The `UserService` class is annotated with `@Component`, so a bean is automatically instantiated from it. The bean's name will be `userService` by default, based on the class name.

To specify a custom name, use the following code:

```
@Component('anAmazingUserService')
public class UserService {
```

There's more...

If the `UserService` bean requires some custom initialization, for example, based on the current environment, it's possible to define and initialize the bean explicitly as explained in the previous recipe, *Defining a bean explicitly with @Bean*.

`@Controller`, `@Service`, and `@Repository` are also component annotations; Spring will automatically instantiate a bean at startup from the classes annotated with them. It's not strictly necessary to use these component annotations, but they make the role of the component class clearer; `@Controller` is used for controller classes, `@Service` is used for service classes (so that's the one we would actually use for our `UserService` class), and `@Repository` is used for persistence classes. They also add minor extra functionality to the component classes. Refer to `http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/beans.html#beans-stereotype-annotations`.

Using a bean via dependency injection with @Autowired

Spring configuration beans, such as the one in the *Defining a bean explicitly with @Bean* recipe are automatically discovered and used by Spring. To use a bean (any kind of bean) in one of your classes, add the bean as a field and annotate it with `@Autowired`. Spring will automatically initialize the field with the bean. In this recipe, we'll use an existing bean in a controller class.

Getting ready

We will use the code from the *Defining a bean implicitly with @Component* recipe, where we defined a `UserService` bean.

How to do it...

Here are the steps to use an existing bean in one of your classes:

1. In the controller class, add a `UserService` field annotated with `@Autowired`:

```
@Autowired
UserService userService;
```

2. In a controller method, use the `UserService` field:

```
@RequestMapping("hi")
@ResponseBody
public String hi() {
  return "nb of users: " + userService.findNumberOfUsers();
}
```

3. In a web browser, go to `http://localhost:8080/hi` to check whether it's working.

How it works...

When the controller class is instantiated, Spring automatically initializes the `@Autowired` field with the existing `UserService` bean. This is called dependency injection; the controller class simply declares its dependency, a `UserService` field. It's Spring that initializes the field by injecting a `UserService` object into it.

If Spring is not able to find an existing bean for that dependency, an exception is thrown.

There's more...

It's possible to set the name of the bean to be used:

```
@Autowired("myUserService")
UserService userService;
```

Dependency injection is useful when interfaces are used. For example, we could replace our `UserService` class by a `UserService` interface and its implementation `UserServiceImpl`. Everything would work the same, except that it's now simple to swap `UserServiceImpl` for another class, for example, for unit testing purposes.

Using a bean directly

It's possible to get a bean directly from Spring instead of using dependency injection by making Spring's `ApplicationContext`, which contains all the beans, a dependency of your class. In this recipe, we'll inject an existing bean into a controller class.

Getting ready

We will use the code from the *Defining a bean implicitly with @Component* recipe, where we defined a `UserService` bean.

How to do it...

Here are the steps to get and use a bean directly:

1. In the controller class, add an `ApplicationContext` field annotated with `@Autowired`:

   ```
   @Autowired
   private ApplicationContext applicationContext;
   ```

2. In a controller method, use the `ApplicationContext` object and its `getBean()` method to retrieve the `UserService` bean:

   ```
   UserService userService =
   (UserService)applicationContext.getBean("userService");
   ```

How it works...

When the controller class is instantiated, Spring automatically initializes the `@Autowired` field with its `ApplicationContext` object. The `ApplicationContext` object references all Spring beans, so we can get a bean directly using its name.

There's more...

It's possible to get a bean by its class, without knowing its name.

```
applicationContext.getBean(UserService.class);
```

Listing all beans

It can be useful, especially for debugging purposes, to list all the beans at a given moment.

Getting ready

We will use the code from the *Defining a bean implicitly with @Component* recipe, where we defined a `UserService` bean.

How to do it...

Here are the steps to retrieve the names of the beans currently in Spring's `ApplicationContext` object:

1. In your class, add an `ApplicationContext` field annotated with `@Autowired`:

```
@Autowired
private ApplicationContext applicationContext;
```

2. In a method of that class, use `ApplicationContext` and its `getBeanDefinitionNames()` method to get the list of bean names:

```
String[] beans =
applicationContext.getBeanDefinitionNames();
for (String bean : beans) {
   System.out.println(bean);
}
```

How it works...

When the controller class is instantiated, Spring automatically initializes the `@Autowired` field with its `ApplicationContext` object. The `ApplicationContext` object references all Spring beans, so we can get a list of all the beans that are using it.

There's more...

To retrieve the bean itself from its name, use the `getBean()` method:

```
applicationContext.getBean("aBeanName");
```

Using multiple configuration classes

A Spring configuration class can get quite long with many bean definitions. At this point, it can be convenient to break it into multiple classes.

Getting ready

We will use the code from the *Defining a bean explicitly with @Bean* recipe.

How to do it...

Here's how to add a second configuration class:

1. Create a new configuration class, for example, `DatabaseConfig` in the `com.springcookbook.config` package:

```
@Configuration
public class DatabaseConfig {
…
```

2. In the `ServletInitializer` class, add the `DatabaseConfig` class in the `getServletConfigClasses()` method:

```
@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[]{AppConfig.class,
DatabaseConfig.class};
}
```

3. Move the `Datasource` bean from the `AppConfig` class to `DatabaseConfig`.

There's more...

If you are using a Spring application without a `ServletInitializer` class, you can include other configuration classes from your primary configuration class:

```
@Configuration
@Import({ DatabaseConfig.class, SecurityConfig.class })
public class AppConfig {
…
}
```

12
Using Aspect-oriented Programming

In this chapter, we will cover the following recipes:

- Creating a Spring AOP aspect class
- Measuring the execution time of methods using an around advice
- Logging method arguments using a before advice
- Logging methods' return values using an after-returning advice
- Logging exceptions using an after-throwing advice
- Using an after advice to clean up resources
- Making a class implement an interface at runtime using an introduction
- Setting the execution order of the aspects

Introduction

Aspect-oriented programming (**AOP**) is about inserting and executing, at runtime, extra pieces of code at various points of the normal execution flow of a program. In AOP terminology, these pieces of code are methods that are called **advices** and the classes containing them are called **aspects**. AOP is complementary to object-oriented programming.

This chapter is about the Spring AOP framework, which enables us to execute advices before and after methods of Spring beans (controller methods, service methods, and so on). For more extensive AOP functionality, **AspectJ** is the reference Java AOP framework and gets integrated seamlessly with Spring. However, it's more complex to use and requires a customized compilation process.

In the first recipe, we will create an aspect class and configure Spring to use it. We will use this aspect class in the following recipes, where we will go through the different types of advice offered by Spring AOP, using practical use cases.

Creating a Spring AOP aspect class

In this recipe, we will create an aspect class and configure Spring to use it. We will use this aspect class and its configuration code in the following recipes.

How to do it...

Here are the steps for creating an aspect class:

1. Add the the AspectJ Weaver Maven dependency in `pom.xml`:

```
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.5</version>
</dependency>
```

2. Create a Java package for the aspects of your application. For example, `com.springcookbook.aspect`.
3. In your aspects package, create a class annotated with `@Component` and `@Aspect`:

```
@Component
@Aspect
public class Aspect1 {

}
```

4. In the Spring configuration, add `@EnableAspectJAutoProxy` and your aspects package to `@ComponentScan`:

```
@Configuration
@EnableAspectJAutoProxy
@ComponentScan(basePackages =
{"com.spring_cookbook.controllers",
"com.spring_cookbook.aspect"})
public class AppConfig {
...
}
```

How it works...

The AspectJ Weaver Maven dependency provides aspect annotations, so we can use regular Java classes to define aspects.

In the aspect class, `@Aspect` declares the class as an aspect. `@Component` allows it to be detected by Spring and instantiated as a bean.

In the Spring configuration, we included our aspect package in `@ComponentScan`, so the `@Component` classes in that package will be detected and instantiated as beans by Spring. `@EnableAspectJAutoProxy` in the Spring configuration will make Spring actually use the aspects and execute their advices.

Measuring the execution time of methods using an around advice

An **around advice** is the most powerful type of advice; it can completely replace the target method by some different code. In this recipe, we will use it only to execute some extra code before and after the target method. With the before code, we will get the current time. With the after code, we will get the current time again, and will compare it to the previous time to calculate the total time the target method took to execute. Our target methods will be the controller methods of the controller classes in the controller package.

Getting ready

We will use the aspect class defined in the previous recipe, *Creating a Spring AOP aspect class*.

How to do it...

Here are the steps for measuring the execution time of controller methods:

1. In the aspect class, create an advice method annotated with `@Around` and take `ProceedingJoinPoint` as an argument:

```
@Around("execution(*
com.spring_cookbook.controllers.*.*(..))")
public Object doBasicProfiling(ProceedingJoinPoint
joinPoint) throws Throwable {
...
}
```

2. In that advice method, measure the execution time of the target method:

```
Long t1 = System.currentTimeMillis();
Object returnValue = joinPoint.proceed();
Long t2 = System.currentTimeMillis();
Long executionTime = t2 - t1;
```

3. Log that execution time preceded by the target method name:

```
String className =
joinPoint.getSignature().getDeclaringTypeName();
String methodName = joinPoint.getSignature().getName();
System.out.println(className + "." + methodName + "() took
" + executionTime + " ms");
```

4. Return the return value of the target method:

```
return returnValue;
```

5. To test the advice, you can use a controller method that takes a long time on purpose:

```
@RequestMapping("user_list")
@ResponseBody
public void userList() throws Exception {
  try {
      Thread.sleep(2500);  // wait 2.5 seconds
  } catch(InterruptedException ex) {
      Thread.currentThread().interrupt();
  }
}
```

6. Test whether it's working. When going to `/user_list` in your browser, you should see this in your server log:

```
com.spring_cookbook.controllers.UserController.userList()
took 2563 ms
```

How it works...

The `@Around` annotation preceding the advice method is a pointcut expression:

```
@Around("execution(* com.spring_cookbook.controllers.*.*(..))")
```

A pointcut expression determines the target methods (the methods to which the advice will be applied). It works like a regular expression. Here, it matches all controller methods. In detail:

- `execution()` means we are targeting a method execution
- The first asterisk means *any return type*
- The second asterisk means *any class* (from the `com.spring_cookbook.controllers` package)
- The third asterisk means *any method*
- `(..)` means *any number of method arguments of any type*

The `joinPoint.proceed()` instruction executes the target method. Skipping this will skip the execution of the target method. A **join point** is another AOP term. It's a moment in the execution flow of the program where an advice can be executed. With Spring AOP, a join point always designates a target method. To summarize, an advice method is applied at different join points, which are identified by a pointcut expression.

We also use the `joinPoint` object to get the name of the current target method:

```
String className =
joinPoint.getSignature().getDeclaringTypeName();
String methodName = joinPoint.getSignature().getName();
```

Logging method arguments using a before advice

A **before advice** executes some extra code before the execution of the target method. In this recipe, we will log the arguments of the target method.

Getting ready

We will use the aspect class defined in the *Creating a Spring AOP aspect class* recipe.

How to do it...

Here are the steps for logging the methods' arguments using a before advice:

1. In your aspect class, create an advice method annotated with `@Before` and take `JoinPoint` as an argument:

```
@Before("execution(*
com.spring_cookbook.controllers.*.*(..))")
public void logArguments(JoinPoint joinPoint) {
...
}
```

2. In that method, get the list of arguments of the target method:

```
Object[] arguments = joinPoint.getArgs();
```

3. Log the list of arguments preceded by the target method name:

```
String className =
joinPoint.getSignature().getDeclaringTypeName();
String methodName = joinPoint.getSignature().getName();
System.out.println("-----" + className + "." + methodName +
"() -----");

for (int i = 0; i < arguments.length; i++) {
  System.out.println(arguments[i]);
}
```

4. Test the advice using a controller method with arguments:

```
@RequestMapping("user_list")
@ResponseBody
public String userList(Locale locale, WebRequest request) {
...
}
```

5. Check whether it's working. When going to `/user_list` in your browser, you should see this in your server log:

```
-----
com.spring_cookbook.controllers.UserController.userList()
-----
en_US
ServletWebRequest:
uri=/spring_webapp/user_list;client=10.0.2.2
```

How it works...

The `@Before` annotation preceding the advice method is a pointcut expression:

```
@Before("execution(* com.spring_cookbook.controllers.*.*(..))")
```

Refer to the *Measuring the execution time of methods using an around advice* recipe for more details.

The `joinPoint.getArgs()` instruction retrieves the argument's values of the target method.

Logging methods' return values using an after-returning advice

An **after-returning advice** executes some extra code after the successful execution of the target method. In this recipe, we will log the return value of the target method.

Getting ready

We will use the aspect class defined in the *Creating a Spring AOP aspect class* recipe.

How to do it...

Here are the steps for logging the return value of methods using an after-returning advice:

1. In your aspect class, create an advice method annotated with `@AfterReturning`. Make it take a `JoinPoint` object and the return value of the target method as arguments:

```
@AfterReturning(pointcut="execution(*
com.spring_cookbook.controllers.*.*(..))",
returning="returnValue")
public void logReturnValue(JoinPoint joinPoint, Object
returnValue) {
...
}
```

2. In that advice method, log the return value preceded by the target method name:

```
String className =
joinPoint.getSignature().getDeclaringTypeName();
String methodName = joinPoint.getSignature().getName();
System.out.println("-----" + className + "." + methodName +
"() -----");
System.out.println("returnValue=" + returnValue);
```

3. Test the advice using a controller method that returns a value:

```
@RequestMapping("user_list")
@ResponseBody
public String userList() {
  return "just a test";
}
```

4. Check whether it's working. When going to `/user_list` in your browser, you should see the following in your server log:

```
-----
com.spring_cookbook.controllers.UserController.userList()
-----
returnValue=just a test
```

How it works...

The `@AfterReturning` annotation preceding the advice method is a pointcut expression:

```
@AfterReturning(pointcut="execution(*
com.spring_cookbook.controllers.*.*(..))",
returning="returnValue")
```

Refer to the *Measuring the execution time of methods using an around advice* recipe for more details. The `returning` attribute is the name of the argument of the advice method to be used for the return value.

Note that if an exception is thrown during the execution of the target method, the after-returning advice won't be executed.

Logging exceptions using an after-throwing advice

An **after-throwing advice** executes some extra code when an exception is thrown during the execution of the target method. In this recipe, we will just log the exception.

Getting ready

We will use the aspect class defined in the *Creating a Spring AOP aspect class* recipe.

How to do it...

Here are the steps for logging an exception using an after-throwing advice:

1. In your aspect class, create an advice method annotated with `@AfterThrowing`. Make it take a `JoinPoint` object and an `Exception` object as arguments:

```
@AfterThrowing(pointcut="execution(*
com.spring_cookbook.controllers.*.*(..))",
throwing="exception")
```

```
public void logException(JoinPoint joinPoint, Exception
exception) {
...
}
```

2. In that advice method, log the exception preceded by the target method name:

```
String className =
joinPoint.getSignature().getDeclaringTypeName();
String methodName = joinPoint.getSignature().getName();
System.out.println("-----" + className + "." + methodName +
"() -----");
System.out.println("exception message:" +
exception.getMessage());
```

3. Test the advice using a controller method throwing an exception:

```
@RequestMapping("user_list")
@ResponseBody
public String userList() throws Exception  {
  throw new Exception("a bad exception");
}
```

4. Check whether it's working. When going to `/user_list` in your browser, you should see the following in your server log:

```
-----
com.spring_cookbook.controllers.UserController.userList()
-----
exception message:a bad exception
```

How it works...

The `@AfterThrowing` annotation preceding the advice method is a pointcut expression:

```
@AfterThrowing(pointcut="execution(*
com.spring_cookbook.controllers.*.*(..))", throwing="exception")
```

Refer to the *Measuring the execution time of methods using an around advice* recipe for more details. The `throwing` attribute is the name of the argument of the advice method to be used for the exception object thrown by the target method.

Note that if no exception is thrown during the execution of the target method, the after-throwing advice won't be executed.

Using an after advice to clean up resources

An **after advice** executes some extra code after the execution of the target method, even if an exception is thrown during its execution. Use this advice to clean up resources by removing a temporary file or closing a database connection. In this recipe, we will just log the target method name.

Getting ready

We will use the aspect class defined in the *Creating a Spring AOP aspect class* recipe.

How to do it...

Here are the steps for using an after advice:

1. In your aspect class, create an advice method annotated with `@After`. Make it take `JoinPoint` as an argument:

```
@After("execution(*
com.spring_cookbook.controllers.*.*(..))")
public void cleanUp(JoinPoint joinPoint) {
...
}
```

2. In that advice method, log the target method name:

```
String className =
joinPoint.getSignature().getDeclaringTypeName();
String methodName = joinPoint.getSignature().getName();
System.out.println("-----" + className + "." + methodName +
"() -----");
```

3. Test the advice using two controller methods: one executes normally and the other one throws an exception:

```
@RequestMapping("user_list")
@ResponseBody
public String userList() {
  return "method returning normally";
}

@RequestMapping("user_list2")
@ResponseBody
public String userList2() throws Exception  {
  throw new Exception("just a test");
}
```

4. Check whether it's working. When going to `/user_list` or `/user_list2` in your browser, you should see this in your server log:

```
-----
com.spring_cookbook.controllers.UserController.userList()
-----
```

How it works...

The `@After` annotation preceding the advice method is a pointcut expression:

```
@After("execution(* com.spring_cookbook.controllers.*.*(..))")
```

Refer to the *Measuring the execution time of methods using an around advice* recipe for more details.

Making a class implement an interface at runtime using an introduction

An **introduction** allows us to make a Java class (we will refer to it as the *target class*) implement an interface at runtime. With Spring AOP, introductions can be applied only to Spring beans (controllers, services, and so on). In this recipe, we will create an interface, its implementation, and make a Spring controller implement that interface at runtime using that implementation. To check whether it's working, we will also add a before advice to the controller method to execute a method from the interface implementation.

Getting ready

We will use the aspect class defined in the *Creating a Spring AOP aspect class* recipe.

How to do it...

Here are the steps for using an introduction:

1. Create the `Logging` interface:

```
public interface Logging {
  public void log(String str);
}
```

2. Create an implementation class for it:

```
public class LoggingConsole implements Logging {

  public void log(String str) {
```

```
        System.out.println(str);
      }
    }
```

3. In your aspect class, add a `Logging` attribute annotated with `@DeclareParents`. Add the implementation class to `@DeclareParents`:

```
@DeclareParents(value =
"com.spring_cookbook.controllers.*+", defaultImpl =
LoggingConsole.class)
public static Logging mixin;
```

4. Add an advice method annotated with `@Before`. Make it take a `Logging` object as an argument:

```
@Before("execution(*
com.spring_cookbook.controllers.*.*(..)) && this(logging)")
public void logControllerMethod(Logging logging) {
...
}
```

5. In the advice method, use the `Logging` object:

```
logging.log("this is displayed just before a controller
method is executed.");
```

6. Test whether it's working with a standard controller method:

```
@RequestMapping("user_list")
@ResponseBody
public String userList() {
  return "method returning normally";
}
```

7. Check whether it's working. When going to `/user_list` in your browser, you should see the following in your server log:

```
this is displayed just before a controller method is
executed.
```

How it works...

In the aspect class, the `@DeclareParents` annotation preceding the Logging attribute is a pointcut expression:

```
@DeclareParents(value = "com.spring_cookbook.controllers.*+",
defaultImpl = LoggingConsole.class)
```

This pointcut expression and the `Logging` attribute define that:

- The introduction will be applied to all controller classes: `com.spring_cookbook.controllers.*+`
- The introduction will make these controller classes implement the Logging interface: `public static Logging mixin;`
- The introduction will make these controller classes use `LoggingConsole` as implementation of the Logging interface: `defaultImpl = LoggingConsole.class`

The before advice works the same way as in the *Measuring the execution time of methods using an around advice* recipe. It only takes one extra condition:

```
this(logging)
```

This means that the advice will be applied only to objects that implement the Logging interface.

Setting the execution order of the aspects

When using several aspect classes, it can be necessary to set the order in which the aspects are executed. In this recipe, we will use two aspect classes with before advices targeting controller methods.

Getting ready

We will use the configuration from the *Creating a Spring AOP aspect class* recipe.

We will use these two aspect classes containing an advice, which logs some text when it's executed:

```
@Component
@Aspect
public class Aspect1 {

  @Before("execution(* com.spring_cookbook.controllers.*.*(..))")
  public void advice1() {
    System.out.println("advice1");
  }

}

@Component
@Aspect
```

```
public class Aspect2 {

  @Before("execution(* com.spring_cookbook.controllers.*.*(..))")
  public void advice2() {
    System.out.println("advice2");
  }

}
```

How to do it...

Here are the steps to set the execution order of the two aspect classes:

1. Add `@Order` with a number as parameter to the first aspect:

```
@Component
@Aspect
@Order(1)
public class Aspect1 {
```

2. Add `@Order` with another number as parameter to the second aspect:

```
@Component
@Aspect
@Order(2)
public class Aspect2 {
```

3. Test whether it's working. When going to `/user_list` in your browser, you should see this in your server log:

```
advice1
advice2
```

4. Switch the `@Order` numbers and check whether the execution order is changed:

```
advice2
advice1
```

How it works...

The aspects are executed in the ascending order set by `@Order`.

There's more...

It's not possible to set an order between advice methods of the same aspect class. If it becomes necessary, create new aspect classes for those advices.

8
Running Batch Jobs

In this chapter, we will cover the following recipes:

- Installing and configuring Spring Batch
- Creating a job
- Executing a job from the command line
- Executing a job from a controller method
- Using job parameters
- Executing a system command
- Scheduling a job
- Creating a read/process/write step
- Reading an XML file
- Generating a CSV file
- Reading from a database
- Unit testing batch jobs

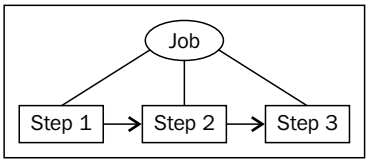
Introduction

A **batch job** is a task executed outside the normal web application workflow (receiving an HTTP request and sending back an HTTP response). It can be executed by the web server as a separate process. It can also be launched directly from the command line.

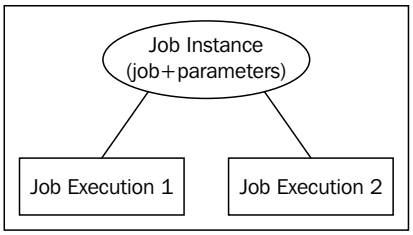
Typically, a batch job either:

- Imports or exports data at a scheduled time. For example, importing a CSV file in the database every night.
- Executes some code asynchronously to avoid long page loads. For example, processing a video uploaded by the user or generating a big file that will be downloaded by the user.

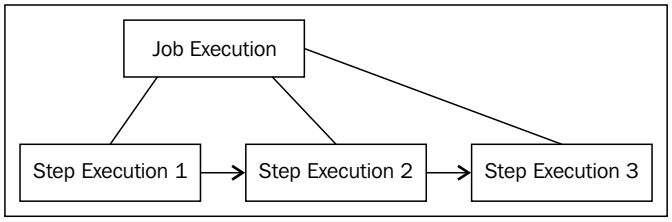
Spring Batch provides a structure to define, run, and monitor batch jobs. A **Job** is defined as a sequence of steps:



A **Job Instance** is the combination of a **job** and some **parameters**. For example, the day's date and the name of the file to process. A **Job Execution** is created for a job instance. If the job execution fails, another job execution can be created for the same job instance.



A **Job Execution** generates a **Step Execution** for each step of the job. If a step execution fails, another step execution can be created for that same step:



Installing and configuring Spring Batch

Spring automatically saves some metadata (start time, end time, and status) about jobs and their steps in a job repository, which consists of several database tables. In this recipe, we'll create these tables. We will also create a Spring configuration class dedicated to batch jobs.

How to do it...

Here are the steps to install and configure Spring Batch:

1. Add the Maven dependencies for Spring Batch in `pom.xml`:

```
<dependency>
    <groupId>org.springframework.batch</groupId>
    <artifactId>spring-batch-core</artifactId>
    <version>3.0.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework.batch</groupId>
    <artifactId>spring-batch-infrastructure</artifactId>
    <version>3.0.2.RELEASE</version>
</dependency>
```

2. Add the Maven dependencies for Spring JDBC and Spring Transaction in `pom.xml`:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>4.1.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>4.1.2.RELEASE</version>
</dependency>
```

3. Add the Maven dependency for your database in `pom.xml`:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.34</version>
</dependency>
```

4. In the database, create the tables for Spring Batch's job repository. The SQL code can be found inside the spring-batch-core dependency in the `org.springframework.batch.core` package. It's also available online at `https://github.com/spring-projects/spring-batch/tree/master/spring-batch-core/src/main/resources/org/springframework/batch/core`.
5. Create a Java package for your Spring Batch classes. For example, `com.spring_cookbook.batch`.
6. Create a Spring configuration class for Spring Batch with the `@EnableBatchProcessing` annotation:

```
@Configuration
@EnableBatchProcessing
public class BatchConfig {
...
}
```

7. Add a `DataSource` bean with the database connection details to the configuration class:

```
@Bean
public DataSource dataSource() {
        DriverManagerDataSource dataSource = new
DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://localhost:3306/db1");
        dataSource.setUsername("root");
        dataSource.setPassword("123");

        return dataSource;
}
```

How it works...

In the configuration class, the `@EnableBatchProcessing` annotation enables Spring Batch and provides reasonable defaults for batch jobs, which can be overridden if necessary (the default `JobLauncher` object, the default `TransactionManager` object, and so on).

Creating a job

We'll create a job that will simply execute some Java code. It will be a job with only one step. The step will be a `Tasklet` object (a single task, as opposed to a read-process-write step, which we'll cover later). We will execute this job in two different ways in the next two recipes.

How to do it...

Create a `Tasklet` class, which you will use to define a step and the job:

1. Create the `Task1` class implementing `Tasklet`:

```
public class Task1 implements Tasklet {

}
```

2. In the `Task1` class, add an `execute()` method with the code to be executed for the job:

```
public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext)
        throws Exception {
    System.out.println("Starting job..");

    // ... your code

    System.out.println("Job done..");
    return RepeatStatus.FINISHED;
}
```

3. In the configuration class, add an autowired `JobBuilderFactory` attribute and an autowired `StepBuilderFactory` attribute:

```
@Autowired
private JobBuilderFactory jobs;

@Autowired
private StepBuilderFactory steps;
```

4. Define the `step1` bean, which will execute our code, from the `Task1` class:

```
@Bean
public Step step1(){
    return steps.get("step1")
            .tasklet(new Task1())
            .build();
}
```

5. Define the `job1` bean that will execute `step1`:

```
@Bean
public Job job1(){
    return jobs.get("job1")
            .start(step1())
            .build();
}
```

How it works...

We defined a `job1` job executing the `step1` step, which will call the `execute()` method in the `Task1` class.

There's more...

To execute more than one step, use the `next()` method in the job definition:

```
@Bean
public Job job1(){
    return jobs.get("job1")
            .start(step1())
            .next(step2())
            .build();
}
```

Executing a job from the command line

A simple and robust way to execute a job is to use the command-line interface. This allows you to use a standard `cron` job (use the `AT` command on Windows) to schedule it, so that the job will be executed even if the web application is down. It's also convenient for testing and debugging a job.

Getting Ready

We'll use the job defined in the *Creating a job* recipe.

How to do it...

Follow these steps to execute the job from the command line:

1. Declare the `maven-assembly-plugin` in `pom.xml` (under `build/plugins`):

```
    <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
            <archive>
                <manifest>
                    <mainClass>

org.springframework.batch.core.launch.support.
CommandLineJobRunner
                    </mainClass>
                </manifest>
            </archive>
            <descriptorRefs>
                <descriptorRef>
jar-with-dependencies</descriptorRef>
            </descriptorRefs>
        </configuration>
    </plugin>
```

2. Generate a JAR file:

```
mvn clean compile assembly:single
```

3. Execute the job by running the JAR file generated in the `target` folder, with the class where the job is defined (`BatchConfig`) and the job name (`job1`) as arguments:

```
java -jar target/springwebapp-jar-with-dependencies.jar
com.spring_cookbook.batch.BatchConfig job1
```

4. The console output should look like this:

```
...
INFO: Job: [SimpleJob: [name=job1]] launched with the
following parameters: [{}]
...
INFO: Executing step: [step1]
Starting job..
Job done..
...
```

```
INFO: Job: [SimpleJob: [name=job1]] completed with the following
parameters: [{}] and the following status: [COMPLETED]
...
```

There's more...

A job can be executed only once for a given set of parameters. To be able to execute the job again, just add a parameter using the `parameterName=parameterValue` syntax:

```
java -jar target/springwebapp-jar-with-dependencies.jar
com.spring_cookbook.batch.BatchConfig job1 p=1
java -jar target/springwebapp-jar-with-dependencies.jar
com.spring_cookbook.batch.BatchConfig job1 p=2
java -jar target/springwebapp-jar-with-dependencies.jar
com.spring_cookbook.batch.BatchConfig job1 p=3
```

In this case, the console output will look like this:

```
...
INFO: Job: [SimpleJob: [name=job1]] launched with the following
parameters: [{p=3}]
...
```

When testing and debugging the job, you can use a Unix timestamp to automatically get a different parameter value each time:

```
java -jar target/springwebapp-jar-with-dependencies.jar
com.spring_cookbook.batch.BatchConfig job1 p=`date +'%s'`
```

A job can be also be executed directly without having to generate a JAR file first:

```
mvn compile exec:java -
Dexec.mainClass=org.springframework.batch.core.launch.support.
CommandLineJobRunner -
Dexec.args="com.spring_cookbook.batch.BatchConfig job1 p=4"
```

Executing a job from a controller method

It's convenient to launch a job from a controller method when that job is triggered by a user action. For example, launching a job to process a video just uploaded by the user.

Getting ready

We'll use the job defined in the *Creating a job* recipe.

How to do it...

Follow these steps to execute the job from a controller method:

1. Add the Spring Batch configuration class to the `getServletConfigClasses()` method in your class extending `AbstractAnnotationConfigDispatcherServletInitializer`:

```
public class ServletInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[]{AppConfig.class,
BatchConfig.class};
}
```

2. In your controller class, add a `JobLauncher` attribute and `Job` attribute both autowired:

```
@Autowired
JobLauncher jobLauncher;

@Autowired
Job job;
```

3. In the controller method, define the job parameters and launch the job:

```
try {
  JobParametersBuilder jobParametersBuilder = new
JobParametersBuilder();
  jobParametersBuilder.addDate("d", new Date());

  jobLauncher.run(job,
jobParametersBuilder.toJobParameters());
} catch (Exception e) {
  ...
}
```

How it works...

We declared `BatchConfig` in the `ServletInitializer` class to make our Spring Batch configuration available to the controller methods.

In the controller method, the job parameters are the same as those in the command line.

Using job parameters

In this recipe, you'll learn how to retrieve and use a job parameter value in `Tasklet`.

Getting ready

We'll use the job defined in the *Creating a job* recipe.

How to do it...

Follow these steps to use the job parameters:

1. In the `Task1` class, add `@StepScope` to the `execute()` method:

```
@StepScope
public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext)
        throws Exception {
...
```

2. In the `execute()` method, retrieve a job parameter value by using the job parameter name:

```
String test =
(String)chunkContext.getStepContext().getJobParameters().
get("test")
```

3. Run the job with a parameter named `test`:

```
mvn compile exec:java -
Dexec.mainClass=org.springframework.batch.core.launch.
support.CommandLineJobRunner -
Dexec.args="com.spring_cookbook.batch.BatchConfig job1
test=hello"
```

How it works...

The `String` test will contain the `hello` parameter value passed on the command line. This recipe will also work if the job is launched from a controller method.

Executing a system command

A step can consist of just an execution of a system command. Spring Batch provides a convenient class for this, `SystemCommandTasklet`.

Getting ready

We'll use the job defined in the *Creating a job* recipe.

How to do it...

In Spring Batch's configuration file, add a `SystemCommandTasklet` bean. Declare the system command to be executed (here, we used the touch Unix command to create an empty file), the directory to execute it from, and the maximum time allowed for its execution:

```
@Bean
public SystemCommandTasklet task1() {
  SystemCommandTasklet tasklet = new
SystemCommandTasklet();

  tasklet.setCommand("touch test.txt");
  tasklet.setWorkingDirectory("/home/merlin");
  tasklet.setTimeout(5000);

  return tasklet;
}
```

How it works...

The `SystemCommandTasklet` class will execute a command from the working directory and kill the process if it exceeds the timeout value.

There's more...

For a more advanced use of system commands (for example, to get the output of the system command) extend `SystemCommandTasklet` and override its `execute()` method.

Scheduling a job

Some jobs need to be executed regularly-every night, every hour, and so on. Spring makes this easy with the `@Scheduled` annotation.

Getting ready

We will use the job defined in the *Creating a job* recipe.

How to do it...

Follow these steps to schedule the job:

1. If it's not done already, add the Spring Batch configuration class to the `getServletConfigClasses()` method in your class extending `AbstractAnnotationConfigDispatcherServletInitializer`:

```
public class ServletInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[]{AppConfig.class,
BatchConfig.class};
}
```

2. Add the `@EnableScheduling` annotation to the Spring Batch configuration class:

```
@Configuration
@EnableBatchProcessing
@EnableScheduling
public class BatchConfig {
...
```

3. Add an autowired `JobLauncher` field:

```
@Autowired
JobLauncher jobLauncher;
```

4. Add a method annotated with `@Scheduled` with a `fixedDelay` attribute in ms:

```
@Scheduled(fixedDelay=10000)
public void runJob1() throws Exception {
...
}
```

5. In that method, run the job:

```
JobParametersBuilder jobParametersBuilder = new
JobParametersBuilder();
jobParametersBuilder.addDate("d", new Date());
jobLauncher.run(job1(),
jobParametersBuilder.toJobParameters());
```

How it works...

The job will start getting executed again and again with a 10-second (10000 ms) interval as soon as the web application is deployed. The `job` parameter with the `new Date()` value is used to set a different parameter value for each launch.

There's more...

The `fixedDelay` attribute sets a delay of 10 seconds after a job has finished its execution before launching the next one. To actually run a job every 10 seconds, use `fixedRate`:

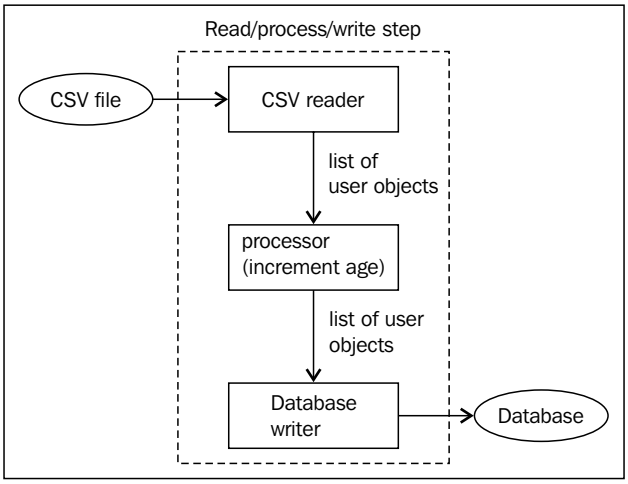
```
@Scheduled(fixedRate=10000)
public void runJob1() throws Exception {
...
}
```

It's also possible to use a regular `cron` expression:

```
@Scheduled(cron="*/5 * * * *")
public void runJob1() throws Exception {
...
}
```

Creating a read/process/write step

A read/process/write step is a common type of step where some data is read somewhere, processed in some way, and finally, saved somewhere else. In this recipe, we'll read a CSV file of users, increment their age, and save the modified users in a database as shown in the following image:



Getting ready

This is our CSV file of users, `input_data.txt`:

```
Merlin, 333
Arthur, 37
Lancelot, 35
Tristan, 20
Iseult, 22
Mark, 56
```

For each line of the CSV file, we'll create a `User` object. So, make sure that the `User` class exists:

```
public class User {
  private String firstName;
  private int age;
…
}
```

Each `User` object will be saved in the database. Make sure that the `user` table exists:

```
CREATE TABLE user  (
    id BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  first_name TEXT,
  age INT
);
```

How to do it...

Follow these steps to process the CSV file:

1. In the Spring Batch configuration class, add a method returning a `LineMapper` object, which generates an `User` object from a line in the CSV file:

```
private LineMapper<User> lineMapper() {
  DefaultLineMapper<User> lineMapper = new
DefaultLineMapper<User>();

  DelimitedLineTokenizer lineTokenizer = new
DelimitedLineTokenizer();
    lineTokenizer.setNames(new
String[]{"firstName","age"});
    lineTokenizer.setIncludedFields(new int[]{0,1});
    lineMapper.setLineTokenizer(lineTokenizer);
```

```
    BeanWrapperFieldSetMapper<User> fieldSetMapper = new
BeanWrapperFieldSetMapper<User>();
    fieldSetMapper.setTargetType(User.class);
    lineMapper.setFieldSetMapper(fieldSetMapper);

    return lineMapper;
}
```

2. Add a `reader()` method returning a `FlatFileItemReader` object, which will read a CSV file (whose path is the file path of the CSV file), and use the previously defined `LineMapper` object to generate users:

```
@Bean
@StepScope
public FlatFileItemReader<User>
reader(@Value("#{jobParameters[file]}") String csvFilePath)
{
    FlatFileItemReader<User> reader = new
FlatFileItemReader<User>();
    reader.setLineMapper(lineMapper());
    reader.setResource(new PathResource(csvFilePath));

    reader.setLinesToSkip(1);
    reader.setEncoding("utf-8");

    return reader;
}
```

3. Define a class implementing `ItemProcessor` with a `process()` method that takes a `User` object, increments its `age`, and returns the modified `User` object:

```
public class UserProcessorIncrementAge implements
ItemProcessor<User, User> {

    public User process(User user) throws Exception {
        int age = user.getAge();
        age++;
        user.setAge(age);
        return user;
    }

}
```

4. Back in the Batch configuration class, define a `UserProcessorIncrementAge` bean:

```
@Bean
private ItemProcessor<User,User> processor() {
    return new UserProcessorIncrementAge();
}
```

5. Define a `Datasource` bean with the database connection details:

```
@Bean
public DataSource dataSource() {
  DriverManagerDataSource dataSource = new
DriverManagerDataSource();

  dataSource.setDriverClassName("com.mysql.jdbc.Driver");
  dataSource.setUrl("jdbc:mysql://localhost:3306/db1");
  dataSource.setUsername("root");
  dataSource.setPassword("123");

  return dataSource;
}
```

6. Add a `writer()` bean that will take a `User` object and save it in the database:

```
@Bean
public JdbcBatchItemWriter<User> writer(){
    JdbcBatchItemWriter<User> writer = new
JdbcBatchItemWriter<User>();
    writer.setDataSource(dataSource());
    writer.setSql("INSERT INTO user (first_name, age) " +
        "VALUES ( :firstName, :age)");
    ItemSqlParameterSourceProvider<User> paramProvider =
new BeanPropertyItemSqlParameterSourceProvider<User>();

writer.setItemSqlParameterSourceProvider(paramProvider);
    return writer;
}
```

7. Add a `JobBuilderFactory` field and a `StepBuilderFactory` field, both autowired:

```
@Autowired
private JobBuilderFactory jobs;

@Autowired
private StepBuilderFactory steps;
```

8. Define a step calling our `reader()`, `processor()`, and `writer()` methods:

```
@Bean
public Step step1(){
    return steps.get("step")
            .<User,User>chunk(1)
            .reader(reader(null))
            .processor(processor())
            .writer(writer())
            .build();
}
```

9. Define a job with the previous defined step:

```
@Bean
public Job job1(){
    return jobs.get("job1")
            .start(step1())
            .build();
}
```

10. Execute the job with the path to the CSV file as parameter:

```
mvn compile exec:java -
Dexec.mainClass=org.springframework.batch.core.launch.
support.CommandLineJobRunner -
Dexec.args="com.spring_cookbook.batch.BatchConfig job1
file=input_data.txt"
```

How it works...

In the `reader()` method, we used `FlatFileItemReader`, which is a class provided by Spring Batch for reading CSV files. Each line is processed by `LineMapper`, which takes a line and returns an object. In this recipe, we used `DefaultLineMapper`, which converts a line to `Fieldset` (using `DelimitedLineTokenizer`) and then saves each field in an object (all of this is done behind the scenes by `BeanWrapperFieldSetMapper`).

In the `writer()` method, we supplied the SQL query, which will create the user in the database. The values come automatically from the `User` object, thanks to the `BeanPropertyItemSqlParameterSourceProvider` class. For example, `:firstName` will get its value from the `User` object's `firstName` field.

In the `step1()` method, we declared the reader, processor, and writer methods. The `chunk()` method allows the data to be processed and saved by groups (in chunks). This is more efficient for large sets of data.

The `@StepScope` annotation is necessary for the `reader()` and `writer()` methods, to allow them to access the job parameters. Otherwise, they are executed too early in the job initialization process.

There's more...

The reader-processor-writer separation makes it easy to swap one component with another. For example, if our CSV file becomes an XML file one day, we will only have to update the `reader()` method. In the next recipes, we will cover other types of readers and writers.

A processor is not required in a read/process/write job, so skip it if you don't need it. It also doesn't need to return an object from the same class. For example, it could take a `UserCSV` object, which would be a direct mapping of a line of the CSV file and return an actual `User` object. This would allow you to keep the CSV reader straightforward and separate the code converting its data to an actual `User` object, your real domain object, making that code easier to understand and maintain.

Our reader and writer code is short enough, so we will put it directly in the Spring Batch configuration. However, it could be moved to separate classes.

Reading an XML file

In this recipe, you'll learn to read an XML file as part of a read/process/write step.

Getting ready

We'll read this XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
    <person>
        <firstName>Shania</firstName>
    <age>49</age>
    </person>
    <person>
        <firstName>Nelly</firstName>
    <age>36</age>
    </person>
</records>
```

For each person's record in the XML file, a `User` object will be created. Make sure that the `User` class exists:

```
public class User {
  private String firstName;
  private int age;
```

How to do it...

To parse the XML file, use `StaxEventItemReader`, which is provided by Spring Batch. To generate `User` objects, use `XStreamMarshaller`, a class from the Spring Object/XML Mapping project. Follow these steps:

1. Add the Maven dependency for Spring Object/XML Mapping in `pom.xml`:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-oxm</artifactId>
  <version>${spring.version}</version>
</dependency>
```

2. Add a `reader()` method returning a `StaxEventItemReader` object to read the XML file and generate `User` objects from its contents:

```
@Bean
@StepScope
public StaxEventItemReader<User>
reader(@Value("#{jobParameters[file]}") String xmlFilePath)
{
    StaxEventItemReader<User> reader = new
StaxEventItemReader<User>();
    reader.setResource(new PathResource(xmlFilePath));
    reader.setFragmentRootElementName("person");

  XStreamMarshaller marshaller = new XStreamMarshaller();
  marshaller.setAliases(Collections.singletonMap("person",
User.class));
  reader.setUnmarshaller(marshaller);

  return reader;
}
```

3. Execute the job with the path to the XML file as a parameter. For example:

```
mvn compile exec:java -
Dexec.mainClass=org.springframework.batch.core.launch.
support.CommandLineJobRunner -
Dexec.args="com.spring_cookbook.batch.BatchConfig job1
file=input_data.xml
```

How it works...

`XStreamMarshaller` generates a `User` automatically for each person's record. This is configured with the following line:

```
marshaller.setAliases(Collections.singletonMap("person",
User.class));
```

Note that the `User` fields have to match the XML fields (`firstName` and `age`).

Generating a CSV file

Write a CSV file as part of a read/process/write step.

Getting ready

We will generate a CSV file from `User` objects. Make sure that the `User` class exists:

```
public class User {
  private String firstName;
  private int age;
```

How to do it...

Use `FlatFileItemWriter` provided by Spring Batch:

1. Add a `writer()` method that will get the fields of a `User` object, build a comma-separated line with them, and write the line to a CSV file:

```
@Bean
@StepScope
public FlatFileItemWriter<User>
writer(@Value("#{jobParameters[fileOut]}") String
csvFilePath) {
    BeanWrapperFieldExtractor<User> fieldExtractor = new
BeanWrapperFieldExtractor<User>();
```

```
    fieldExtractor.setNames(new
String[]{"firstName","age"});

    DelimitedLineAggregator<User> lineAggregator = new
DelimitedLineAggregator<User>();
    lineAggregator.setDelimiter(",");
    lineAggregator.setFieldExtractor(fieldExtractor);

    FlatFileItemWriter<User> writer = new
FlatFileItemWriter<User>();
    writer.setLineAggregator(lineAggregator);
    writer.setResource(new PathResource(csvFilePath));

    return writer;
}
```

2. Execute the job with the path to the output CSV file as a parameter:

```
mvn compile exec:java -
Dexec.mainClass=org.springframework.batch.core.launch.
support.CommandLineJobRunner -
Dexec.args="com.spring_cookbook.batch.BatchConfig job1
file=input_data.txt fileOut=output_data.txt
```

3. The resulting CSV file will look like this:

```
Merlin,334
Arthur,38
Lancelot,36
Tristan,21
Iseult,23
Mark,57
```

How it works...

`BeanWrapperFieldExtractor` extracts the declared fields (`firstName` and `age`) from the `User` object. `DelimitedLineAggregator` builds a comma-separated line with them. `FlatFileItemWriter` writes the line to the file.

Reading from a database

This recipe shows you how to read data from a database as part of a read/process/write step.

Getting ready

Each user will be read from the database. Make sure that the `user` database table exists with some data in it:

```
CREATE TABLE user  (
    id BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  first_name TEXT,
  age INT
);
```

For each user row in the database, we'll create a `User` object. Make sure that the `User` class exists:

```
public class User {
  private String firstName;
  private int age;
```

Make sure that the `Datasource` bean is defined with the database connection information.

How to do it...

Add a `reader()` method returning `JdbcCursorItemReader`-a class provided by Spring Batch:

```
@Bean
@StepScope
public JdbcCursorItemReader<User> reader() {
  JdbcCursorItemReader<User> reader = new
JdbcCursorItemReader<User>();
  reader.setDataSource(dataSource());

  reader.setSql("SELECT first_name, age FROM user");

  reader.setRowMapper(new
BeanPropertyRowMapper<User>(User.class));

  return reader;
}
```

How it works...

A SQL query is executed to get users from the database. `BeanPropertyRowMapper` generates `User` objects from the result. Note that the SQL result's columns (`first_name`, `age`) have to match the User fields (`firstName` and `age`). If the database table has different column names, use SQL aliases to ensure that:

```
SELECT name1 as first_name, the_age as age FROM user
```

Unit testing batch jobs

Spring Batch provides different ways to test a batch job; the whole job, only one step, or just a `Tasklet` class can be tested.

How to do it...

Follow these steps to unit test batch jobs:

1. Add the Maven dependency for `spring-batch-test` in `pom.xml`:

```
<dependency>
  <groupId>org.springframework.batch</groupId>
  <artifactId>spring-batch-test</artifactId>
  <version>3.0.2.RELEASE</version>
</dependency>
```

2. In the unit test class, if using JUnit, load the Spring Batch configuration class like this:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {BatchConfig.class})
public class BatchJob1Test {
...
```

3. If using TestNG, load the Spring Batch configuration class as follows:

```
@ContextConfiguration(classes = {BatchConfig.class})
public class BatchJob1Test extends
AbstractTestNGSpringContextTests {
...
```

4. Add an autowired `JobLauncherTestUtils` field:

```
@Autowired
private JobLauncherTestUtils jobLauncherTestUtils;
```

5. This is how you can test an entire job, check its exit status, and the number of steps that were executed:

```
@Test
public void testJob() throws Exception {
    JobExecution jobExecution =
jobLauncherTestUtils.launchJob();
    Assert.assertEquals(ExitStatus.COMPLETED,
jobExecution.getExitStatus());
    Assert.assertEquals(1,
jobExecution.getStepExecutions().size());
}
```

6. This is how you can test a specific step:

```
@Test
public void testStep() throws Exception {
    JobExecution jobExecution =
jobLauncherTestUtils.launchStep("step1");
    Assert.assertEquals(ExitStatus.COMPLETED,
jobExecution.getExitStatus());
}
```

7. This is how you can test Tasklet:

```
@Test
public void testTasklet() throws Exception {
    Task1 task1 = new Task1();
    Assert.assertEquals(RepeatStatus.FINISHED,
task1.execute(null, null));
}
```

How it works...

The Spring Batch configuration class has to be loaded, so that the test methods can access the job and its steps. `JobLauncherTestUtils` is a helper class that is used to easily execute a job or one of its steps.