CHAPTER 2

Spring Bean Lifecycle and Configuration

The Spring Framework provides an easy way to create, initialize, and connect software components into practical, decoupled, easy-to-test, enterprise-ready applications. Every software application consists of software components that interact, collaborate, and depend on other components to successfully execute a set of tasks. The relationships between components are established during the design phase, and linking dependents with their dependencies is called *dependency injection*. Spring provides a very simplistic way to define the connections between them to create an application.

Before Spring entered the picture, defining connections between classes and composing them required development to be done by following different design patterns, including *Factory, Abstract Factory, Singleton, Builder, Decorator, Proxy, Service Locator,* and Reflection[1] (which is not an option in Java 9+ unless the module is configured to support it).

Spring was built to make dependency injection easy. This software design pattern implies that dependent components delegate the dependency resolution to an external service that will take care of injecting the dependencies. The dependent component is not allowed to call the injector service and has very little to say when it comes to the dependencies that will be injected. This is why the behavior is also known as the "*Don't call us, we'll call you!"* principle, and it is technically known as *inversion of control (IoC)*. If you do a quick Google search, you will find a lot of conflicting opinions about dependency injection and inversion of control. You will find programming articles calling them programming techniques, programming principles, and design patterns.

[1]If you are interested in more books about Java Design Patterns, you can check out this book from Apress: https://www.apress.com/gp/book/9781484240779.

I. Cosmina, *Pivotal Certified Professional Core Spring 5 Developer Exam*,
https://doi.org/10.1007/978-1-4842-5136-2_2

I recommend an article by Martin Fowler (see https://martinfowler.com/articles/injection.html#InversionOfControl), which is recognized in the Java world as the highest authority when it comes to design patterns. If you do not have the time to read it, here is a summary: *Inversion of control is a common characteristic of frameworks that facilitate injection of dependencies. And the basic idea of the dependency injection pattern is to have a separate object that injects dependencies with the required* ***behavior****, based on an interface contract.*

The software components that Spring uses to build applications are called *beans*, which are *Plain Old Java Objects* (POJOs) that are created, assembled (dependencies are injected), initialized, and managed by the Spring *IoC* container and located in the Spring application context. The order of these operations and the relationships between objects are provided to the Spring IoC container using XML configuration files prior to Spring version 2.5. Starting with 2.5, a small set of annotations was added for configuring beans, and with Spring 3, Java configuration was introduced, and a Spring application could be configured without any XML needed at all. In Spring 4, even more configuration annotations were introduced, most of them specialized for an application type (e.g., JPA, WEB), as if the intention was to remove the need for XML configuration completely, which eventually happened in Spring 5.

This chapter covers everything a developer needs to know to configure a basic Spring application using Java configuration. A few code samples using XML configuration are covered just to give you an idea of the evolution of the Spring configuration style. From now on, each chapter contains a section that covers how to configure a project with Spring Boot. Spring Boot is a project that makes it very practical to create stand-alone, production-grade, Spring-based applications that you can "just run," reducing a developer's effort when configuring an application.

When configuring Spring applications, there are typical groups of infrastructure beans that have to be configured in a certain way, depending on the application we are building. After years of Spring applications being built, a pattern of configuration has emerged. When the same configuration is used in 90% of the applications written, this makes a good case for favoring *convention over configuration*. This is a software design paradigm used by software frameworks; it attempts to decrease the number of decisions that a developer using the framework is required to make without necessarily losing flexibility. **Spring Boot is the epitome of convention over configuration**.

Old-Style Application Development

In the most competent development style, a Java application should be composed of POJOs—simple Java objects, each with a single responsibility. In the previous chapter, the entity classes that will be used throughout the book were introduced along with the relationships between them. To manage this type of object at the lowest level, the *DAO (repository) layer* of the application—classes called *repositories*, will be used. The purpose of these classes is to retrieve, update, create, and delete entities from the storage support, which usually is some type of database.

In the code for this book, the names of the *repositories* are created by concatenating a short denomination for the type of entity management, the name of the entity object being managed, and the *Repo* postfix. For example, a class managing `Person` entities should be named `PersonRepo`. Each class implements a simple interface that defines the methods to be implemented to provide the desired behavior for that entity type. All interfaces extend a common interface declaring the common methods that should be implemented for any type of entity. For example, saving, searching by ID, and deleting should be supported for every type of entity. This method of development is used because Java is a very object-oriented programming language, and in this case, the inheritance principle is very well respected. Also, generic types make possible such a level of inheritance.

In Figure 2-1, the *AbstractRepo* interface and the child interfaces for each entity type are depicted. You can see how the *AbstractRepo* interface defines typical method skeletons for every entity type, and the child interface defines the method skeletons designed to work with only a specific type of entity.

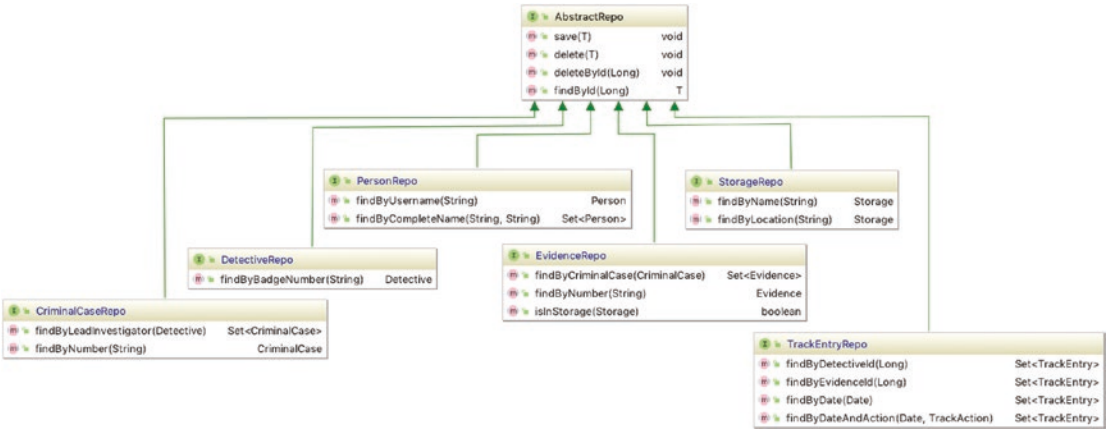

Figure 2-1. *Repository interfaces hierarchy*

At the end of this section, you can take a break from reading to get comfortable with this implementation. The project is named `pojos-practice`, and you can find it under `chapter02`. It contains stub[2] implementations for the repository classes, which can be found in the test sources under the `com.apress.cems.pojos.repos.stub` package. In Figure 2-2, the stub classes are depicted. Again, inheritance was used to reduce the amount of code and avoid writing duplicate code.

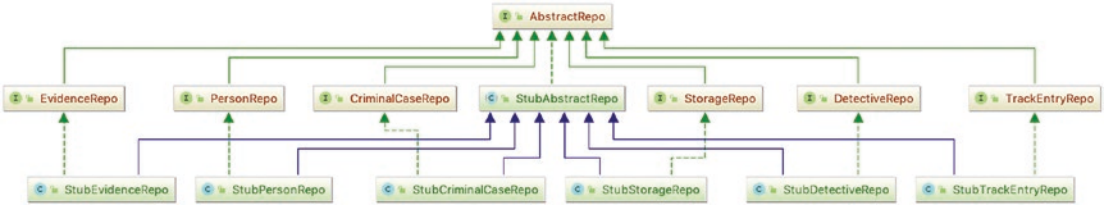

Figure 2-2. *Repository stub implementations*

The stub repositories store all the data created by the user in a map data structure named `records`. The unique ID for each record is generated based on the size of this map. The implementation is in the `StubAbstractRepo` class, which is depicted as follows with all the basic repository operations.

```
package com.apress.cems.pojos.repos.stub;

import com.apress.cems.dao.AbstractEntity;
import com.apress.cems.pojos.repos.AbstractRepo;

import java.util.HashMap;
import java.util.Map;

public abstract class StubAbstractRepo <T extends AbstractEntity>
      implements AbstractRepo<T> {

    protected Map<Long, T> records = new HashMap<>();

    @Override
    public void save(T entity) {
        if (entity.getId() == null) {
```

[2]In software development a stub is a piece of code used to stand in for actual functionality to help test another desired functionality in isolation.

```
            Long id = (long) records.size() + 1;
            entity.setId(id);
        }
        records.put(entity.getId(), entity);
    }

    @Override
    public void delete(T entity) {
        records.remove(entity.getId());
    }

    @Override
    public void deleteById(Long entityId) {
        records.remove(entityId);
    }

    @Override
    public T findById(Long entityId) {
        return records.get(entityId);
    }
}
```

The next layer after the DAO (repository) layer is the *service layer*. This layer is composed of classes doing modifications to the entity objects before being passed on to the repositories for persisting the changes to the storage support (database). The service layer is the bridge between the web layer and the DAO layer and will be the main focus of the book. It is composed of specialized classes that work together to implement behavior that is not specific to web or data access. It is also called *the business layer*, because most of the application business logic is implemented here. Each service class implements an interface that defines the methods that it must implement to provide the desired behavior. Each service class uses one or more repository fields to manage entity objects. Typically, for each entity type, a service class also exists, but more complex services can be defined that can use multiple entity types to perform complex tasks. In the code for this book, a complex service class is the `SimpleOperationsService`, which contains methods useful for executing common operations such as create a `CriminalCase` record, assign a lead investigator to it, link the evidence, and solve the case.

In Figure 2-3, all the service classes and interfaces are depicted.

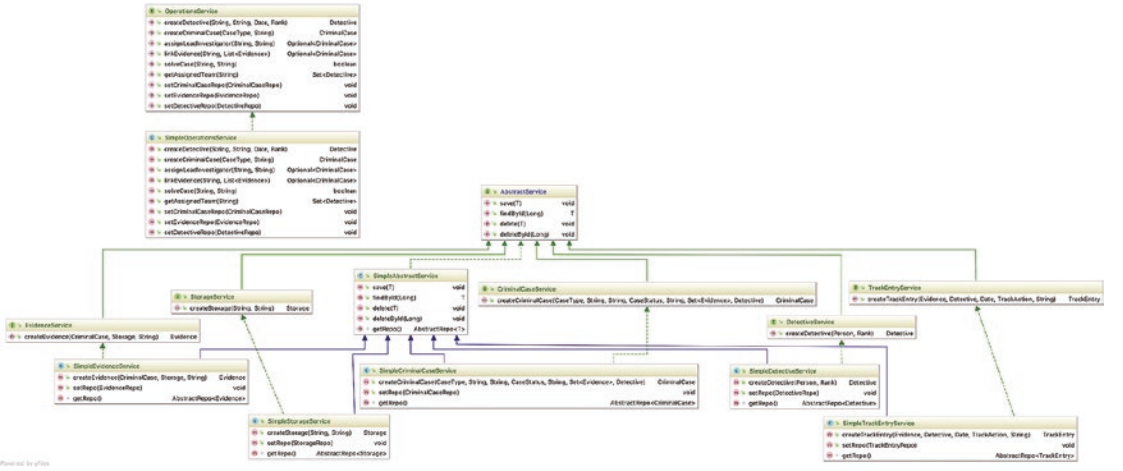

Figure 2-3. *Service interfaces and implementations*

All the classes presented here are parts that will be assembled to create an application that will manage criminal cases data. In a production environment, a service class needs to be instantiated, and a repository instance must be set for it so data can be managed properly. Applications running in production support complex operations such as transactions, security, messaging, remote access, and caching. To test them, pieces of them have to be isolated, and some of them that are not the object of testing are replaced with simplified implementations. In a test environment, stub or mock[3] implementations can replace implementations that are not meant to be covered by the testing process. In Figure 2-4, you can see a service class and a dependency needed for it in a production and test environment side by side.

[3]In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways.

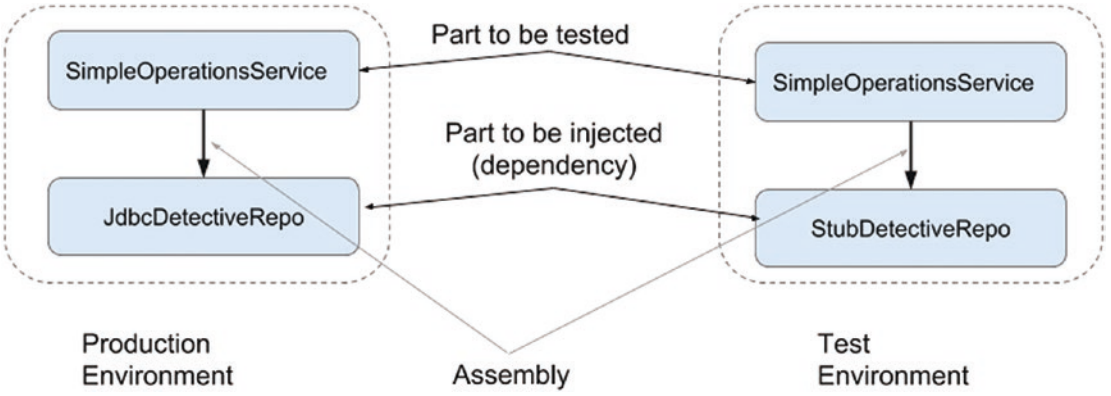

Figure 2-4. *Service class and its dependency in different running environments*

Both `JdbcDetectiveRepo` and `StubDetectiveRepo` classes implement the same interface, named `DetectiveRepo`.

This is practical because the interface type can be used when declaring references so that concrete implementations can be easily swapped one with another. The code snippet relevant to the previous image is depicted as follows, which is a piece of the `SimpleOperationsService` class definition.

```
package com.apress.cems.pojos.services.impl;

import com.apress.cems.pojos.repos.DetectiveRepo;

public class SimpleOperationsService
    implements OperationsService {

    private DetectiveRepo detectiveRepo;
    ...

    public void setDetectiveRepo(DetectiveRepo detectiveRepo) {
        detectiveRepo = detectiveRepo;
    }
}
```

The dependency is defined using the interface type, `DetectiveRepo` interface, so any implementation is a suitable dependency. So in a production environment, an instance of type `JdbcDetectiveRepo` will be provided, and that type will be defined to implement the `DetectiveRepo` interface.

```
public class JdbcDetectiveRepo extends JdbcAbstractRepo<Detective>
                 implements DetectiveRepo {
    //implementation not relevant at this point
...
}
```

The creation of an instance of type `SimpleOperationsService` requires the following steps.

1. Instantiate and initialize the repository instance.

   ```
   DetectiveRepo detectiveRepo = new JdbcDetectiveRepo(...);
   ```

2. Instantiate the service class.

   ```
   OperationsService service = new SimpleOperationsService();
   ```

3. Inject the dependency.

   ```
   service.setDetectiveRepo(detectiveRepo);
   ```

In a test environment, a mock or a stub will do, as long as it implements the same interface.

```
public class StubDetectiveRepo extends StubAbstractRepo<Detective>
          implements DetectiveRepo {
    //implementation not relevant at this point
...
}
```

For the test environment, the assembly steps are the same.

```
1.   DetectiveRepo detectiveRepo = new StubDetectiveRepo(...);
2.   OperationsService service = new SimpleOperationsService();
3.   service.setDetectiveRepo(detectiveRepo);
```

Spring makes assembling the components a very pleasant job. Swapping them (depending on the environment) is also possible in a practical manner, which is supported by the fact that the two types are linked together by implementing the same interface. Because connecting components is so easy, writing tests becomes a breeze also, since each part can be isolated from the others and tested without any unknown influence. Spring provides support for writing tests via the spring-test.jar library, but that will be the topic of Chapter 3.

And now that you know what Spring can help you with, you are invited to have a taste of how things are done without it. Take a look at the `pojo-practice` project. In the `SimpleOperationsService` class, there is a method named `createResponse` that needs an implementation. The following are the steps to create a `CriminalCase` instance.

1. Retrieve the `Detective` instance using `detectiveRepo`.
2. Save the `Evidence` instance collection using `evidenceRepo` and retrieve `Storage` instances using `storageRepo`.
3. Instantiate a `CriminalCase` instance.
4. Populate the `CriminalCase` instance.
    a. Set the `shortDescription` property.
    b. Set the `caseType` property.
    c. Set the `leadInvestigator` property to the detective instance.
    d. Save the `CriminalCase` instance using the `criminalCaseRepo`.
    e. Add all `Evidence` instances to the `CriminalCase` instance.
    f. Save all `Evidence` instances using the `evidenceRepo`.

Figure 2-5 depicts the sequence of operations.

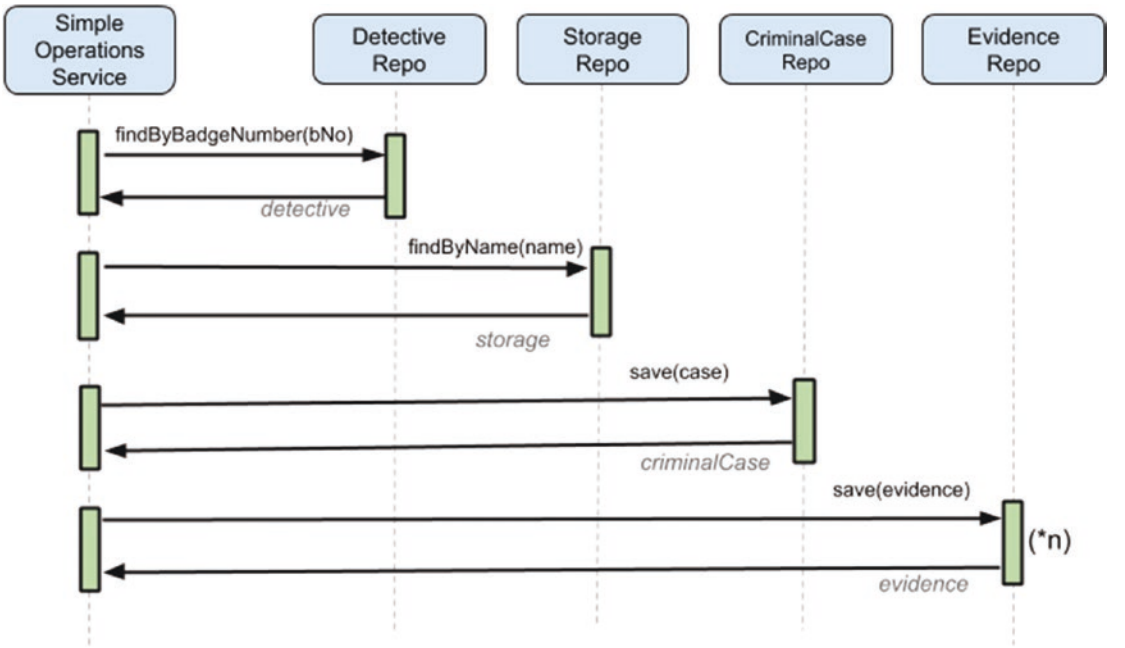

Figure 2-5. *SimpleOperationsService create criminal case abstract functional diagram. The (∗n) stands for multiple calls of a method*

! In this example, because we are assuming pure Java implementation with no frameworks, there is no dependency to handle persistency to the database, so the `Evidence` instances must be saved explicitly. In a real, complex implementation a persistency framework, such as Hibernate will propagate the save operation from the `CriminalCase` instance to the `Evidence` instances linked to it.

Because Java 11 is used, `Optional` has been declared as a wrapper class for results returned by repository implementations to avoid `NullPointerException`. You can create a `CriminalCase` without a lead investigator, because this is a position that can be filled later, so take that into consideration when writing your implementation.

To run the implementation, search for the class `com.apress.cems.pojos.services.SimpleOperationsServiceTest` under the test directory of the `pojo-practice` project. Inside this class there is a method annotated with `@Test`. This is a JUnit annotation. More information about testing tools is covered in Chapter 3. To run a unit test in IntelliJ IDEA,

just right-click the method name, and a menu like the one in Figure 2-6 is displayed. Select the Run option to run the test. Select Debug if you want to run the test in debug mode and check field values.

Figure 2-6. *JUnit test contextual menu in IntelliJ IDEA*

If the implementation is not correct, the test will fail, and in the IntelliJ IDEA console you should see something similar to what is depicted in Figure 2-7. And yes, a lot of messages written in red is a clear sign that the test failed.

Figure 2-7. *JUnit test failure in IntelliJ IDEA*

If the implementation is correct, the test will pass, and there will not be much red in the IntelliJ IDEA console, exactly as depicted in Figure 2-8.

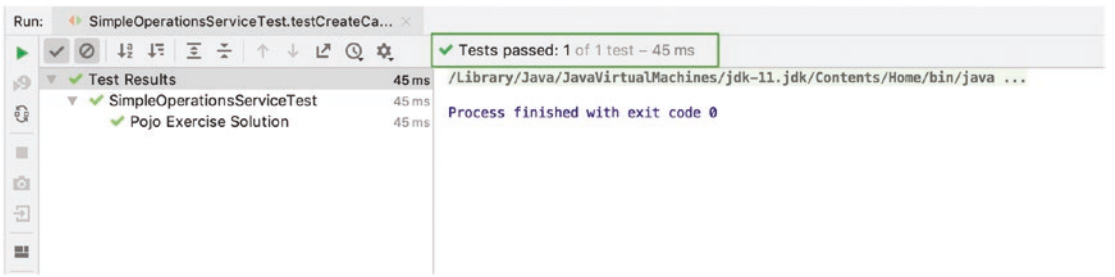

Figure 2-8. *JUnit test passed in IntelliJ IDEA*

You can check the solution by comparing it with the proposed code from the `chapter02/pojos` project.

Spring IoC Container and Dependency Injection

The Spring Framework IoC component is the nucleus of the framework. It uses dependency injection to assemble Spring-provided (also called infrastructure components) and development-provided components to rapidly wrap up an application. Figure 2-9 depicts where the Spring IoC container fits in the application development process; the option of providing configurations using XML is kept because it is still supported.

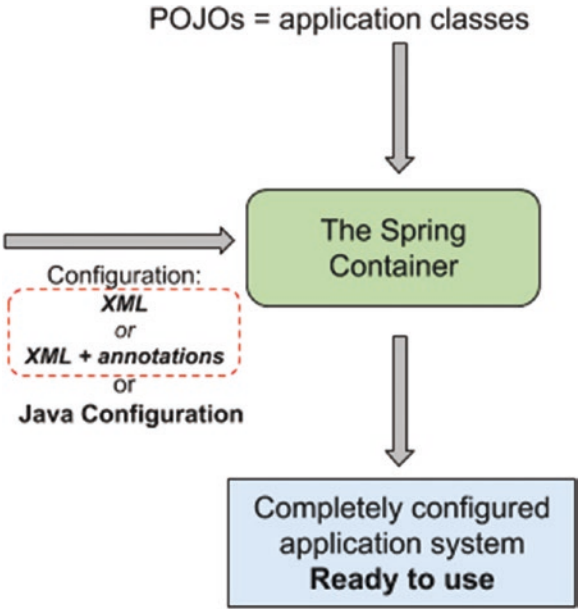

Figure 2-9. *Spring IoC Container purpose*

So, the Spring IoC container is tasked with the responsibility of connecting beans together to build a working application and it does so by reading a configuration provided by the developer. The Spring IoC container is thus an external authority that passes a dependency to a dependent object that will use it. Providing dependencies, a process called *injection*, happens at runtime, when the application is being put together after being compiled, and this allows a lot of flexibility, because the functionality of an application can be extended by modifying an external configuration without a full recompile of the application.

> **!** For example, some parameter values can be declared into files with the `*.properties` extension, which are not packaged into the application, but their location can be provided when the application starts. This allows those files to be modified, and reloaded while the application is running. If those values happen to specify concrete types to be injected, they will require an application context restart.

Having an external responsible for injecting dependencies allows very loosely coupled applications to be built. And since low coupling often correlates with high cohesion, Spring applications are very easy to navigate through, very easy to test and maintain.

The application being developed over the course of this book includes service classes that are built using repository instances. For example, this is how a `DetectiveService` implementation can be defined.

```
public class SimpleDetectiveService extends SimpleAbstractService<Detective>
                implements DetectiveService {
    private DetectiveRepo repo;

    public SimpleDetecitveService(DetectiveRepo detectiveRepo) {
        this.repo = detectiveRepo;
    }
    ...
}
```

As you can see, the dependency is injected using a constructor, so creating the service instance requires a repository instance to be provided as a parameter. The repository instance is needed to retrieve and persist `Detective` objects in the database. The repository can be defined like the following.

```
import javax.sql.DataSource;

public class JdbcDetectiveRepo extends JdbcAbstractRepo<Detective>
       implements DetectiveRepo  {

    public JdbcDetectiveRepo(DataSource dataSource) {
        super(dataSource);
    }
    ...
}
```

```
//JdbcAbstractRepo.java contains common
//implementation for all repository classes
import javax.sql.DataSource;

public class JdbcAbstractRepo<T extends AbstractEntity>
      implements AbstractRepo<T> {
    protected DataSource dataSource;

    public JdbcAbstractRepo(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public void save(T entity) {...}

    @Override
    public void delete(T entity) {... }

    @Override
    public void deleteById(Long entityId) {...}

    @Override
    public T findById(Long entityId) {...}
}
```

A repository instance requires a `DataSource` instance to be injected into it, which is used to connect to the database.

The Spring IoC container decides how to create these objects and how to link them together based on the configuration.

Although XML is no longer a topic for the Spring certification exam, the next section will cover a basic Spring XML configuration file, just to give you an idea of how things used to be done before Spring 5.

Providing Configuration via XML Files

The configuration in Figure 2-10 depicts the contents and template of an XML Spring Configuration file used to define the components that will make up the application. The beans that are being injected into other beans and their locations are made obvious by the dotted red arrows.[4]

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="simpleDetectiveService"
             class="com.apress.cems.pojos.services.impl.SimpleDetectiveService">
       <property name="repo" ref="detectiveRepo"/>
   </bean>

   <bean id="detectiveRepo" class="com.apress.cems.xml.repos.impl.JdbcDetectiveRepo">
        <property name="dataSource" ref="dataSource"/>
   </bean>

   <bean id="dataSource" class="oracle.jdbc.pool.OracleDataSource">
       <property name="URL" value="jdbc:oracle:thin:@localhost:1521:xe"/>
       <property name="user" value="prod"/>
       <property name="password" value="prod"/>
       <property name="loginTimeout" value="300"/>
   </bean>
</beans>
```

Figure 2-10. *Application context and the beans managed by it*

The XML file is usually placed in the project directory under `src/main/resources/spring`. The name is not mandatory, and you can use any name you wish, or no directory at all, Spring configuration can be located directly under the `resources` directory if you so desire. The `spring` directory tells you that files under it are Spring configuration files, because an application can have more XML configuration files for other purposes

[4]Oracle was used for data storage in this example because most production applications use Oracle for storage, and this book aims to provide real configurations such as you will probably encounter and need while working in software development. If you want to try this configuration, you can do so for a better understanding of Spring. A README file is provided in the project to instruct how to install Docker and use Oracle in a Docker container to avoid installing Oracle on your machine, because we all know how much of a pain that is.

(configuring other infrastructure components like Hibernate, configuring caching with Ehcache, logging, etc.). All the beans declared in the previous configuration will be used to create an application context.

In the previous example, a lot of information is new for you if you are looking into Spring for the first time. Unfortunately, it is also deprecated (sort of) as the preferred way to configure Spring applications starting with Spring 5 is using Java configuration. But before getting into that, it is important to explain the XML configuration a little. XML configuration is still supported for Spring 5, but the elements available for creating a configuration are specific to Spring 4, no new XML elements (like `<bean ../>`), nor namespaces were added in Spring 5. When writing a Spring XML configuration file, the elements you are allowed to use are defined by special namespaces each containing element definitions grouped by purpose.

The following is a list with the namespaces that you are most likely to find in Spring applications written to use XML configuration files. In Spring 5, everything can be configured using specialized annotations and Java configuration.[5]

- `beans`: Also known as the core namespace, this is the only configuration needed to create a basic Spring application configuration. All the versions of this namespace are publicly available at `www.springframework.org/schema/beans/`. The most recent is Spring version 4.3. This namespace is the only one used in the previous example, because the configuration is quite simple and does not require anything else.
- `context`: Defines the configuration elements for the Spring Framework's application context support, basically extends the beans namespace with elements that make configuration more practical to write. All the versions of this namespace are publicly available at `www.springframework.org/schema/context/`.
- `util`: Provides the developer utility elements, such as elements for declaring beans of `Collection` types, accessing static fields, and so forth. All the versions of this namespace are publicly available at `www.springframework.org/schema/util/`.

[5]You can see the full list of namespaces available by accessing this url: `http://www.springframework.org/schema/`

- `aop`: Provides the elements for declaring aspects. All the versions of this namespace are publicly available at `www.springframework.org/schema/aop/`.
- `jdbc`: Provides the elements for declaring embedded databases useful for testing without a full-blown database. All the versions of this namespace are publicly available at `www.springframework.org/schema/jdbc/`.
- `tx`: Provides the elements for declaring transactional behavior. All the versions of this namespace are publicly available at `www.springframework.org/schema/tx/`.
- `jee`: Provides the elements useful when writing an application that uses JEE components such as EJBs. All the versions of this namespace are publicly available at `www.springframework.org/schema/jee/`.
- `jms`: Provides the elements to configure message driven beans. All the versions of this namespace are publicly available at `www.springframework.org/schema/jms/`.
- `mvc`: Provides the elements to configure Spring web applications (controllers, interceptors, and view components). All the versions of this namespace are publicly available at `www.springframework.org/schema/mvc/`.
- `security`: Provides the elements to configure Secured Spring applications. All the versions of this namespace are publicly available at `www.springframework.org/schema/security/`.

When writing XML configuration files, it is a recommended practice not to use the version of the namespace in its `schemaLocation`, so that the version of the namespace will be picked up automatically based on the Spring version on the classpath. In the configuration provided as an example in the `chapter02/xml` project, the version picked up is declared within the `spring-beans.jar` found in the classpath. And if you open that jar and look into the `spring-beans.xsd` file, you will see that the version in the file is Spring 4.3.

An application context is an instance of any type implementing `org.springframework.context.ApplicationContext`, which is the central interface for providing configuration for a Spring application. The application context will manage all

objects instantiated and initialized by the Spring IoC container, which from now on, I will refer to it as beans to get you accustomed to the Spring terminology. The relationship among these objects and the application context is depicted in Figure 2-11 along with their unique identifier.

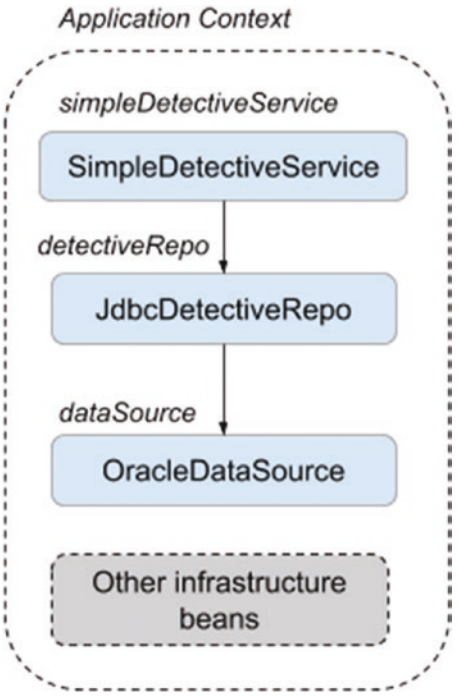

Figure 2-11. *Application context and the beans managed by it*

There are more implementations for the application context provided by Spring, and the one to use depends on the location and the resources containing the configuration. For XML, the class `org.springframework.context.support.ClassPathXmlApplicationContext` is used.

```
//creating the context
(1)ApplicationContext context = new ClassPathXmlApplicationContext
                ("classpath:spring/application-cfg-prod.xml");

// Get the bean to use to invoke the service
(2)DetectiveService detectiveService =
    (DetectiveService)context.get("simpleDetectiveService");
```

```
// create user entity
(3)Detective detective = new Detective();
// populate detective

// invoking the save method of the bean
(4)detectiveService.save(detective);
```

`classpath` is a common prefix used in Spring applications configured using XML files; it tells the Spring IoC container where the configuration is located. Any path declared with the `classpath:` prefix is relative to the `src/main/resources/` directory. The bean unique identifier in the application context, the bean `id` was evidentiated by underlining its value to give you a hint as to how the identification of a certain instance is done. More on this topic is covered in the following sections.

Using Java Configuration

The beans definitions that make up a Spring application are provided using XML files, annotations, Java-based Configuration, or all of them together. When Spring 1.0 was released in 2004, it supported only XML as a method of configuration. The annotation concept was not even invented yet.

I remember that the first time I had contact with Spring was in 2006. To a young coder eager to learn to write Java code, writing applications using XML did not seem appealing. As soon as the idea of annotations emerged, Spring adopted it and rapidly provided its own annotations (the stereotypical annotations: `@Component` and its specializations, `@Service` and `@Repository`, etc.) to make configuring Spring applications more practical. This happened in 2007, when Spring 2.5 was released. In this version, XML was still needed. Starting with Spring 3.0 in 2009 and the introduction of Java configuration (annotations `@Configuration` and `@Bean`), a configuration method based on annotations placed inside the Java code, XML became expendable. The stereotype annotations and the Java configuration annotations complement each other to provide a practical, non-XML way to define the configuration for a Spring application. XML configuration is still supported because of legacy code and because XML might still be suitable for certain application configurations. Indeed, there are programmers who still prefer to completely decouple all configurations from the code. Small XML configuration snippets can still be found in the official Spring Reference Documentation, and if you have any questions unanswered by this book, you can look for the answers there.[6]

[6]Official Spring Reference Documentation here: https://docs.spring.io/spring/docs/current/spring-framework-reference/

In this section, all aspects of configuration and dependency injection types will be covered, so get yourself a big cup of coffee (or tea) and start reading.

For developer bean definitions to be discovered and created by the Spring IoC container, many Spring-provided beans (the infrastructure beans) must be created too. That is why a few core Spring modules must be added as dependencies to your project.

- `spring-core`: The fundamental parts of the Spring Framework, basic utility classes, interfaces, and enums that all other Spring libraries depend on.
- `spring-beans`: Together with `spring-core` provide the core components of the framework, including the Spring IoC container and dependency Injection features.
- `spring-context`: Expands the functionality of the previous two, and it contains components that help build and use an application context. The `ApplicationContext` interface is part of this module, being the interface that every application context class implements.
- `spring-context-support`: Provides support for integration with third-party libraries; for example, Quartz, FreeMarker, and a few more.
- `spring-expressions`: Provides a powerful expression language (Spring Expression Language, also known as SpEL) used for querying and manipulating objects at runtime; for example, properties can be read from external sources decided at runtime and used to initialize beans. But this language is quite powerful, since it also supports logical and mathematical operations, accessing arrays, and manipulating collections.

But you do not need to bother that much with the libraries required in your classpath, as everything has been taken care of for you. The project attached to this book is a Gradle multimodule project with configuration files that are very easy to read and easy to customize to your personal projects. Also, since we are using Java 11, each module of the project has its own `module-info.java` configuration files, containing all

required module configurations. Mentions of specific module configurations appear in the book, so you might want to take a look at the module configuration files, even if they are not a topic for the exam.

Let's start with an overview of the annotations used in this book and in the code attached to it.

The Annotations

The core annotation in Spring is the `@Component` from the `org.springframework.stereotype` package. This annotation marks a class from which a bean will be created. Such classes are automatically picked up using annotation-based configuration (classes annotated with `@Configuration`) and classpath scanning (enabled by annotating a configuration class with `@ComponentScan`). In Figure 2-12, the most important annotations used in this book are depicted and are grouped by their purpose. (Spring Boot annotations are not part of this diagram).

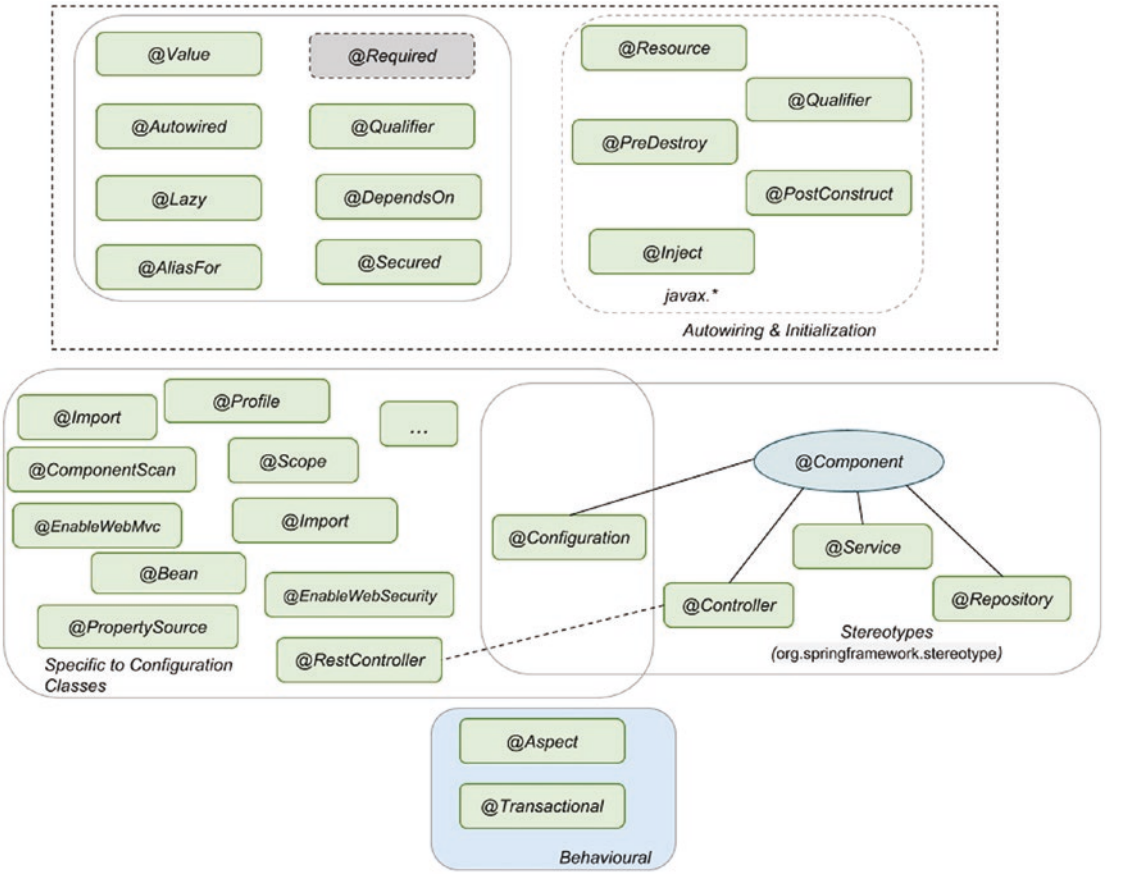

Figure 2-12. *Most important annotations used in this book*

- Stereotype annotations are used to mark classes according to their purpose.
  - `@Component:` template for any Spring-managed component(bean).
  - `@Repository:` template for a component used to provide data access, specialization of the `@Component` annotation for the DAO layer.
  - `@Service:` template for a component that provides service execution, specialization of the `@Component` annotation for the Service layer.

  - `@Controller`: template for a web component, specialization of the `@Component` annotation for the web layer.[7]
  - `@Configuration`: configuration class containing bean definitions (methods annotated with `@Bean`).[8]

- Autowiring and initialization annotations are used to define which dependency is injected and what the bean looks like. For example.
  - `@Autowired`: core annotation for this group; is used on dependencies to instruct the Spring IoC to take care of injecting them. Can be used on fields, constructors, setters and even methods mentioned in the **Injection Types** section. Use with `@Qualifier` from Spring to specify name of the bean to inject.
  - `@Inject`: equivalent annotation to `@Autowired` from `javax.inject` package. Use with `@Qualifier` from `javax.inject` to specify name of the bean to inject.
  - `@Resource`: equivalent annotation to `@Autowired` from `javax.annotation` package. Provides a `name` attribute to specify name of the bean to inject.
  - `@Required`: Spring annotation that marks a dependency as mandatory. It can be used on setter methods, but since Spring 5.1 was deprecated as of in favor of using constructor injection for required settings.
  - `@Lazy`: dependency will be injected the first time it is used. Although this annotation exists, avoid using it if possible. When a Spring application is started `ApplicationContext` implementations

[7]The *@Component*, *@Repository*, *@Service,* and *@Controller* are part of the *org.springframework.stereotype* package and are the core annotations for creating beans. You will find them referred in official documentation and in this book as *the stereotype annotations*.

[8]There is quite a controversy regarding if the *@Configuration* annotation is a stereotype annotation or not. According to the Spring Reference Documentation we could consider that *@Configuration* is a marker for any class that fulfils the role or stereotype of a configuration class. *@ComponentScan* is able to detect it according a technical perspective because it has *@Component*, but semantically would have no sense, because a bean of type configuration will rarely be used for anything else than bean declaration. Also, because it is not part of the Spring stereotype package, we could conclude that this annotation is not a stereotype annotation.

eagerly create and configure all singleton beans as part of the initialization process, this is useful because configuration errors in the configuration or supporting environment (e.g., database) can be spotted fast. When `@Lazy` is being used spotting these errors might be delayed.

- Annotations that appear only in (and on) classes annotated with `@Configuration` are infrastructure specific; they define the configuration resources, the components, and their scope.
- Behavioral annotations are annotations that define behavior of a bean. They might as well be named proxy annotations, because they involve proxies being created to intercept requests to the beans being configured with them.

JSR 250 annotations contained in JDK, package `javax.annotations` are supported. Also in Spring 3, support for JSR 330[9] was added, and some Spring annotations analogous of some annotations in JSR 330. A complete list of the annotations in each package is depicted in Figure 2-13, and on the right, you can see a few Spring annotations connected using red lines to the analogous annotations in JSR 330. Most of the annotations in the JSRs are Java annotations that provide a minimum behavior of the Spring annotations.

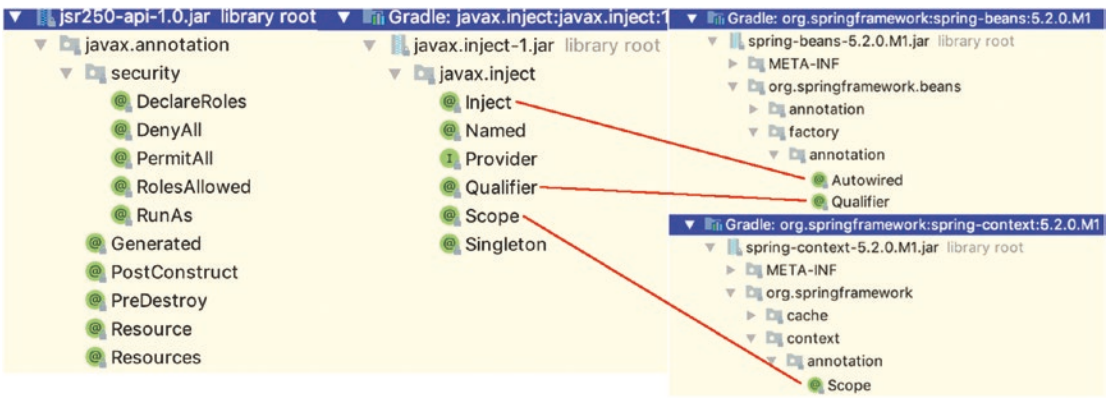

Figure 2-13. *JSR 250 and JSR 330 annotations supported by Spring*

Now that the stars of the book have been introduced, let the show begin!

[9]Extension of the Java dependency injection API at https://jcp.org/en/jsr/detail?id=330.

Spring Configuration Classes and the Application Context

The most important annotation when creating Java configuration classes for Spring applications is the `@Configuration` annotation. Classes annotated with this annotation either contain bean declarations(`@Bean`) or can be further configured by adding extra annotations (e.g., `@Profile`, `@Scope`) to tell the Spring IoC container how the bean declarations can be found.

Any Spring application has at its core one or more configuration classes. These classes either contain bean declarations or are configured to tell the Spring IoC container where to look for bean declarations. Configuration classes can be combined with XML configuration files. These classes can be bootstrapped[10] in many ways, depending on the configuration setup.

The simplest Spring configuration class is just an empty Java class annotated with the `@Configuration`.

```
package com.apress.cems.beans.simple;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SimpleConfig {
}
```

This class can create the Spring application by creating an application context based on it.

[10]Bootstrapping in Spring means loading an application context.

! But before starting Spring applications, we have to enable more detailed logging so we can see what Spring is doing under the hood, because there is no better way to learn and understand Spring than looking at it in action. All applications in the book use Logback as the logging library and in every `resource` directory there is a configuration file named `logback.xml` with the logging configuration for the application. For test modules, which is where all our test classes reside, the configuration file is named `logback-test.xml`. If you open any of these files you might notice the next configuration lines.

```
<logger name="com.apress.cems" level="debug"/>
<logger name="org.springframework" level="trace"/>
```

The first line configures the level of logging for the package where the book sources are. The second line configures the log level for the Spring packages. Logback offers seven levels of logging: `OFF`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`, and `ALL`, which are represented by numeric values in the `ch.qos.logback.classic.Level` enum. `ALL`, which is the most granular, is associated with `Integer.MIN_VALUE` and enables logging messages of any level. At the other end of the interval is the `OFF` value, which is associated with `Integer.MAX_VALUE` and disables writing logs. The other values enable writing logs at their specific level. So if you want to add your own code to the project and need to silence the Spring Framework, just change the log setting in the second line to `OFF` or `INFO`.

The easiest way to create an application context is to directly initialize it.

```
package com.apress.cems.simple;

import com.apress.cems.config.RepositoryConfig;
import com.apress.cems.config.TestDataSourceConfig;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

public class OneSimpleConfigTest {
    private Logger logger = LoggerFactory.getLogger(OneSimpleConfigTest.
    class);

    @Test
    void testSimpleConfiguration() {
        ApplicationContext ctx =
                new AnnotationConfigApplicationContext(SimpleConfig.class);

        for (String beanName : ctx.getBeanDefinitionNames()) {
            logger.info("Bean " + beanName);
        }
    }
}
```

The `org.springframework.context.ApplicationContext` is the interface implemented by classes that provide the configuration for an application. This interface is an extension of the interface `org.springframework.beans.factory.BeanFactory`, which is the root interface for accessing a Spring Bean container. Implementations of `ApplicationContext` manage a number of bean definitions uniquely identified by their name. Multiple Spring application context implementations exist, each one specific to development needs. For example, if the configuration for an application is provided using an XML file the implementation to use to read the bean declarations is `XmlWebApplicationContext` or any of its extensions. You will be introduced to the most important of them in this book. An `ApplicationContext` implementation provides the following.

- Access to beans using bean factory methods
- The ability to load file resources using relative or absolute paths or URLs
- The ability to publish events to registered listeners
- The ability to resolve messages and support internationalization (most used in international web applications)

An application context is created by the Spring IoC container and initialized with a configuration provided by a resource that can be an XML file (or more) or a configuration class (or more) or both. When the resource is provided as a `String` value, the Spring container tries to load the resource based on the prefix of that string value. When instantiating an application context, different classes are used, based on the prefix (see Table 2-1**)**.

Table 2-1. *Prefixes and Corresponding Paths*

Prefix	Location	Comment
no prefix	In root directory where the class creating the context is executed	In the `main` or `test` directory. The resource being loaded has a type that depends on the ApplicationContext instance being used. (A detailed example is presented after the table.)
`classpath:`	The resource should be obtained from the class-path	In the `resources` directory and the resource will be of type ClassPathResource. If the resource is used to create an application context, the `ClassPathXmlApplicationContext` class is suitable.
`file:`	In the absolute location following the prefix	Resource is loaded as a URL, from the filesystem and the resource will be of type UrlResource. If the resource is used to create an application context, the `FileSystemXmlApplicationContext` class is suitable.
`http:`	In the web location following the prefix	Resource is loaded as a URL and the resource will be of type UrlResource. If the resource is used to create an application context, the `WebApplicationContext` class is suitable.

To provide the functionality for loading resources, an application context must implement the `org.springframework.core.io.ResourceLoader` interface. Here is an example of resource loading without using a prefix.

```
Resource template = ctx.getResource("application-config.xml");
```

Depending on the context class used, the resource loaded can have one of the following types.

- If `ctx` is a `ClassPathXmlApplicationContext` instance resource type will be `ClassPathResource`
- If `ctx` is a `FileSystemXmlApplicationContext` instance resource type will be `FileSystemResource`
- If `ctx` is a `WebApplicationContext` instance resource type will be `ServletContextResource`

And this is where prefixes come in. If we want to force the resource type, no matter what context type is used, the resource must be specified using the desired prefix.

An application context can be created based on a configuration class using a corresponding type of implementation of the `ApplicationContext` interface: the `org.springframework.context.annotation.AnnotationConfigApplicationContext` class, and exactly this was done in the `OneSimpleConfigTest` class. The method in that class is not really a test method, we are using it more like a practical way to start up an application. When the `testSimpleConfiguration` test method is run, a Spring application is started, meaning an application context is created that contains beans declared by the `SimpleConfig` class. The class is empty in this case, so in the log, you only see a minimal list of infrastructure beans that the Spring IoC container needs to build a Spring application.

```
INFO  c.a.c.s.OneSimpleConfigTest - Bean
  org.springframework.context.annotation.internalConfigurationAnnotationProcessor
INFO  c.a.c.s.OneSimpleConfigTest - Bean
  org.springframework.context.annotation.internalAutowiredAnnotationProcessor
INFO  c.a.c.s.OneSimpleConfigTest - Bean
  org.springframework.context.annotation.internalCommonAnnotationProcessor
INFO  c.a.c.s.OneSimpleConfigTest - Bean
  org.springframework.context.event.internalEventListenerProcessor
INFO  c.a.c.s.OneSimpleConfigTest - Bean
  org.springframework.context.event.internalEventListenerFactory
INFO  c.a.c.s.OneSimpleConfigTest - Bean simpleConfig
```

The `@Configuration` annotation is a specialization of the `@Component` annotation, which is the core annotation for creating beans. This means that the configuration class itself is a declaration for a bean, which is why in the previous log, a bean of type `SimpleConfig` is listed and the name of that bean is the same with the class name with the first letter in lower case.

The configuration classes can have beans declarations inside them. A bean declaration inside a configuration class is any method annotated with `@Bean` that returns an instance of a class.

```
package com.apress.cems.simple;

import com.apress.cems.beans.ci.SimpleBean;
import com.apress.cems.beans.ci.SimpleBeanImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OneBeanConfig {

    @Bean
    SimpleBean simpleBean(){
        return new SimpleBeanImpl();
    }
}
```

The following is the `SimpleBean` and `SimpleBeanImpl` code.

```
// SimpleBean.java
package com.apress.cems.beans.ci;

public interface SimpleBean {
}

// SimpleBeanImpl.java
package com.apress.cems.beans.ci;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
```

```
@Component
public class SimpleBeanImpl implements SimpleBean {

    private Logger logger = LoggerFactory.getLogger(SimpleBeanImpl.class);

    public SimpleBeanImpl() {
        logger.info("[SimpleBeanImpl instantiation]");
    }

    @Override
    public String toString() {
        return "SimpleBeanImpl{ code: " + hashCode() + "}";
    }
}
```

Although inside the `simpleBean()` method a constructor is called, annotating it with `@Bean` ensures that once an instance of type `SimpleBeanImpl` has been created, calling the `simpleBean` method will always return the same instance. This is because every bean declared in a Spring application is a singleton by default, unless explicitly configured otherwise. This method is being called by the Spring IoC container when the application context is initially created. This is quite easy to test; we just modify the preceding test class, and we use the context instance to access the declared bean.

```
package com.apress.cems.simple;

import com.apress.cems.beans.ci.SimpleBean;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;
```

```
public class OneSimpleConfigTest {

    @Test
    void testOneBeanConfiguration() {
        ApplicationContext ctx =
                new AnnotationConfigApplicationContext(OneBeanConfig.class);

        SimpleBean simpleBeanOne = ctx.getBean(SimpleBean.class);
        SimpleBean simpleBeanTwo = ctx.getBean(SimpleBean.class);
        Assertions.assertEquals(simpleBeanTwo, simpleBeanOne);
    }
}
```

In the previous code snippet, the `getBean(..)` method is called on the context twice and we assume that the two beans that were returned are one and the same.

Beans that depend on `simpleBean` can be declared in two ways: using the `@Autowired` annotation (covered later in the chapter) and using the `@Bean` annotation. The one that is interesting here is when a bean is declared with the `@Bean` annotation. Let's declare a new bean type named `DependentBeanImpl` and its interface named `DependentBean`.

```
 // DependentBean.java
 package com.apress.cems.beans.db;

public interface DependentBean {
}

// DependentBeanImpl.java
package com.apress.cems.beans.db;

import com.apress.cems.beans.ci.SimpleBean;

public class DependentBeanImpl implements DependentBean {
    private SimpleBean simpleBean;

    public DependentBeanImpl(SimpleBean simpleBean) {
        this.simpleBean = simpleBean;
    }
}
```

To declare a bean of this type in a configuration class, a method named `dependentBean()` is declared.

```
package com.apress.cems.beans.db;

import com.apress.cems.beans.ci.SimpleBean;
import com.apress.cems.beans.ci.SimpleBeanImpl;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SimpleDependentCfg {
     private Logger logger = LoggerFactory.getLogger(SimpleDependentCfg.
     class);

    @Bean
    SimpleBean simpleBean(){
        logger.info("---> Creating 'simpleBean' ");
        return new SimpleBeanImpl();
    }

    @Bean
    DependentBean dependentBean(){
      return new DependentBeanImpl(simpleBean());
    }
}
```

As you can see, it is annotated with `@Bean` and it looks like it is calling the `simpleBean()`. When the application context is created, the Spring IoC intercepts that call. And instead of creating a new instance of `SimpleBeanImpl`, it returns the one already created. This is done using a mechanism called *proxying*, which you will learn more about in Chapter 4. Without getting too deep into how this is done *under the hood*, the easiest way to test that the `simpleBean()` is called only once is to add a statement to write a log message when the method is called. To test that the method is called only once, a test method can be written, which even retrieves the `simpleDependentCfg()` configuration bean and calls the `simpleBean()` on it.

```
package com.apress.cems.beans.db;

import com.apress.cems.beans.ci.SimpleBean;
import org.junit.jupiter.api.Test;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

import static org.junit.jupiter.api.Assertions.*;

public class SimpleDependentCfgTest {

    @Test
    void testSimpleBeans() {
        ConfigurableApplicationContext ctx =
             new AnnotationConfigApplicationContext(SimpleDependentCfg.
             class);
        ctx.registerShutdownHook();

        SimpleDependentCfg simpleDependentCfg =
            ctx.getBean(SimpleDependentCfg.class);
        assertNotNull(simpleDependentCfg);

        SimpleBean simpleBean = simpleDependentCfg.simpleBean();
        assertNotNull(simpleBean);
    }
}
```

If logging is turned off for the Spring Framework, the only log messages we expect to see in the console are the one printed by the `SimpleBeanImpl` constructor and the one printed by the `simpleBean()` method.

```
14:35:00.785 [main] INFO  c.a.c.b.d.SimpleDependentCfg - ---> Creating
'simpleBean'
14:35:00.788 [main] INFO  c.a.c.b.c.SimpleBeanImpl - [SimpleBeanImpl
instantiation]
```

The beans declarations that are not part of the configuration class are identified using a process named component scanning that is enabled by annotating the configuration class with the `@ComponentScan` annotation. When used with no attributes,

it activates various annotations to be detected in the bean classes in the current package and all subpackages: Spring's `@Required` and `@Autowired`, JSR 250's `@PostConstruct`, `@PreDestroy`, and `@Resource` and all stereotype annotations: `@Component`, `@Service`, `@Repository`, `@Controller`, and `@Configuration`. Using the `@ComponentScan` naked (without arguments) is not recommended, because of performance concerns; in a project there will be packages with classes are not configured as bean declarations, so there is no use scanning them. The `@ComponentScan` annotation can be used with a various list of attributes for filtering and reducing scope of scanning, in the next example, scanning for bean annotations was enabled only for package `com.apress.cems.simple`.

```
package com.apress.cems.beans.ci;
import org.springframework.context.annotation.Configuration;

@ComponentScan(basePackages =  {"com.apress.cems.simple"})
@Configuration
public class SimpleConfig {
}
```

Let's play with configuration classes a little more. The `DataSourceConfig` depicted next is a typical Java configuration class, which contains two bean definitions and some properties that are injected from a properties file using the `@PropertySource` annotation. This annotation provides a convenient and declarative mechanism for adding an `org.springframework.core.env.PropertySource` instance to the Spring environment. Placeholders like these `${..}` present in a class annotated with `@PropertySource` will be resolved against the set of property sources registered against the environment. In the following code snippet, the `@PropertySource` annotation receives as argument the location where the properties can be loaded.

```
package com.apress.cems.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.
PropertySourcesPlaceholderConfigurer;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
```

```
import javax.sql.DataSource;
import java.sql.SQLException;

@Configuration
@PropertySource("classpath:db/test-datasource.properties")
public class TestDataSourceConfig {

    @Value("${db.driverClassName}")
    private String driverClassName;
    @Value("${db.url}")
    private String url;
    @Value("${db.username}")
    private String username;
    @Value("${db.password}")
    private String password;

    @Bean
    public static PropertySourcesPlaceholderConfigurer
        propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Bean
    public DataSource dataSource() throws SQLException {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driverClassName);
        ds.setUrl(url);
        ds.setUsername(username);
        ds.setPassword(password);
        return ds;
    }
}
```

! The `propertySourcesPlaceholderConfigurer` bean declaration is done using a static method. The reason for this is so that the bean declaration is picked up when the context is created, earlier than the configuration class annotated with `@Configuration`, and so the property values are added to the Spring environment and become available for injection in the said configuration class, before this class is initialized.

The `@Bean` annotation tells Spring that the result of the annotated method will be a bean that has to be managed by it. The `@Bean` annotation together with the method are treated as a bean definition, and the method name becomes the bean id, so be careful with the method naming. There are more ways to change names, but this topic is covered later in the chapter.

The `@PropertySource` annotation adds a bean of type `PropertySource` to Spring's environment that will be used to read property values from a property file set as argument. The configuration also requires a bean of type `PropertySourcesPlaceholderConfigurer` to replace the placeholders set as arguments for the `@Value` annotated properties.

Spring applications provide a way to access the environment in which the current application is running using a bean of type `org.springframework.core.env.Environment`. This bean models two key aspects of an application environment: *properties* (which will be covered now) and *profiles* (which is covered later). Properties play an important role in all applications because pair of keys and values enable or disable certain capabilities and customize application behavior, most times without the need to recompile the application. Properties may originate from a variety of sources: property files (which have already been introduced), JVM system properties, system environment properties, JNDI, `Properties` instances, `Map` instances, and so forth. The role of the `Environment` bean is to provide a convenient way to declare property sources and inject property values from them where required. In the previous example, the `Environment` bean is not used directly, but resolves the placeholders in the `@Value` annotations under the hood. The same class can be used by injecting the `Environment` infrastructure bean and using it to read the properties from it.

```
 package com.apress.cems.config;
...
import org.springframework.core.env.Environment;

@Configuration
@PropertySource("classpath:db/test-datasource.properties")
public class EnvTestDataSourceConfig {

    @Autowired
    Environment environment;

    @Bean
    public static PropertySourcesPlaceholderConfigurer
         propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Bean
        public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(environment.getProperty("db.
        driverClassName"));
        ds.setUrl(environment.getProperty("db.url"));
        ds.setUsername(environment.getProperty("db.username"));
        ds.setPassword(environment.getProperty("db.password"));
        return ds;
    }
}
```

We've crossed over to bean declarations, and since we are already here, let's introduce the types of dependency injection Spring supports. Beans can be declared in various ways—depending on the way the dependencies are injected. We'll come back to the configuration later in this chapter.

Injection Types

In the preceding configuration, we defined a bean with the `simpleBeanImp|` ID of type `com.apress.cems.beans.ci.SimpleBeanImpl` that has no dependencies. To define a bean with dependencies, we have to decide how those dependencies are injected. Spring supports three types of dependency injection.

- **constructor injection**: The Spring IoC container injects the dependency by providing it as an argument for the constructor
- **setter injection**: The Spring IoC container injects the dependency as an argument for a setter
- **field injection**: The Spring IoC container injects the dependency directly as a value for the field (via reflection and this requires the `open` directive in the `module-info.java` file)

The central annotation used to declare dependencies in Spring is the `@Autowired` annotation and can be used on fields, constructors, setters, and even methods.[11]

The term `autowire` is the short version for automatic dependency injection. This is possible only in Spring applications using component scanning and stereotype annotations to create beans. The `@Autowire` annotation indicates that Spring should take care of injecting that dependency. This raises an interesting question: how does Spring know what to inject?

Every bean in the application context has a unique identifier. It is the developer's responsibility to name beans accordingly if needed. In XML, this can be done using the `id` or `name` attributes, but when using annotations, the line between the ID and the name is blurred, so in this book, only the name will be referred to as a unique bean identifier. To declare beans, we must annotate classes with stereotype annotations: `@Component` and its specializations: `@Service`, `@Repository`, and `@Controller`.

```
package com.apress.cems.beans.ci;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
```

[11]You will see an example of this in Chapter 8.

```
@Component
public class SimpleBeanImpl implements SimpleBean {

    private Logger logger = LoggerFactory.getLogger(SimpleBean.class);

    public SimpleBeanImpl() {
        logger.info("[SimpleBeanImpl instantiation]");
    }

    @Override
    public String toString() {
        return "SimpleBeanImpl{ code: " + hashCode() + "}";
    }
}
```

CC When stereotype annotations are used without any arguments, the responsibility of naming the beans is passed to the Spring container and this guy likes to keep things simple; when doing component scanning and finding classes with stereotype annotations, it creates the beans and just down-cases the first letter of the class names and sets them as the bean names.

That is why in the previous example, the bean name was `simpleBeanImpl`, because the class name was `SimpleBeanImpl`.

If you want to rename it, all you have to do is give the name to the `@Component` annotation (or any of the stereotype annotations) as argument.

```
package com.apress.cems.beans.ci;

@Component("simple")
public class SimpleBeanImpl implements SimpleBean {
...
}
```

To test that the bean is really named `simpleBeanImpl`, we can write a test class that creates an application context containing the previous bean, and we use the context reference to call its `getBean(..)` method to obtain a reference to the bean. This method has more than one version; the version that we are interested now is the one that receives as arguments the bean name and the assumed bean type. The configuration class is named `SimpleAppCfg`.

```
package com.apress.cems.beans.ci;

import org.junit.jupiter.api.Test;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

import static org.junit.jupiter.api.Assertions.*;

class SimpleAppCfgTest {
   @Test
    void testBeanNames() {
       ConfigurableApplicationContext ctx =
            new AnnotationConfigApplicationContext(SimpleAppCfg.class);

       SimpleBean simpleBean = ctx.getBean("simpleBeanImpl", SimpleBean.
       class);
       assertNotNull(simpleBean);
       assertTrue(simpleBean instanceof SimpleBeanImpl);
    }
}
```

If the previous test executes successfully, it means that a bean named `simpleBeanImpl` was found in the context and has the expected type. If you want to see the name that the container assigns to this bean, all you have to do is to print all the bean names in the context, and look for yours and this can be done by calling `getBeanDefinitionNames()` provided by the application context and iterating the returned results.

```
ApplicationContext ctx =
      new AnnotationConfigApplicationContext(SimpleAppCfg.class);
for (String beanName : ctx.getBeanDefinitionNames()) {
            logger.info("Bean " + beanName + " of type "
                + ctx.getBean(beanName).getClass().getSimpleName());
 }
```

Aside from other infrastructure beans, here is what we can see in the console.

```
DEBUG AnnotationConfigApplicationContext - Refreshing
..AnnotationConfigApplicationContext
...
INFO   SimpleAppCfgTest - [Bean simpleBeanImpl of type SimpleBeanImpl]
DEBUG AnnotationConfigApplicationContext - Closing
..AnnotationConfigApplicationContext
```

But what if we want a different name for the bean? Not a problem, stereotype annotations can receive a name as argument.

```
package com.apress.cems.beans.ci;
...
@Component("simple")
public class SimpleBeanImpl implements SimpleBean {
...
}
```

By annotating the class `SimpleBeanImpl` with `@Component("simple")` the bean being created is now is named `simple`. But what if, for some reason we need to give multiple names to a bean? Aliases cannot be defined using stereotype annotations, but a request has been made to provide such a feature in 2010 and it is still open, so there is still hope.[12]

It is possible to have multiple names for a bean, but one of them will be the unique identifier and all the other names are just aliases. Declaring aliases using stereotype annotations is not supported at the moment, but it is possible using the `@Bean` annotation. The caveat of declaring aliases using the `@Bean` annotation is that the method name will no longer be used as a bean name, only the names declared as values for the `name` attribute in the annotation will be used.

```
package com.apress.cems.beans.naming;
...
@Configuration
public class AliasesCfg {
```

[12]If you are interested in this feature, you can follow the evolution of the issue here: https://github.com/spring-projects/spring-framework/issues/11402.

```
    @Bean(name= {"beanOne", "beanTwo"})
    SimpleBean simpleBean(){
        return new SimpleBeanImpl();
    }
}
```

Let's test the assumptions made so far, shall we?

```
package com.apress.cems.beans.naming;
...
public class AliasesCfgTest {

    private Logger logger = LoggerFactory.getLogger(AliasesCfg.class);

    @Test
    void testSimpleBeans() {
        ConfigurableApplicationContext ctx =
            new AnnotationConfigApplicationContext(AliasesCfg.class);

        SimpleBean simpleBean = ctx.getBean("beanOne", SimpleBean.class);
        assertNotNull(simpleBean);
        assertTrue(simpleBean instanceof SimpleBeanImpl);

        SimpleBean simpleBean2 = ctx.getBean("beanTwo", SimpleBean.class);
        assertEquals(simpleBean2, simpleBean);

        // no bean named 'simpleBean'
        assertThrows(NoSuchBeanDefinitionException.class, () -> {
            ctx.getBean("simpleBean", SimpleBean.class);
        });

        ctx.close();
    }
}
```

The previous test, if it passes, proves that there is a bean of type `SimpleBeanImpl` that can be referred by two names, `beanOne` and `beanTwo`, and cannot be referred by `simpleBean`.

Related to bean naming is the `@Description` annotation, which was added in Spring 4.x. This annotation adds a description to a bean, which is quite useful when beans are exposed for monitoring puposes. It can be used with `@Bean` and

@Component (and its specializations). An example of its usage is depicted in the following code listing.

```
 package com.apress.cems.scopes;
 import org.springframework.context.annotation.Description;
 ...
@Description("Salary for an employee might change,
      so this is a suitable example for a prototype scoped bean")
@Component
public class Salary {
...
}

// or
package com.apress.cems.beans.db;
...
public class SimpleDependentCfg {

    @Description("This bean depends on 'simpleBean'")
    @Bean
    DependentBean dependentBean(){
      return new DependentBeanImpl(simpleBean());
    }
    ...
}
```

! The Spring convention is to use the standard Java convention for instance field names when naming beans. This means that bean names start with a lowercase letter and are camel-cased from then on. The purpose of good bean naming is to make the configuration quick to read and understand. Also, when using more advance features like AOP, it is useful to apply advice to a set of beans related by name. But if you want to define your own naming configurations, just know that anything goes; special characters are accepted as part of a bean name. Whatever you decide, just keep it consistent in the context of an application.

CC Out of the box, Spring will try to **autowire by type**, because rarely in an application more than one bean of a type is needed. Spring inspects the type of dependency necessary and will inject the bean with that exact type.

So let's start slow and cover each type of injection type, before going deeper into how Spring does it under the hood.

Constructor Injection

Constructor injection is a mechanism of providing dependencies through constructor arguments if the bean definition looks like the following.

```
package com.apress.cems.beans.ci;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class ComposedBeanImpl implements ComposedBean {

    private SimpleBean simpleBean;

    @Autowired
    public ComposedBeanImpl(SimpleBean simpleBean) {
        this.simpleBean = simpleBean;
    }
}
```

! Starting with Spring 4.3 using `@Autowired` if the class has a single constructor is no longer necessary. Even if redundant, the annotation was used in the previous example for teaching purposes to show `@Autowired` being used on a constructor.

The `ComposedBeanImpl` class has a field of type `SimpleBean` defined, which is initialized when the constructor is called with the value passed as argument. Because of the `@Autowired` annotation, the Spring IoC container knows that before creating the `composedBeanImpl`, it first has to create a bean of type `SimpleBean` and inject it as an argument. The preceding code and configuration snippets provide all the necessary

information so that the Spring IoC container can create a bean of type `SimpleBean` named `simpleBeanImpl` and then inject it into a bean of type `ComposedBean` named `composedBeanImpl`. The following is done behind the scenes.

```
SimpleBean simpleBean = new SimpleBeanImpl();
ComposedBean composedBean = new ComposedBeanImpl(simpleBean);
```

Spring creates the beans in the order they are needed. The dependencies are first created and then injected into the beans that need them.

CC By default, if Spring cannot decide which bean to autowire based on type (because there are more beans of the same type in the application), it first searches for any type of bean declared with a `@Qualifier` annotation on it. If nothing is found, then it defaults to autowiring by name. The name considered as the criterion for searching the proper dependency is the name of the field being autowired.

In the previous case, if Spring cannot decide what to autowire based on type, it looks for a bean declared with a `@Qualifier` annotation. And if nothing is found, it looks for a bean with the required type, which is named `simpleBean`. If nothing is found, then an exception of type `org.springframework.beans.factory.NoSuchBeanDefinitionException` occurs.

The configuration class used to register the bean declaration listed previously and its dependencies can be seen in the next code listing.

```
package com.apress.cems.beans.ci;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = {"com.apress.cems.beans.ci"} )
public class SimpleAppCfg {
}
```

The class is empty because it exists only to be annotated with the configuration annotations necessary to enable bean discovery and to provide a basis to create an application context. Component scanning enables for classes annotated with stereotype annotations to auto-detected using classpath scanning, and a bean to be created for each one of them within the application context. Creating an application context based on that class is easy, just instantiate an `AnnotationConfigApplicationContext`. The full class to test the bean is depicted in the next code listing.

```
package com.apress.cems.beans.ci;

...
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

import static org.junit.jupiter.api.Assertions.*;

class SimpleAppCfgTest {

    private Logger logger =
        LoggerFactory.getLogger(SimpleAppCfgTest.class);

    @Test
    void testSimpleBeans() {
        ConfigurableApplicationContext ctx =
            new AnnotationConfigApplicationContext(SimpleAppCfg.class);

        ComposedBean composedBean = ctx.getBean(ComposedBean.class);
        assertNotNull(composedBean);
        ctx.close();
    }
```

If we check the log, it will become obvious which type of autowiring was used.

```
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
singleton
   bean 'simpleBeanImpl'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
singleton
   bean 'composedBeanImpl'
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Autowiring by type from bean
name
   'composedBeanImpl' via constructor to bean named 'simpleBeanImpl'
TRACE o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance of
   bean 'composedBeanImpl'
```

If we were to declare another bean of type SimpleBean when starting up the Spring application, the results might become unpredictable. But, how do we create another bean of type SimpleBean? We either create another class that implements that interface and annotate it with @Component, or we chose an easier way and we develop the configuration class a little bit more.

```
package com.apress.cems.beans.ci;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = {"com.apress.cems.beans.ci"} )
public class SimpleAppCfg {
    @Bean
    SimpleBean anotherSimpleBean(){
        return new SimpleBeanImpl();
    }
}
```

In the previous code snippet, a bean is declared explicitly using the @Bean annotation. This annotation is used at the method level to indicate that the method produces a bean managed by the Spring IoC container. Typically, @Bean methods are declared within @Configuration classes and they are detected by the Spring IoC container without the need for classpath scanning. If we run the previous test class with the new configuration class, the console log will display the cause why the application context could not be created.

```
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
singleton
   bean 'anotherSimpleBean'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
singleton
   bean 'simpleBeanImpl'
...
```

```
org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating
    bean with name 'composedBeanImpl' defined in file [..ComposedBeanImpl.
    class]:
    Unsatisfied dependency expressed through constructor parameter 0;
    nested exception 0
    is org.springframework.beans.factory.the
    NoUniqueBeanDefinitionException:
    No qualifying bean of type 'com.apress.cems.beans.ci.SimpleBean'
    available:
    expected single matching bean but found 2: simpleBeanImpl,another
    SimpleBean
```

In this case, we have two beans with the same `SimpleBean` type; neither is annotated with `@Qualifier` to make it a candidate for injection. And because neither of them is named `simpleBean`, Spring does not know which dependency to inject, so it unable to create a proper application context. This is a conflict that cannot be resolved without additional configuration changes. In this case, we have quite a few choices.

- Specify a name for the bean to be created by annotating `SimpleBeanImpl` and, in our case, to remove any ambiguity, we annotate the class with `@Component("simpleBean")`.
- Specify a name for the bean created in the configuration class; in our case, we annotate the method to `@Bean("simpleBean")`.
- Specify a name for the bean created in the configuration class by modifying the method name from `anotherSimpleBean` to `simpleBean`.
- Specify the name of the bean to inject in the `ComposedBeanImpl` class using the `@Qualifer` annotation on the constructor parameter.

```
 package com.apress.cems.beans.ci;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
...
```

```
@Component
public class ComposedBeanImpl implements ComposedBean {

    private SimpleBean simpleBean;

    @Autowired
    public ComposedBeanImpl(@Qualifier("anotherSimpleBean")) {
        this.simpleBean = simpleBean;
    }
...
}
```

The `@Autowire` annotation can be used on constructors to tell Spring to use autowiring to provide arguments for that constructor. The way Spring will identify the autowiring candidate is by using the same logic presented before: it will try to find a unique bean of the parameter type. If it finds more than one, the one annotated with `@Qualifier` will be considered, if there is no such bean, the one named as the parameter will be injected. In using `@Autowired` on constructors, it makes no sense to have more than one constructor annotated with it, and Spring will complain about it because it will not know what constructor to use to instantiate the bean.

Everything that was mentioned about the capabilities of `@Autowired` and `@Qualifier` and how to use them to control bean autowiring applies to setter injection. It applies to instance variables too, even if field injection is not recommended, because of performance costs caused by the use of reflection.

Everything that was mentioned about the capabilities of `@Autowired` and `@Qualifier` and how to use them to control bean autowiring applies to the JSR-330 `@Inject` and `@Named` as well.

All of this is fine, but what if my dependency is a `String` or some other simple type of object, or a primitive that should be read from a `properties` file? For that, the `@Value` annotation was introduced. So, let's make our `ComposedBean` more complicated by adding two more fields one of type `String` and one of type `boolean` and add them to the constructor to see the power of Spring IoC container. And since reading from properties files requires extra infrastructure beans to be created, we'll keep it simple for now and just declare the values to be injected on the spot.

```
package com.apress.cems.beans.ci;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class ComposedBeanImpl implements ComposedBean {

    private SimpleBean simpleBean;
    private String code;
    private boolean complicated;

    @Autowired
    public ComposedBeanImpl(SimpleBean simpleBean,
          @Value("AB123") String code, @Value("true") boolean complicated) {
        this.simpleBean = simpleBean;
        this.code = code;
        this.complicated = complicated;
    }

    public SimpleBean getSimpleBean() {
        return simpleBean;
    }

    public String getCode() {
        return code;
    }

    public boolean isComplicated() {
        return complicated;
    }
}
```

The `@Value` annotation is used when the value to inject is a scalar.[13] Text values, numbers, and booleans can be used as arguments for the constructor using the `@Value` attribute. So what happens here is equivalent to the following.

```
ComposedBean composedBean = new ComposedBeanImpl(simpleBean, "AB123", true);
```

Developers choose to use constructor injection when it is mandatory for the dependencies to be provided, since the bean depending on them cannot be used without them. Constructor injection enforces this restriction since the dependent bean cannot even be created if the dependencies are not provided.

! The `@Autowired` annotation provides an attribute named `required` which, when it is set to `false` declares the annotated dependency as not being required. When `@Autowired` is used on a constructor, this argument can never be used with a value of `false`, because it breaks the configuration. It's logical if you think about it, you cannot make optional the only means you have to create a bean, right? We'll come back to this topic when we talk about setter injection.

If the Spring IoC container cannot find a bean to inject into when creating a bean declared to use constructor injection, an exception of type `org.springframework.beans.factory.UnsatisfiedDependencyException` will be thrown and the application will fail to start. You can cause such an issue yourself in the `chapter02/beans` project by commenting the `@Component` annotation in class `SimpleBeanImpl` and then running the `SimpleAppCfgTest` test class.

Dependency injection also is suitable when a bean needs to be immutable by assigning the dependencies to final fields. The most common reason to use constructor injection is that sometimes third-party dependencies are used in a project, and their classes were designed to support only this type of dependency injection. In creating a bean, there are two steps that need to be executed one after the other. The bean first needs to be **instantiated**, and then the bean must be **initialized**. The constructor

[13]The term *"scalar"* comes from linear algebra, where it differentiates a number from a vector or matrix. In computing, the term has a similar meaning. It distinguishes a single value such as an integer or float from a data structure like an array. In Spring, *scalar* refers to any value that is not a bean and cannot be treated as such.

injection combines two steps into one, because injecting a dependency using a constructor means basically instantiating and initializing the object at the same time.

Enough reading; it's time to write some code. In the `chapter02/beans-practice` project, the `Item` interface and the `Book` class implementing it have been declared. This class has a single field of type `String` that has to be injected with a value of your choosing when the bean is created. There is also a `Human` interface and a `Person` class implementing it, which has a single field of type `Item`. Your assignment is to declare define two beans, one of type `Person`, one of type `Book`, make sure the `book` bean is injected into the `person` bean.

To test your implementation run the class `com.apress.cems.beans.HumanAppCfgTest` class that contains a few test statements making sure everything was configured correctly. If you get stuck, you can take a look at the proposed beans configurations in the `chapter02/beans` project.

This section has introduced the simplest way to create beans using constructor injection. There are other things that can be done when creating a bean—all configurable, but since they are more related to customization than creation, they will be covered in future sections.

The equivalent XML configuration for the previously declared bean is depicted in the following code snippet.

```
 <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="simpleBeanImpl"
         class="com.apress.cems.beans.ci.SimpleBeanImpl" />

    <bean name="composedBeanImpl"
        class="com.apress.cems.beans.ci.ComposedBeanImpl">
        <constructor-arg index="0" ref="simpleBeanImpl"/>
        <constructor-arg index="1" value="AB123" />
        <constructor-arg index="2" value="true" />
    </bean>

</beans>
```

An important thing to point out about XML configuration: the `ref` attribute is used to specify bean dependencies, and the `value` attribute is used to specify scalar dependencies.

In XML, arguments can be specified using indexes defined by the order of the parameters in the constructor by setting the `index` attribute to specify the argument where each value is injected. If the next code snippet, you can see the declaration of the `ComposedBeanImpl(..)` constructor.

```
package com.apress.cems.beans.ci;

@Component
public class ComposedBeanImpl implements ComposedBean {

    private SimpleBean simpleBean;
    private String code;
    private Boolean complicated;

    @Autowired
    public ComposedBeanImpl(
                /* index = "0" */  SimpleBean simpleBean,
                /* index = "1" */  String code,
                /* index = "2" */  Boolean complicated
        ) {
        this.simpleBean = simpleBean;
        this.code = code;
        this.complicated = complicated;
    }
}
```

In the XML, the names of the beans are explicitly configured, so the configuration will perfectly match the one created using annotations. The XML configuration can be simplified using the **c-namespace**, a special namespace, without a schema location that exists with the sole purpose of simplifying XML configuration. A sample of the previous configuration using this namespace is depicted in the following configuration listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
```

```
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="simpleBeanImpl"
        class="com.apress.cems.beans.ci.SimpleBeanImpl" />

    <bean name="composedBeanImpl"
        class="com.apress.cems.beans.ci.ComposedBeanImpl"
        c:_0-ref="simpleBeanImpl" c:_1="AB123" c:_2="true"/>

</beans>
```

XML arguments can also be specified using the names of the parameters in the constructor. The following snippet shows how to created beans using constructor argument names with the **c-namespace** and without.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    <bean name="simpleBeanImpl"
        class="com.apress.cems.beans.ci.SimpleBeanImpl" />

     <bean name="composedBeanImpl"
          class="com.apress.cems.beans.ci.ComposedBeanImpl">
        <constructor-arg name="simpleBean" ref="simpleBeanImpl"/>
        <constructor-arg name="code" value="AB123" />
        <constructor-arg name="complicated" value="true" />
    </bean>
```

And if the preceding declaration looks a little too verbose, there is always the **c-namespace** syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

 <bean name="composedBeanImpl"
     class="com.apress.cems.beans.ci.ComposedBeanImpl"
     c:simpleBean-ref="simpleBeanImpl" c:code="AB123"
c:complicated="true"/>

</beans>
```

Although XML is not required for the certification exam, samples are presented in the book for comparison, so that you have a complete understanding of how Spring applications are configured and to cover things that require a typical Java configuration.

Setter Injection

To use the setter injection, the class type of the bean must have setter methods used to set the dependencies. A constructor is not mandatory. If no constructor is declared, Spring will use the default no argument constructor, which every class automatically inherits from the Java `Object` class to instantiate the object and the setter methods to inject dependencies. If a no argument constructor is explicitly defined, Spring will use it to instantiate the bean. If a constructor with parameters is defined, the dependencies declared in this way will be injected using constructor injection, and the ones defined using setters will be injected via setter injection.

In conclusion, when creating a bean using setter injection, the bean is first instantiated by calling the constructor. If there are any dependencies declared as arguments for the constructor these will be obviously initialized first. When the constructor does not require arguments, the bean is then initialized by injecting the dependencies using setters, so in this case, instantiation and initialization are two different steps.[14] So if your bean definition looks like the following.

```
package com.apress.cems.beans.si;

import com.apress.cems.beans.ci.SimpleBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class AnotherComposedBeanImpl implements AnotherComposedBean {

    private SimpleBean simpleBean;

    @Autowired
    public void setSimpleBean(SimpleBean simpleBean) {
        this.simpleBean = simpleBean;
    }
```

[14]Later in this chapter, you will learn that initialization can be split into two steps as well: setting dependencies and calling a special initialization method that makes use of those dependencies.

```
    public SimpleBean getSimpleBean() {
        return simpleBean;
    }
}
```

The preceding code and configuration snippet provide all the necessary information so that the Spring container can create a bean of type `AnotherComposedBeanImpl` and then inject into it a bean of `SimpleBean` type. The `SimpleBean` interface that was introduced in the previous section is used as a type for the dependency. This allows a bean of any type extending this interface to be injected as a dependency. The dependency is injected after the bean is instantiated by calling the `setSimpleBean` setter method and providing the bean of `SimpleBeanImpl` type as argument. Assuming that there is a class named `AnotherSimpleBeanImpl` that implements the `SimpleBean` interface, the following is done behind the scenes.

```
SimpleBean simpleBean = new AnotherSimpleBeanImpl();
AnotherComposedBeanImpl complexBean = new AnotherComposedBeanImpl();
complexBean.setSimpleBean(simpleBean);
```

What is obvious for setter injection in Spring is that the `@Autowired` annotation must be present on setter methods to tell the Spring IoC container where the dependency must be injected. If a dependency is not mandatory you can always annotate the setter with `@Autowired(required = false)`, but in this case, you have to carefully design your code so that `NullPointerExceptions` will be avoided. The next code sample contains the declaration of a class named `BadBeanImpl` that can be used to create a bean with an optional dependency of `MissingBean` type.

```
 package com.apress.cems.beans.aw;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class BadBeanImpl implements BadBean {
    private MissingBean missingBean;
    private BeanTwo beanTwo;
```

```
    public MissingBean getMissingBean() {
        return missingBean;
    }

    @Autowired(required = false)
    public void setMissingBean(MissingBean missingBean) {
        this.missingBean = missingBean;
    }

    public BeanTwo getBeanTwo() {
        return beanTwo;
    }

    @Autowired
    public void setBeanTwo(BeanTwo beanTwo) {
        this.beanTwo = beanTwo;
    }
}
```

Interfaces and implementations for `MissingBean` and `BeanTwo` are empty and irrelevant for this example. Assuming we have a configuration class named `NotRequiredBeanCfg` that is used to create an application context containing the bean declared previously, testing that the `missingBean` dependency is not provided is quite easy.

```
package com.apress.cems.beans.cw;
...
public class NotRequiredBeanCfgTest {

    @Test
    void testAutowire(){
        ConfigurableApplicationContext ctx =
             new AnnotationConfigApplicationContext(NotRequiredBeanCfg.class);
        assertNotNull(ctx);

        BadBean badBean = ctx.getBean(BadBean.class);
        assertNotNull(badBean.getBeanTwo());
        assertNull(badBean.getMissingBean());
    }
}
```

By declaring the dependency optional, if there is no bean of `MissingBean` type found in the application context, the bean `BadBean` type can be created without it, a null value will be used instead and no exception of `NoSuchBeanDefinitionException` type will be thrown. But beware of using this, and be careful with your design, because it opens up the possibility for errors.

And this is all that can be said about the setter injection for now. Choosing between setter and constructor injection depends only on the needs of the application; the code that is already written and cannot be changed (legacy code) and third-party libraries. In Spring, constructor injection is preferred as it allows beans to be immutable and dependencies not `null` which will ensure that beans will always be completely initialized and fully functional. When deciding on which type of injection to configure, just make the decisions based on the existing code and best coding practices (e.g., do not define a constructors with more than six arguments, because this is considered a bad practice named *code smell*).

There are three main reasons for using setter injection.

- It allows reconfiguration of the dependent bean, as the setter can be called explicitly later in the code, and a new bean can be provided as a dependency (this obviously means that a bean created using setter injection is not immutable)
- Preferably, it is used for bean dependencies that can be set with default values inside the bean class
- Third-party code only supports setter injections.

A constructor and setter injection can create the same bean. We can modify the `AnotherComposedBeanImpl` class to add a constructor too. Since we just introduced optional dependencies, let's make use of them.

```
package com.apress.cems.beans.si;

import com.apress.cems.beans.ci.SimpleBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
```

```
@Component
public class AnotherComposedBeanImpl implements AnotherComposedBean {

    private SimpleBean simpleBean;

    private boolean complex;

    @Autowired
    public AnotherComposedBeanImpl(@Value("true") boolean complex) {
        this.complex = complex;
    }

    @Autowired(required = false)
    public void setSimpleBean(SimpleBean simpleBean) {
        this.simpleBean = simpleBean;
    }

    public SimpleBean getSimpleBean() {
        return simpleBean;
    }

    public boolean isComplex() {
        return complex;
    }
}
```

The previous bean can be configured using an XML file with the following contents.

```
 <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="anotherSimpleBeanImpl"
        class="com.apress.cems.beans.si.AnotherSimpleBeanImpl" />
```

```
    <bean name="anotherComposedBeanImpl"
        class="com.apress.cems.beans.si.AnotherComposedBeanImpl"
        c:complex="true">
        <property name="simpleBean" ref="anotherSimpleBeanImpl" />
    </bean>
```

The following code snippet depicts the simplified version of the same configuration using the **p-namespace**, which simplifies the XML even more. In a similar way, the **c-namespace** simplifies bean configuration when using constructors.

```
    <?xml version="1.0" encoding="UTF-8"?>
<beans ../>
    <!-- using the p-namespace -->
    <bean name="anotherComposedBeanImpl"
          class="com.apress.cems.beans.si.AnotherComposedBeanImpl"
          c:complex="true"
          p:simpleBean-ref="anotherSimpleBeanImpl"/>

   </beans>
```

When XML configuration is used bean declarations using setter dependency injection allows dependencies to be optional. In the `com.apress.cems.xml.ApplicationContextTest` from the `chapter02/xml` project, there is a test case named method `testJdbcRepo()` using a configuration file named `application-opt-prod.xml` that contains a bean declaration with a dependency that could be injected using a setter, but the configuration to do so was commented out.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <!-- Missing dependency -->
    <bean id="detectiveRepo"
       class="com.apress.cems.xml.repos.impl.JdbcDetectiveRepo">
        <!-- <property name="dataSource" ref="dataSource"/> -->
    </bean>
</beans>
```

If you run that method, you will see that the test passes and that the method contains this line: `assertThrows(NullPointerException.class, () -> detectiveRepo.findById(1L));`. This calls a repository method making use of the `dataSource` dependency. It does not expect the dependency to be there, and a `NullPointerException` is thrown.

All other examples can be found in the `chapter02/beans` project. The package containing all beans and configurations for trying setter injection is named `com.apress.cems.beans.si`.

Field Injection

Another type of injection supported in Spring is *field-based injection*. In this case, the `@Autowired` annotation is placed directly on a class field and the Spring IoC Container is in charge of injecting a bean as value to the field when the application is started. So, another way of writing a composed bean that makes use of field injection is depicted in the following code snippet.

```
package com.apress.cems.beans.fi;

import com.apress.cems.beans.ci.ComposedBean;
import com.apress.cems.beans.ci.SimpleBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class BadComposedBean implements ComposedBean {

    @Autowired
    private SimpleBean simpleBean;

    private String code;
    private boolean complicated;

    public BadComposedBean(@Value("AB123") String code,
             @Value("true") boolean complicated) {
        this.code = code;
        this.complicated = complicated;
    }
```

```
    @Override
    public SimpleBean getSimpleBean() {
        return simpleBean;
    }

    @Override
    public String getCode() {
        return code;
    }

    @Override
    public boolean isComplicated() {
        return complicated();
    }
}
```

The `com.apress.cems.beans.ci.ComposedBean` interface was implemented and a constructor that initializes the fields that are not injected with beans was added. The configuration class for this example is named `FiAppCfg` and is an empty configuration class annotated with `@Configuration` and `@ComponentScan`. The test class that test the proper creation of this bean is called `FiAppCfgTest` and contains the following code.

```
package com.apress.cems.beans.fi;

import com.apress.cems.beans.ci.ComposedBean;

import org.junit.jupiter.api.Test;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

import static org.junit.jupiter.api.Assertions.*;

public class FiAppCfgTest {
    @Test
    void testSimpleBeans() {
        ConfigurableApplicationContext ctx =
             new AnnotationConfigApplicationContext(FiAppCfg.class);
```

```
        ComposedBean composedBean = ctx.getBean(ComposedBean.class);
        assertNotNull(composedBean);
        assertNotNull(composedBean.getSimpleBean());
        assertEquals("AB123", composedBean.getCode());
        assertTrue(composedBean.isComplicated());
        ctx.close();
    }
}
```

The code that is underlined is the line that verifies that a bean of type `SimpleBean` was injected into the `simpleBean` field.

! Although it seems practical to use field injection, because the need for writing setters disappears, and the classes become more readable, using field injection hides the dependencies of a class. As a developer you would not know the dependencies of a class without looking at the source code. This practice had led to people referring to Spring as "magic," and dependencies being wired "automagically" in Spring applications. So the fields defining the state of an object should be publicly available to avoid objects being created with an inconsistent state and the risk of `NullPointerExceptions` being thrown.

Another reason why I personally do not like field injection is because it makes my classes more difficult to test (especially when writing unit tests). A class designed for field-injection is harder to use outside of a Spring context. So if I want to test a small functionality in my class, and the injected field is declared private and there is no setter for it, I would not be able to inject a stub or a mock implementation. Sure, there is always reflection, and testing libraries like Mockito provide utility methods to avoid the typical reflection boilerplate, but that should not even be considered for many reasons (performance and security), but also because starting with Java 9, it become trickier as hell.

Also, if the previous arguments did not convince you, constructor and setter injection is done using autowiring, which involves proxying. That is why these two types of dependency injection types are recommended. Field injection is done using reflection, because this is the only way to access a private field in Java, which can lead to a slow performance so, avoid using it unless you really have no other choice.

I recommend using field injection only in the following contexts.

- `@Configuration` classes: Bean A is declared in configuration class A1, and bean B declared in configuration class B1, depends on bean A.
- `@Configuration` classes: To inject infrastructure beans that are created by the Spring IoC container and need to be customized. (Be careful when doing this because it ties your implementation to Spring.)
- Test classes: The tested bean should be injected using field injection as it keeps things readable.

Bean Scopes

The word *singleton* was used a little bit so far in the book in relation to beans, and you've definitely noticed it in the logs. I never really explained why, because there is a time for everything. A *bean scope* is a term that describes how long a bean's lifespan is. Spring refers to the beans as singletons, because that is the default scope.[15] A singleton bean is created when the application is bootstrapped, and is managed by the Spring IoC container until the application is shutdown or the context is closed. If you've executed some of the sources, you've probably noticed an important piece of the log in the console.

```
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Destroying singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory@10e92f8f:
defining beans simpleBean1,simpleBean2,complexBean,
o.s.c.a.internalConfigurationAnnotationProcessor...
```

When the Spring IoC instantiates beans, it creates a single instance for each bean, which is destroyed when the application context is shut down. The scope of a bean can be changed by using a special Spring annotation. This annotation is `@Scope`, and the default scope for a bean is `singleton`. The scopes are defined in Table 2-2.

[15]The Singleton design pattern is therefore used heavily in Spring.

Table 2-2. *Bean Scopes*

Scope	Annotation	Description
singleton	none, @Scope("singleton"), @Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)	The Spring IoC creates a single instance of this bean, and any request for beans with a name (or aliases) matching this bean definition results in this instance being returned.
prototype	@Scope("prototype"), @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)	Every time a request is made for this specific bean, the Spring IoC creates a new instance.
request	@Scope("request"), @RequestScope, @Scope(WebApplicationContext.SCOPE_REQUEST)	The Spring IoC creates a bean instance for each HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
session	@Scope("session"), @SessionScope, @Scope(WebApplicationContext.SCOPE_SESSION)	The Spring IoC creates a bean instance for each HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
application	@Scope("application"), @ApplicationScope, @Scope(WebApplicationContext.SCOPE_APPLICATION)	The Spring IoC creates a bean instance for the global application context. Only valid in the context of a web-aware Spring ApplicationContext.
websocket	@Scope("websocket")	The Spring IoC creates a bean instance for the scope of a `WebSocket`. Only valid in the context of a web-aware Spring ApplicationContext.
thread	@Scope("thread")	Introduced in Spring 3.0, it is available, but not registered by default, so the developer must explicitly register it in the same way as if a custom scope would be defined.

The SCOPE_REQUEST, SCOPE_SESSION and SCOPE_APPLICATION scope variables are declared in the org.springframework.web.context.WebApplicationContext interface, because they are specific to web applications.

So when a bean is created in the simplest way just by annotating a class with a stereotype annotation (@Component or any of its specializations), like we've done until now.

```
@Component
public class DepBean {
...
}
```

The default scope is singleton. The scope of a bean can be changed by annotating the class with @Scope.

```
 import org.springframework.context.annotation.Scope;
 ...
@Scope("prototype")
@Component
public class DepBean {
}
```

The preceding declaration is the same as the following one.

```
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
 ...
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Component
public class DepBean {
}
```

There are constants matching the scope types, which are listed in Table 2-3. They are declared in various interfaces based on their domain. For example, the ones for basic configuration like *singleton* and prototype are declared in the org.springframework.beans.factory.config.ConfigurableBeanFactory:

```
package org.springframework.beans.factory.config;
 ...
public interface ConfigurableBeanFactory extends
      HierarchicalBeanFactory, SingletonBeanRegistry {
        String SCOPE_SINGLETON = "singleton";
        String SCOPE_PROTOTYPE = "prototype";
        ....
}
```

Now that we know that more beans scopes are available, how do we solve dependencies between beans with different scopes? When we have a `prototype` bean depending on a `singleton`, there is no problem. Every time the prototype bean is requested from the context, a new instance is created, and the singleton bean is injected into it. But the other way around, things get a little complicated.

The domain that is most sensitive when it comes to dependencies among beans with different scopes is the web applications domain. There are three main bean scopes designed to be used in web applications: `request`, `session`, and `application`. Let's assume that we have a service bean called `ThemeManager` that manages updates on an object of type `UserSettings` containing the settings that a `User` has for an interface in a web application. This means that the `ThemeManager` bean has to work with a different `UserSettings` bean for each HTTP session. Obviously, this means that the `UserSettings` bean should have the scope session.

```
import org.springframework.web.context.WebApplicationContext;
import org.springframework.context.annotation.Scope;
...

@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION)
public class UserSettings {...}

\\ ThemeManager.java contents
@Component
public class ThemeManager {
    private UserSettings userSettings;
```

```
    @Autowired
    public void setUserSettings(UserSettings userSettings) {
        this.userSettings = userSettings;
    }
}
```

But how can the problem with single instantiation be solved? On different HTTP sessions, methods like `themeManager.saveSettings(userSettings)` should be called with the `userSettings` bean specific to that session. But the preceding configuration does not allow for this to happen. The preceding configuration just sets the scope for the bean, but does nothing about it. The `ThemeManager` bean is created, it requires an instance of type `UserSettings` as dependency, the dependency is injected, and that's it. Since the setter method is not called explicitly with a different instance, how can Spring refresh that dependency? Well, something extra has to be added to the `@Scope` annotation.

```
import org.springframework.web.context.WebApplicationContext;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
...
@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION,
        proxyMode = ScopedProxyMode.INTERFACES)
public class UserSettings implements BasicUserSettings {...}

\\ ThemeManager.java contents
@Component
public class ThemeManager {
    private UserSettings userSettings;

    @Autowired
    public void setUserSettings(UserSettings userSettings) {
        this.userSettings = userSettings;
    }
}
```

The `ScopedProxyMode` enum is part `spring-context` module and contains a list of values for scoped-proxy options. To customize a bean based on its annotations Spring makes use of the Proxy pattern[16]. A proxy is an implementation that wraps around the target implementation and provides extra behavior in a transparent manner. The proxy implements the same interface(s) the target implementation does, so the same API is available. When the target type is not an interface, but a class, the proxy will be a class extending the target class. This was supported in previous versions of Spring by adding a library like CGLIB on the classpath. This is no longer needed in Spring 5 because CGLIB was repackaged and is now part of Spring AOP.[17] By annotating the `UserSettings` class with `@Scope`, we can tell the Spring IoC container how we want our proxy to behave. The `WebApplicationContext.SCOPE_SESSION` means that the proxy should reinstantiate the target every time a new HTTP Session is created. The `ScopedProxyMode.INTERFACES` means that the proxy that will be wrapped around our target (of type `UserSettings`) will implement the `BasicUserSettings` interface. The `TeamManager` bean will not be injected with a simple `UserSettings` bean, but with a proxy that is wrapped around a bean of type `UserSettings` to provide the behavior of refreshing its state based on the HTTP Session.

If the proxy type used is `ScopedProxyMode.TARGET_CLASS`, then this scope configuration annotation.

```
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
```

It can be replaced with the `@SessionScope` annotation from package `org.springframework.web.context.annotation`, which is a specialization of the `@Scope` annotation that fixes the proxy type to class-based proxies. The Spring Web MVC module provides specializations for most common used scopes in web applications.

The AOP framework was introduced here because it was important to show how beans with different scopes can be used correctly. The AOP framework complements the Spring IoC container. The Spring container can be used without AOP in small applications (teaching applications mostly) that do not require the use of security or

[16]More info here https://en.wikipedia.org/wiki/Proxy_pattern

[17]AOP is an acronym for aspect-oriented programming, which is a programming paradigm aiming to increase modularity by allowing the separation of cross-cutting concerns. This is done by defining something called "pointcut," which represents a point in the code where new behavior will be injected. You can read more about it in Chapter 4.

transactions, because these are the key crosscutting concerns for enterprise applications. But keep in mind that even if you do not use any AOP components in your application, Spring uses AOP a lot under the hood. The Spring AOP framework has the entire fourth chapter of this book dedicated to it.

And since the previous code sample is web specific and difficult to play with, let's build an example that makes use of a prototype bean that can be tested easily. Let's consider an employee that constantly lies when asked about his salary. So each time `employee.getSalary()` is called a different salary is returned. The implementation consists of a `Salary` class that contains a declaration of a prototype bean, which is declared as having a field of type `Integer` that is initialized with a random value when the constructor is called.

```
package com.apress.cems.scopes;
...
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;

import java.util.Random;

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class Salary {
    private Logger logger = LoggerFactory.getLogger(Salary.class);

    private Integer amount;

    public Salary() {
        logger.info(" -> Creating new Salary bean");
        Random rand = new Random();
        this.amount = rand.nextInt(10_000) +  50_000;
    }

    public Integer getAmount() {
        return amount;
    }
}
```

> **!** Fun fact: Using `@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)` does nothing. Because the Spring IoC container is not being told what kind of proxy to wrap the bean in, instead of throwing an error, just creates a singleton. So if you really want to customize scope `proxyMode` attribute must be set.

Which value should be used for the `proxyMode` in the previous implementation to make sure that it always gets a fresh new instance when the bean is accessed? If you really want to delegate that decision to Spring, you can use `proxyMode = ScopedProxyMode.DEFAULT` and the container will play it safe and create a CGLIB-based class proxy by default. But if you want to make the decision, just look at the class code. If the class implements an interface, then `proxyMode = ScopedProxyMode.INTERFACES` can be used. But if the class does not implement an interface, the only possible option is `proxyMode = ScopedProxyMode.TARGET_CLASS`.

So, to fix the preceding bean declaration, we should annotate the class properly.

```
package com.apress.cems.scopes;
...
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;

import java.util.Random;

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE,
       , proxyMode = ScopedProxyMode.TARGET_CLASS)
public class Salary {
    private Logger logger = LoggerFactory.getLogger(Salary.class);

    private Integer amount;

    public Salary() {
        logger.info(" -> Creating new Salary bean");
        Random rand = new Random();
        this.amount = rand.nextInt(10_000) +  50_000;
    }
```

```
    public Integer getAmount() {
        return amount;
    }
}
```

And now that we have the proper annotation for `Salary`, to make this bean a prototype, we can declare the `Employee` class, the type for our singleton bean that depends on the prototype bean declared previously.

```
package com.apress.cems.scopes;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Employee {
    private Salary salary;

    public Employee(Salary salary) {
        this.salary = salary;
    }

    @Autowired
    public void setSalary(Salary salary) {
        this.salary = salary;
    }

    public Salary getSalary() {
        return salary;
    }
}
```

We're all set. How do we test that this works as intended? The simplest way is to just get a reference to the `salary` bean and call `employee.getSalary().getAmount()` a few times and if we get a different value, this means it works.

```
package com.apress.cems.scopes;

import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

import static org.junit.jupiter.api.Assertions.assertNotNull;

public class AppConfigTest {
    private Logger logger = LoggerFactory.getLogger(AppConfigTest.class);

    @Test
    void testBeanLifecycle() {
        ConfigurableApplicationContext ctx =
             new AnnotationConfigApplicationContext(AppConfig.class);
        ctx.registerShutdownHook();

        Employee employee = ctx.getBean(Employee.class);
        assertNotNull(employee);

        Salary salary = employee.getSalary();
        assertNotNull(salary);
        logger.info("Salary bean actual type: {}", salary.getClass().
        toString());

        logger.info("Salary: {}", salary.getAmount());
        logger.info("Salary: {}", salary.getAmount());
        logger.info("Salary: {}", salary.getAmount());
    }
}
```

The configuration class for this example is named `AppConfig` and is just an empty configuration class annotated with `@Configuration` and `@ComponentScan` and not really relevant for the topic in this section so it won't be depicted here, but you can find it in the source code for this book.

When I ran the example, I saw the following in the log.

```
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
       singleton bean 'employee'
...
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'scopedTarget.salary'
INFO  c.a.c.s.Salary -  -> Creating new Salary bean
INFO  c.a.c.s.AppConfigTest - Salary: 59521
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'scopedTarget.salary'
INFO  c.a.c.s.Salary -  -> Creating new Salary bean
INFO  c.a.c.s.AppConfigTest - Salary: 52210
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'scopedTarget.salary'
INFO  c.a.c.s.Salary -  -> Creating new Salary bean
INFO  c.a.c.s.AppConfigTest - Salary: 54566
...
```

In the preceding log, you can see that our proxy works, every time we access the `salary` bean to retrieve the `amount` property, the `salary` bean is accessed by calling `getAmount()` and a new amount is returned. But what exactly is happening *under the hood*? If we look in the console log and scroll up a little, we find something that looks very weird.

```
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory -
          Creating shared instance of singleton bean 'employee'
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'employee'
DEBUG o.s.b.f.s.DefaultListableBeanFactory -
           Creating shared instance of singleton bean 'salary'
...
TRACE o.s.a.f.CglibAopProxy - Creating CGLIB proxy: SimpleBeanTargetSource
for target
        bean 'scopedTarget.salary' of type [com.apress.cems.scopes.Salary]
```

```
TRACE o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance of
bean 'salary'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Autowiring by type from bean
name
  'employee' via constructor to bean named 'salary'
...
```

The first underlined line in the previous snippet says *Creating shared instance of singleton bean 'salary'*.

? Wait, what??? What the actual...? What is the Spring IoC container doing? Wasn't the salary bean supposed to be a prototype? Why are the logs saying it is a singleton?

Remember how I mentioned that a proxy bean wraps around the intended bean, and the proxy is injected into the dependent bean? Well, the bean named `salary` is that proxy, and that bean is a singleton. It can easily be proven. We can modify the test class and add the following two lines.

```
Salary salary = employee.getSalary();
logger.info("Salary bean actual type: {}", salary.getClass().toString());
```

The two previous lines, print the type of the instance. If we run the test class again, we can see the following in the log.

```
...
INFO  c.a.c.s.AppConfigTest - Salary bean actual type: class
  com.apress.cems.scopes.Salary$$EnhancerBySpringCGLIB$$474c423f]
...
```

So, when the `employee` bean is created, it requires a dependency of type `Salary`. Since it's not practical to call the setter repeatedly to inject a new instance of type `Salary`, because that defeats the purpose of agnostic dependency injection, the `salary` bean that was injected is a proxy bean. This bean is of a type generated internally that is a subclass of `Salary` (because of the `proxyMode = ScopedProxyMode.TARGET_CLASS`). This means that the same methods can be called on it as on a `Salary` bean, but because it is a different type of bean, the methods do something else. In our case, calling `getAmount()`

on the proxy bean causes it to create a new instance of `Salary`, call `getAmount()` on it and returns the result. This behavior is depicted visually in Figure 2-14.

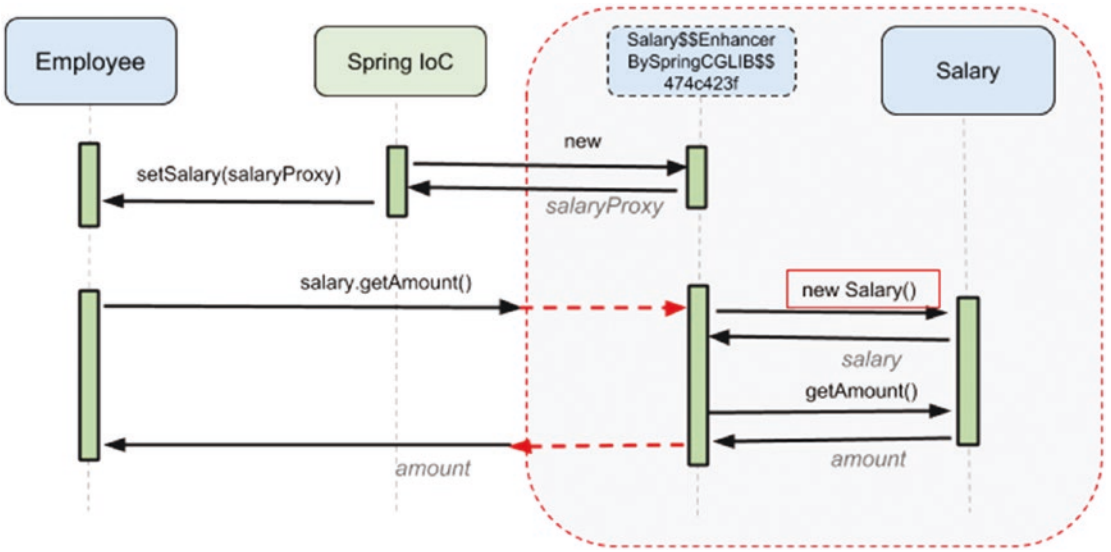

Figure 2-14. *Proxy bean behavior for scope prototype*

The `employee` bean calls the `getAmount()` on the proxy bean and gets the result, which is agnostic to how that result is obtained. The proxy bean ensures the prototype behavior by creating a new instance every time it is being accessed. The instances being created internally are referred to in the log as `scopedTarget.salary`.

Proxies created with `proxyMode = ScopedProxyMode.TARGET_CLASS` are referred to as a *CGLIB-based class proxy*, because they are created at runtime by subclassing the target type. To do this, additional libraries are necessary; in this case, CGLIB, which fortunately was included in Spring so that you won't need to declare it as a dependency in your project configuration. The disadvantage of these proxies is that they can only intercept public methods, since non-public methods cannot be overridden. Also, if for some reason, you would need to declare your class or methods final, using this type of proxying is not possible.

Proxies created with the `proxyMode = ScopedProxyMode.INTERFACES` are referred to as *JDK interface-based proxies*, because they are created at runtime by implementing the same interface as the target class and do not require additional libraries in the classpath. This unfortunately means that the class must implement at least one interface. The advantage here is that interfaces definitely do not have non-public methods that might escape proxying. 😉

! Starting with Java 8, interfaces can be declared to contain *private* and *default* methods. For obvious reasons, related to their access modifier, private methods are not proxied. Default methods are methods that are declared in the interface, so that classes implementing the interface don't have to. They are inherited by the classes, so they are proxied just like any normal method, with the specific behavior being executed before the call being forwarded to the target object.

Proxy beans are quite powerful. Spring uses proxy beans to add behavior to developers created beans, behavior which implemented explicitly would lead to a lot of redundancy (e.g., adding transactional behavior), but depending on the type of behavior we need added it can lead to performance issues. In our example, because we need a new salary every time we call `getAmount()` multiple beans are created and discarded, imagine what would happen if the `Salary` class would define some big object to be stored in memory or that the application is running on a system with little memory (memory issues might be unavoidable). The discussion about proxies will stop here because it continues in Chapter 4.

The `@Scope` annotation can annotate `@Bean` declarations as well. The bean declaration in the `Salary` class is identical to the following.

```
package com.apress.cems.scopes;

import org.springframework.context.annotation.*;

@Configuration
@ComponentScan(basePackages = {"com.apress.cems.scopes"} )
public class AppConfig {

    @Bean
    @Scope(value = "prototype",
        proxyMode = ScopedProxyMode.TARGET_CLASS)
    Salary salary(){
        return new Salary();
    }
}
```

We can even create a specialization of the `@Scope` annotation specific to `Salary` instances.

```
package com.apress.cems.scopes;

import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.core.annotation.AliasFor;

import java.lang.annotation.*;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public @interface SalaryScope  {

    /**
     * Alias for {@link Scope#proxyMode}.
     * <p>Defaults to {@link ScopedProxyMode#TARGET_CLASS}.
     */
    @AliasFor(annotation = Scope.class)
    ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET_CLASS;
}
```

** The list of annotations applied to the `@SalaryScope` annotation are the same ones that annotate the `@Scope` annotation and have the following purpose.

- `@Target` indicates the contexts in which an annotation type is applicable; in our case, the annotation is applicable at class and method levels.
- `@Retention` indicates how long annotations with the annotated type are to be retained; in our case, they are recoded at compile time and they are retained by the VM at runtime.
- `@Documented` indicates to a tool like Javadoc that annotations like these must be displayed in its output.

So, the previous bean declaration can be simplified to the following.

```
package com.apress.cems.scopes;

import org.springframework.context.annotation.*;

@Configuration
@ComponentScan(basePackages = {"com.apress.cems.scopes"} )
public class AppConfig {

    @Bean
    @SalaryScope
    Salary salary(){
        return new Salary();
    }
}
```

This is all that can be said about bean scopes, and it is more than enough information for you to pass the exam.☺

! Since this is quite an interesting section, some practice is in order. In project `chapter02/config-practice` look in the `com.apress.cems.scopes` package. It contains classes `Employee` and `Salary`. The Salary class is used to declare a bean of scope prototype. TODO 14 requires you to modify the implementation to make use of JDK-based interface proxying. TODO 15 requires you to create a specialization of the `@Scope` annotation equivalent to the configuration used on your bean.

Use the `com.apress.cems.scopes.AppConfigTest` class to test your implementation.

A proposed solution can be found in the `chapter02/config` project in the `com.apress.cems.scopes` package.

@AliasFor

In Spring 4.2, the `@AliasFor` annotation was added. This annotation is set on annotation attributes to declare aliases for them. And since I've just recently mentioned the `@Scope` annotation, let's take a look at its source code.[18]

```
 package org.springframework.context.annotation;
 ...
 @Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Scope {
        @AliasFor("scopeName")
        String value() default "";

        @AliasFor("value")
        String scopeName() default "";

        ScopedProxyMode proxyMode() default ScopedProxyMode.DEFAULT;
}
```

As you can see, attributes `value` and `scopeName` are used as aliases for each other, so they are interchangeable. Thus, the following three `@Scope` annotation usages are equivalent.

```
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
@Scope(scopeName = ConfigurableBeanFactory.SCOPE_SINGLETON)
```

And even more can be done with it. Aliases for meta-annotation attributes can be declared. In this case, the attribute annotated with `@AliasFor` is an alias for an attribute in another annotation. The best example here is for the `@Component` and `@Repository` annotations. The `@Component` is a meta-annotation, because it can annotate other

[18]You can inspect it on GitHub here: https://github.com/spring-projects/spring-framework/blob/master/spring-context/src/main/java/org/springframework/context/annotation/Scope.java or just click <Ctrl> + left-click or <Command> + left-click, and if you have an Internet connection, IntelliJ IDEA will download the code for you.

annotations; this is a fact that is indicated by the `@Target(ElementType.TYPE)`,[19] which is declared as follows.

```
 package org.springframework.stereotype;
 ...
 @Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Indexed
public @interface Component {
...
        String value() default "";
}
```

The `@Repository`, which is its specialization for declaring data access objects called *repositories*|, declares an attribute named `value` that is declared as an alias for the `value` attribute of the `@Component` annotation.

```
package org.springframework.stereotype;
 ...
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Repository {

...
        @AliasFor(annotation = Component.class)
        String value() default "";
}
```

The `@AliasFor` on the `value` attribute ensures that using `@Repository("myRepo")` on a bean, the bean will be named `myRepo`. Since the `@Component` annotation has an attribute named *value*, the `@AliasFor` annotation does not even have to reference the

[19] *java.lang.annotation.ElementType.TYPE* includes: class, interface (including annotation type), or enum declaration

attribute name, because it matches the attribute name in the `@Repository` annotation. So `@AliasFor(annotation = Component.class)` is equivalent to `@AliasFor(annotation = Component.class, attribute = "value")`.

Bean Lifecycle Under the Hood

In the previous section, we could not mention bean scope unless we made a connection with the bean lifespan. This section covers the steps that are executed when a Spring application is run. The bean starts up as a class that is annotated with a stereotype annotation or is instantiated in a method annotated with `@Bean` from a configuration class. In all the previous sections, the focus was on the technical details of a configuration and what an application context is; the references to bean creation steps were scarce. So we need to fix that.

A Spring application has a lifecycle of three phases.

- **Initialization**: In this phase, bean definitions are read, beans are created, dependencies are injected, and resources are allocated, also known as the bootstrap phase. After this phase is complete, the application can be used.
- **Use**: In this phase, the application is up and running. It is used by clients, and beans are retrieved and used to provide responses for their requests. This is the main phase of the lifecycle and covers 99% of it.
- **Destruction**: The context is being shut down, resources are released, and beans are handed over to the garbage collector.

These three phases are common to every type of application, whether it is a JUnit System test, a Spring or JEE web, or an enterprise application. Look at the following code snippet (the sources can be found in the `/chapter-02/config` project); it was modified to make it obvious where each phase ends.

```
package com.apress.cems.lc;
import org.springframework.context.ConfigurableApplicationContext;
...
public class ApplicationContextTest {
    private Logger logger = LoggerFactory.getLogger(ApplicationContextTest.
    class);
```

```
    @Test
    void testSimpleBeans() {
        ConfigurableApplicationContext ctx =
             new AnnotationConfigApplicationContext(DataSourceCfg.class);
        ctx.registerShutdownHook();
        logger.info(" >> init done.");

        DataSource dataSource = ctx.getBean(DataSource.class);
        assertNotNull(dataSource);

        logger.info(" >> usage done.");
    }
}
// DataSourceCfg.java contents
@Configuration
@PropertySource("classpath:db/prod-datasource.properties")
public class DataSourceCfg {
    @Value("${driverClassName}")
    private String driverClassName;
    @Value("${url}")
    private String url;
    @Value("${username}")
    private String username;
    @Value("${password}")
    private String password;

    @Bean
    public static PropertySourcesPlaceholderConfigurer
        propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driverClassName);
        ds.setUrl(url);
        ds.setUsername(username);
```

```
        ds.setPassword(password);
        return ds;
    }
}
```

The initialization phase of a Spring application ends when the application context initialization process ends. If the logger for the Spring Framework is set to TRACE before the *init done* message is printed, a lot of log entries show what the Spring IoC container is doing. The following is a simplified sample.

```
TRACE o.s.c.a.AnnotationConfigApplicationContext - Refreshing
 <<1>> ..AnnotationConfigApplicationContext@fa36558, started on ...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
singleton bean
'org.springframework.context.annotation.
internalConfigurationAnnotationProcessor'
...
TRACE o.s.c.a.ConfigurationClassBeanDefinitionReader - Registering bean
definition
<<2>> for @Bean method ..DataSourceCfg.
propertySourcesPlaceholderConfigurer()
TRACE o.s.c.a.ConfigurationClassBeanDefinitionReader - Registering bean
definition
   for @Bean method ..DataSourceCfg.dataSource()
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
singleton bean
<<3>>  'propertySourcesPlaceholderConfigurer'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
singleton bean
   'org.springframework.context.annotation.internalAutowiredAnnotation
   Processor'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
singleton bean
   'org.springframework.context.annotation.internalCommonAnnotationProcessor'
...
```

```
TRACE o.s.b.f.s.DefaultListableBeanFactory - Pre-instantiating singletons in
..DefaultListableBeanFactory@7d3e8655: defining beans
[..internalConfigurationAnnotationProcessor,
..internalAutowiredAnnotationProcessor,
..internalCommonAnnotationProcessor,... dataSourceCfg,propertySources
PlaceholderConfigurer,
  dataSource]; root of factory hierarchy
...
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'dataSourceCfg'
TRACE o.s.b.f.a.InjectionMetadata - Registered injected element on class
  [com.apress.cems.lc.DataSourceCfg$$EnhancerBySpringCGLIB$$8f107e20]:
  AutowiredFieldElement for private java.lang.String
   ..DataSourceCfg.driverClassName
TRACE o.s.b.f.a.InjectionMetadata - Registered injected element on class
  [com.apress.cems.lc.DataSourceCfg$$EnhancerBySpringCGLIB$$8f107e20]:
   AutowiredFieldElement for private java.lang.String ..DataSourceCfg.
   username
...
DEBUG o.s.c.e.PropertySourcesPropertyResolver - Found key 'driverClassName'
   in PropertySource 'class path resource [db/prod-datasource.properties]'
   with value of type String
TRACE o.s.u.PropertyPlaceholderHelper - Resolved placeholder
'driverClassName'
TRACE o.s.c.e.PropertySourcesPropertyResolver - Searching for key
'username'
  in PropertySource 'class path resource [db/prod-datasource.properties]'
TRACE o.s.u.PropertyPlaceholderHelper - Resolved placeholder 'username'
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared instance of
   singleton bean 'dataSourceCfg'
<<4>>  DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating shared
instance of
  singleton bean 'dataSource'
DEBUG o.s.j.d.DriverManagerDataSource - Loaded JDBC driver: oracle.jdbc.
OracleDriver
```

```
TRACE o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance of
bean 'dataSource'
INFO  c.a.c.l.ApplicationContextTest -  >> init done.
TRACE o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance of
   singleton bean 'dataSource'
INFO  c.a.c.l.ApplicationContextTest -  >> usage done.
DEBUG o.s.c.a.AnnotationConfigApplicationContext - Closing
 ..AnnotationConfigApplicationContext@fa36558, started on ...
<<7>>   TRACE o.s.b.f.s.DefaultListableBeanFactory - Returning cached
instance of
  singleton bean 'lifecycleProcessor'
<<9>>TRACE o.s.b.f.s.DefaultListableBeanFactory - Destroying singletons in
 ..DefaultListableBeanFactory@7d3e8655: defining beans
 [..internalConfigurationAnnotationProcessor,..
internalAutowiredAnnotationProcessor,
 ..internalCommonAnnotationProcessor,dataSourceCfg,
 propertySourcesPlaceholderConfigurer,dataSource]; root of factory hierarchy
```

Although the log is incomplete, all the important details were marked (either by numbers in <<*>> or by underlining). To make things easier, the following steps were marked in the log with the corresponding step number.

1. The application context is initialized.
2. The bean definitions are loaded (from the class `DataSourceCfg` in this case).
3. The bean definitions are processed (in our case a bean of type `PropertySourcesPlaceholderConfigurer` is created and used to read the properties from `prod-datasource.properties`, which are then added to the `dataSource` bean definition).
4. Beans are instantiated.
5. Dependencies are injected (not visible from the log, since the `dataSource` bean does not require any dependencies).
6. Beans are processed (also not visible from the log, since the `dataSource` does not have any processing defined).
7. Beans are used.

8. The context starts the destruction process. (This is not visible in the log.)
9. Beans are destroyed.

Figure 2-15 depicts the whole application context lifecycle and the bean lifecycle.

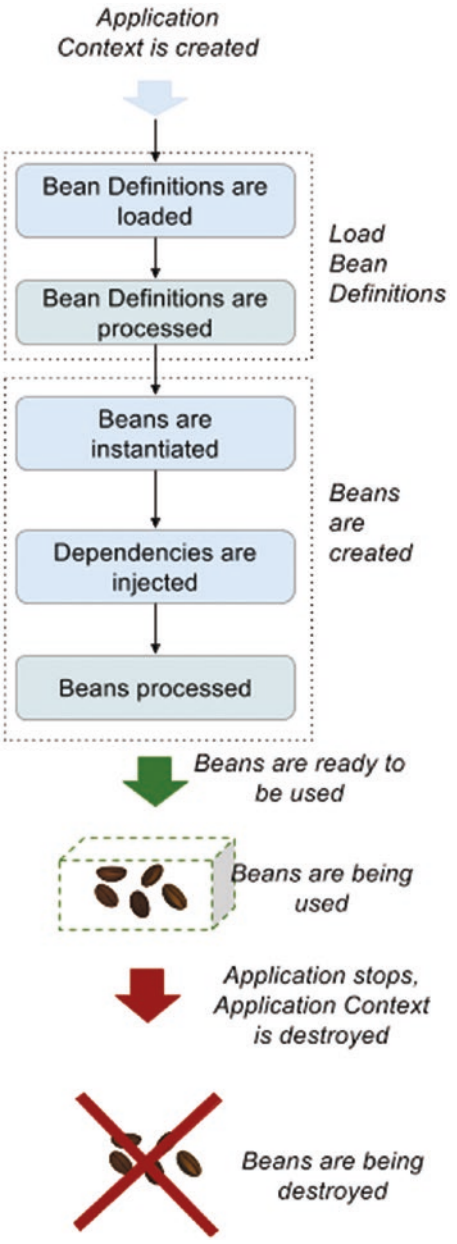

Figure 2-15. *Application context and bean lifecycle*

In the **Load Bean Definition** step, the Configuration classes are parsed, Java classes are scanned for configuration annotations and bean definitions are loaded into the application context, indexed by names. The bean definitions are then processed by beans called *bean factory post processors* that are automatically picked up by the application context, created, and applied before any other beans are created. These bean types implement the `org.springframework.beans.factory.config.BeanFactoryPostProcessor` interface, which is how the application context recognizes them. In the previous example, the `PropertySourcesPlaceholderConfigurer` type was mentioned. This class is a type of bean factory post processor implementation that resolves placeholders like `${propName}` within bean declarations by injecting property values read from the Spring environment and its set of property sources, declared using the `@PropertySource`.

! The `propertySourcesPlaceholderConfigurer` bean declaration needs to be a static method picked up when the context is created, earlier than the configuration class annotated with `@Configuration`, so the property values are added to the Spring Environment and become available for injection in the said configuration class, before this class is initialized.

Since the `propertySourcesPlaceholderConfigurer` modifies the declaration of a configuration class, this obviously means that these classes are proxied by Spring IoC container, and this obviously means that these classes cannot be final. The infrastructure bean responsible for bootstrapping the processing of `@Configuration` annotated classes is the bean named `internalConfigurationAnnotationProcessor` and is of type `org.springframework.context.annotation.ConfigurationClassPostProcessor` which is an implementation of `BeanFactoryPostProcessor`.

The `BeanFactoryPostProcessor` interface contains a single method definition (annotated with `@FunctionalInterface` so that it can be used in lambda expressions) that must be implemented: `postProcessBeanFactory(ConfigurableListableBeanFactory)`. The parameter with which this method will be called is the factory bean used by the application context to create the beans. Developers can create their own bean factory post processors and define a bean of this type in their configuration, and the application context will make sure to invoke them.

Figure 2-16 depicts the effect of a `PropertySourcesPlaceholderConfigurer` bean used on the `dataSourceCfg` configuration bean definition.

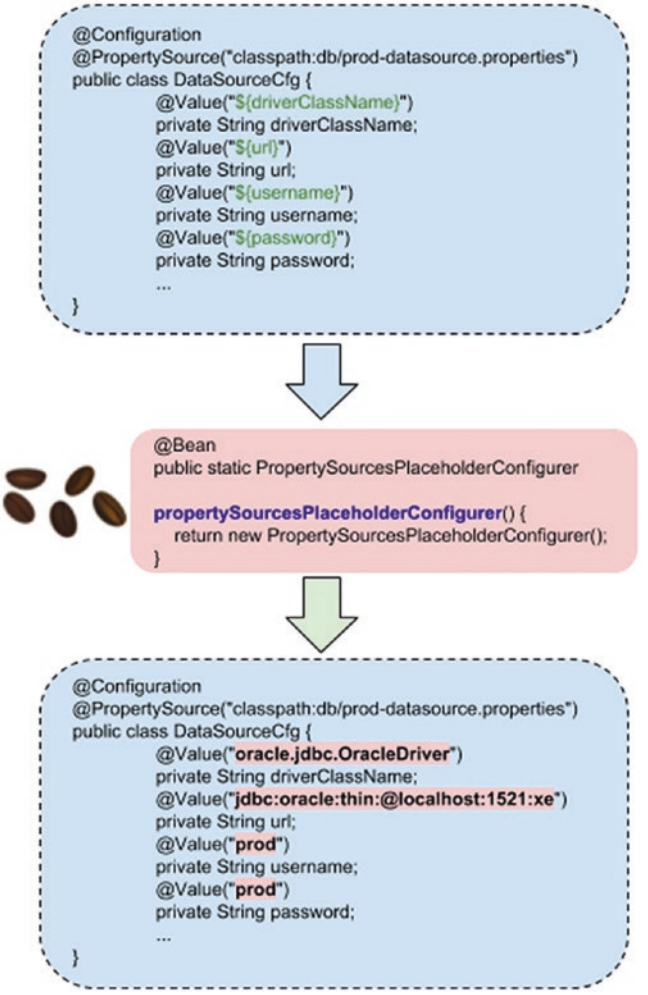

Figure 2-16. *Effect of a PropertySourcesPlaceholderConfigurer on a bean definition using placeholders*

These types of beans are useful, because they can process bean definitions at runtime and change them based on resources that are outside the application, so the application does not need to be recompiled to change a bean definition. It is also useful in development too, because configuration values can be swapped easily, without a recompile being needed, and this is especially useful when applications get bigger.

In the preceding example, if the property values used to configure the `dataSourceCfg` bean are hard-coded in the configuration class, when the database connection details change, the configuration class has to be changed, and the application has to be recompiled and restarted. Not that practical, right?

The **bean creation** process can be split into a small number of stages.

1. In the first stage, **the beans are instantiated**. This basically means that the bean factory is calling the constructor of each bean. If the bean is created using constructor dependency injection, the dependency bean is created first and then injected where needed. For beans that are defined in this way, the instantiation stage coincides with **the dependency injection** stage.
2. In the second stage, **dependencies are injected**. For beans that are defined having dependencies injected via setter, this stage is separate from the instantiation stage. The same goes for dependencies inject using field injection.
3. The next stage is the one in which **bean post process beans are invoked before initialization**.[20]
4. In this stage, **beans are initialized**.
5. The next stage is the one in which **bean post process beans are invoked after initialization**.

As you can see, there are two stages that involve the bean post process beans being called. What is the difference between them? The stage between them, the initialization stage, splits them into bean post processors that are invoked before it and after it. Since the bean post processor subject is quite large, the initialization stage will be covered first.

Long story short, a bean can be defined so that a certain method is called right after the bean is created and dependencies are injected to execute code. This method is called an *initialization method.* When using Java configuration, this method is set by annotating it with the `@PostConstruct` annotation. The method must return void. The must have no arguments defined, and can have any access rights since Spring uses

[20]Unfortunately, there is no other way to formulate this. We are talking about beans that have the ability to post process other beans. And they are called, confusingly: **bean post process beans.**

reflection to find and call it (just make sure you add the proper configuration in your `module-info.java` if needed).

Some developers recommend that you make it private so that it cannot be called from outside the bean, to make sure that Spring has total control over it, and so that it calls it only one time during the bean lifecycle. In the following code snippet, you can see the configuration of such a method for a bean of type `ComplexBean`. The bean is a very simple one with two dependencies: one is provided using a setter and one is set with an instance created within the initialization method. This means that the object created within the method is not a bean; it is not managed by Spring.

```
package com.apress.cems.lc;
...
@Component
public class ComplexBean {
    private Logger logger = LoggerFactory.getLogger(ComplexBean.class);

    private SimpleBean simpleBean;

    private AnotherSimpleBean anotherSimpleBean;

    public ComplexBean() {
        logger.info("Stage 1: Calling the constructor.");
    }

    @Autowired
    public void setSimpleBean(SimpleBean simpleBean) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean = simpleBean;
    }

    /**
     * The initialization method.
     * Just for fun: it instantiates the anotherSimpleBean only
     * if the current time is even.
     */
```

```
    @PostConstruct
    private void initMethod() {
        logger.info("Stage 3: Calling the initMethod.");
        long ct = System.currentTimeMillis();
        if (ct % 2 == 0) {
            anotherSimpleBean = new AnotherSimpleBean();
        }
    }
}
```

When initializing an application context based on the preceding configuration, the following is what is seen in the log.

```
 ...
TRACE o.s.b.f.s.DefaultListableBeanFactory -
      Creating instance of bean 'anotherSimpleBean'
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'complexBean'
INFO  c.a.c.l.ComplexBean - Stage 1: Calling the constructor.
TRACE o.s.c.a.CommonAnnotationBeanPostProcessor -
        Found init method on class com.apress.cems.lc.ComplexBean:
                private void com.apress.cems.lc.ComplexBean.initMethod()
TRACE o.s.c.a.CommonAnnotationBeanPostProcessor -
   Registered init method on class [com.apress.cems.lc.ComplexBean]:
   o.s.b.f.a.InitDestroyAnnotationBeanPostProcessor$LifecycleElement@21c6920f
...
TRACE o.s.b.f.a.AutowiredAnnotationBeanPostProcessor - Autowiring by type
    from bean name 'complexBean' to bean named 'simpleBean'
INFO  c.a.c.l.ComplexBean - Stage 2: Calling the setter.
TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Invoking init method
   on bean 'complexBean': private void com.apress.cems.lc.ComplexBean.
   initMethod()
INFO  c.a.c.l.ComplexBean - Stage 3: Calling the initMethod.
TRACE o.s.b.f.s.DefaultListableBeanFactory -
      Finished creating instance of bean 'complexBean'
 ...
```

As you can see, the bean is created, dependencies are injected, and then the `initMethod` is called. Annotations support was introduced in Spring 2.5 for a small set of annotations. Since then, Spring has evolved, and currently with Spring 5.0, an application can be configured using only annotations (stereotypes and the complete set of Java configuration).

The `@PostConstruct` annotation is part of the JSR 250[21] and is used on a method that needs to be executed after dependency injection is done to perform initialization. The annotated method must be invoked before the bean is used, and, like any other initialization method chosen, may be called **only once** during a bean lifecycle. If there are no dependencies to be injected, the annotated method will be called after the bean is instantiated. Only one method should be annotated with `@PostConstruct`. To use this annotation, the `jsr250` library must be in the classpath and dependency of the `jsr250.api` module must be configured in the `module-info.java` file. The method annotated with `@PostConstruct` is picked up by enabling component scanning (annotating configuration classes with `@ComponentScanning`) and called by a pre-init bean named `org.springframework.context.annotation.internalCommonAnnotationProcessor`[22] of a type that implements the `org.springframework.beans.factory.config.BeanPostProcessor` interface named `CommonAnnotationBeanPostProcessor`. Classes implementing this interface are factory hooks that allow for modifications of bean instances. The application context autodetects these types of beans and instantiates them before any other beans in the container, since after their instantiation they are used to manipulate other beans managed by the IoC container.

The `BeanPostProcessor` interface declares two methods to be implemented, which have been declared as default interface methods so that the developer can freely implement only the one that presents interest. The methods are named `postProcessBeforeInitialization` and `postProcessAfterInitialization`. In Figure 2-17, the two pink rectangles depict when the bean post processor is invoking methods on the bean.

[21]Java Request Specification 250 https://jcp.org/en/jsr/detail?id=250.

[22]The name of this bean and its purpose and a few others are part of the *org.springframework.context.annotation.AnnotationConfigUtils* class. You can see the code here https://github.com/spring-projects/spring-framework/blob/3a0f309e2c9fdbbf7fb2d348be861528177f8555/spring-context/src/main/java/org/springframework/context/annotation/AnnotationConfigUtils.java

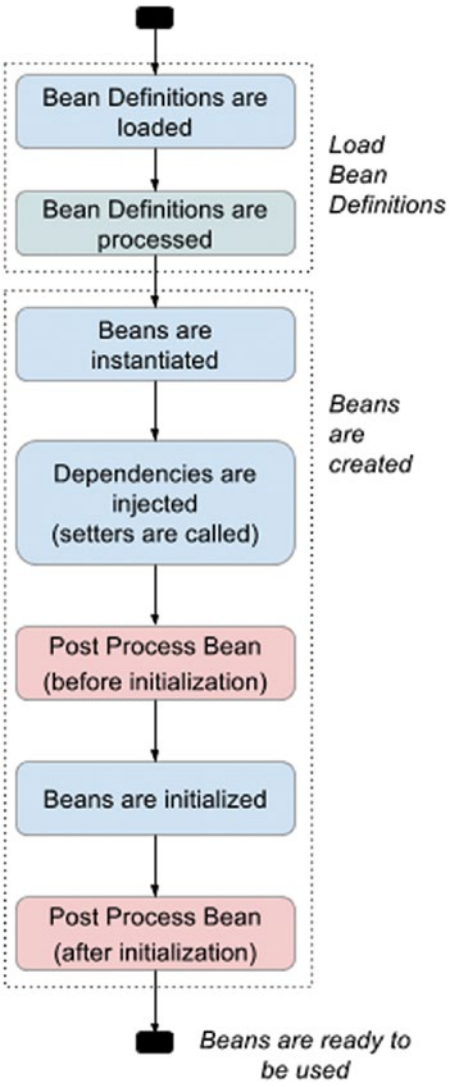

Figure 2-17. *Bean lifecycle expanded*

Typically, post processors that populate bean properties pick up methods annotated with `@PostConstruct` implement `postProcessBeforeInitialization`, while post processors that wrap beans with proxies will normally implement `postProcessAfterInitialization`.

! From the behavior described so far, you've probably realized that invocation of the `@PostConstruct` annotated method is specific to the third stage of bean creation when bean post process beans are invoked **before initialization**, but because there is no other actual initialization method defined, the line is kinda fuzzy here as well. So the annotated method is called the initialization method because it is the only method with this purpose declared for this bean.

If an `init` method is not specified using the `@PostConstruct` annotation, there are other two ways of initializing a bean available in Spring. The following is the complete list.[23]

- Implementing the `org.springframework.beans.factory.InitializingBean` interface and providing an implementation for the method `afterPropertiesSet` (most times, this is not recommended since it couples the application code with Spring infrastructure).
- Annotating with `@PostConstruct` the method that is called right after the bean is instantiated and dependencies injected.
- Using Java configuration by annotating an initialization method with `@Bean(initMethod="...")`. This method of initialization is very useful when the code comes from a third-party library or a dependency and cannot be edited.

! Fun fact: all these three initialization methods can be configured together for the same bean, although technically I cannot see a valid reason for doing that.

The `InitializingBean` interface is implemented by beans that need to react once all their dependencies have been injected by the BeanFactory. It defines a single method that is named accordingly, `afterPropertiesSet()`, which is called by the factory creating the bean. And, because this method is not the responsibility of a bean post processor, its execution corresponds to the initialization stage of a bean. In the following code snippet, the `ComplexBean` class is modified to implement this interface.

[23]The XML way of initialization configuration is not taken into consideration here.

```
package com.apress.cems.ib;

import org.springframework.beans.factory.InitializingBean;
  ...
@Component
public class ComplexBean implements InitializingBean {
    private Logger logger = LoggerFactory.getLogger(ComplexBean.class);

    private SimpleBean simpleBean;

    private AnotherSimpleBean anotherSimpleBean;

    public ComplexBean() {
        logger.info("Stage 1: Calling the constructor.");
    }

    @Autowired
    public void setSimpleBean(SimpleBean simpleBean) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean = simpleBean;
    }

    @Override
    public void  afterPropertiesSet() throws Exception {
        logger.info("Stage 3: Calling the afterPropertiesSet.");
        long ct = System.currentTimeMillis();
        if (ct % 2 == 0) {
            anotherSimpleBean = new AnotherSimpleBean();
        }
    }
}
```

If an application context is created using the previous configuration, the following is seen in the log.

```
...
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'complexBean'
INFO  c.a.c.l.ComplexBean - Stage 1: Calling the constructor.
TRACE o.s.b.f.s.DefaultListableBeanFactory - Eagerly caching bean
'complexBean'
    to allow for resolving potential circular references
TRACE o.s.b.f.a.AutowiredAnnotationBeanPostProcessor - Autowiring by type
from
    bean name 'complexBean' to bean named 'simpleBean'
INFO  c.a.c.l.ComplexBean - Stage 2: Calling the setter.
TRACE o.s.b.f.s.DefaultListableBeanFactory - Invoking afterPropertiesSet() on
    bean with name 'complexBean'
INFO  c.a.c.l.ComplexBean - Stage 3: Calling the afterPropertiesSet.
TRACE o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance of
bean 'complexBean'
...
```

Calling the `afterProperties` method is a responsibility of the same factory bean that creates the `complexBean`; in this case, the factory bean is an infrastructure bean of type `DefaultListableBeanFactory`.

? If the previous bean class would be modified to add an initialization method annotated with `@PostConstruct` when creating the bean, which method do you think will be executed first?

The third type of bean initialization is done by declaring beans using the `@Bean` annotation and by setting the value for the `initMethod` to the initialization method name. So, let's write a class named `AnotherComplexBean` that can configure a bean by using the `@Bean(initMethod = "...")` annotation.

```
package com.apress.cems.im;
   ...

  public class AnotherComplexBean {
    private Logger logger = LoggerFactory.getLogger(AnotherComplexBean.class);
```

```
    private SimpleBean simpleBean;

    private AnotherSimpleBean anotherSimpleBean;

    public AnotherComplexBean() {
        logger.info("Stage 1: Calling the constructor.");
    }

    @Autowired
    public void setSimpleBean(SimpleBean simpleBean) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean = simpleBean;
    }

    /**
     * The initialization method.
     * Just for fun: it instantiates the anotherSimpleBean only
     * if the current time is even.
     */
    public void  beanInitMethod() {
        logger.info("Stage 3: Calling the beanInitMethod.");
        long ct = System.currentTimeMillis();
        if (ct % 2 == 0) {
            anotherSimpleBean = new AnotherSimpleBean();
        }
    }
}
```

The configuration of the bean is quite simple.

```
 package com.apress.cems.im;
 import org.springframework.context.annotation.Bean;
...

@Configuration
@ComponentScan(basePackages = {"com.apress.cems.im"} )
public class AcbConfig {
```

```
    @Bean(initMethod = "beanInitMethod")
    AnotherComplexBean anotherComplexBean(){
        return new AnotherComplexBean();
    }
}
```

When executing a test class and creating an application context containing a bean of type `AnotherComplexBean`, if we look in the console, we can see the three stages of creating this bean in the printed log messages.

```
...
INFO  c.a.c.i.AnotherComplexBean - Stage 1: Calling the constructor
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'simpleBean'
TRACE o.s.b.f.a.AutowiredAnnotationBeanPostProcessor - Autowiring by type
from bean
   name 'anotherComplexBean' to bean named 'simpleBean'
INFO  c.a.c.i.AnotherComplexBean - Stage 2: Calling the setter.
TRACE o.s.b.f.s.DefaultListableBeanFactory - Invoking init
method  'beanInitMethod' on
  bean with name 'anotherComplexBean'
INFO  c.a.c.i.AnotherComplexBean - Stage 3: Calling the beanInitMethod.
TRACE o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance of
   bean 'anotherComplexBean'
...
```

When configuring a bean initialization method using `@Bean(initMethod = "...")`, the responsible for calling the method is the same bean factory responsible with the instantiation of the bean as well: `DefaultListableBeanFactory`. So the execution of the method matches the initialization stage of a bean.

We are at the end of the section, so now we have to cover what happens to beans after they have been used and are no longer needed and the application shuts down, or only the context managing them.[24] When a context is closed, it destroys all the beans; that is obvious. But some beans (singletons) work with resources that might refuse to

[24]Complex Spring applications can have multiple application contexts and they can be closed independently.

release them if they are not notified before destruction. In Spring, this can be done in two ways: [25]

- Modify the bean to implement the `org.springframework.beans.factory.DisposableBean` interface and provide an implementation for the `destroy()` method (not recommended, since it couples the application code with Spring infrastructure).
- Annotate a method with `@PreDestroy`, which is also part of JSR 250 and one of the first supported annotations in Spring.
- Configure a bean using `@Bean(destroyMethod="...")`.

The `destroy` method of a bean has the same purpose as the `finalize` method for POJOs. The `ComplexBean` was modified to make use of the `destroy` method.

```
package com.apress.cems.lc;
...
import javax.annotation.PreDestroy;

@Component
public class ComplexBean {
    private SimpleBean simpleBean;
    private AnotherSimpleBean anotherSimpleBean;

    public ComplexBean() {
        logger.info("Stage 1: Calling the constructor.");
    }

    @Autowired
    public void setSimpleBean(SimpleBean simpleBean) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean = simpleBean;
    }
```

[25] If XML configuration is considered, there are three ways.

```
   @PostConstruct
    private void initMethod() {
        logger.info("Stage 3: Calling the initMethod.");
        long ct = System.currentTimeMillis();
        if (ct % 2 == 0) {
            anotherSimpleBean = new AnotherSimpleBean();
        }
    }

    @PreDestroy
    private void destroy(){
        logger.info("Stage 4: Calling the destroy method.");
    }
}
```

To view something useful in the log, we need to close the application context gracefully, and this can be done by calling the `close()` or the `registerShutdownHook()` method.

```
@Test
public void testBeanCreation() {
       ConfigurableApplicationContext ctx =
           new AnnotationConfigApplicationContext(SimpleConfig.class);
        ctx.registerShutdownHook();

        ComplexBean complexBean = ctx.getBean(ComplexBean.class);
        assertNotNull(complexBean);
}
```

And *voila*. (Only the last part of the log is depicted, where the invocation of the destroy method is logged.)

```
...
TRACE o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
'complexBean'
INFO  c.a.c.l.ComplexBean - Stage 1: Calling the constructor.
...
```

```
TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Registered destroy method
    on class [com.apress.cems.lc.ComplexBean]: org.springframework.beans.
    factory.annotation.
    InitDestroyAnnotationBeanPostProcessor$LifecycleElement@4748d4bc
...
INFO  c.a.c.l.ComplexBean - Stage 2: Calling the setter.
TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Invoking init method on
bean
   'complexBean': private void com.apress.cems.lc.ComplexBean.initMethod()
INFO  c.a.c.l.ComplexBean - Stage 3: Calling the initMethod.
TRACE o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance of
   bean 'complexBean'
...
DEBUG o.s.c.a.AnnotationConfigApplicationContext -
  Closing o.s.c.a.AnnotationConfigApplicationContext@fa36558, started on ...
TRACE o.s.b.f.s.DefaultListableBeanFactory - Destroying singletons in
   ..DefaultListableBeanFactory@14dd7b39: defining beans
   (...,simpleConfig,anotherSimpleBean,complexBean,
   dataSourceCfg,simpleBean,...; root of factory hierarchy
TRACE o.s.c.a.CommonAnnotationBeanPostProcessor - Invoking destroy method on
   bean 'complexBean': private void com.apress.cems.lc.ComplexBean.destroy()
INFO  c.a.c.l.ComplexBean - Stage 4: Calling the destroy method.
```

As you can see, the `destroy()` method is registered and invoked by the `CommonAnnotationBeanPostProcessor` bean.

! For the `destroy` method, the same rules and recommendations regarding signature and accessors apply as for the `init` method.

The destroy method may be called **only once** during the bean lifecycle.

The destroy method can have any accessor; some developers even recommend to make it private, so that only Spring can call it via reflection.

The destroy method must not have any parameters.

The destroy method must return void.

And in case it was not obvious by now (and from the log), since only singleton beans get to live until the application context is shutdown, all of these rules apply only to singleton beans.

As you probably suspect by now, there is a matching interface for `InitializingBean` named `DisposableBean` that provides a method to be implemented for bean destruction, and it is named (obviously) `destroy()`. If the `ComplexBean` implements `DisposableBean`, the dirty work is delegated to a bean of type `org.springframework.beans.factory.support.DisposableBeanAdapter`.[26]

```
 package com.apress.cems.ib;
 import org.springframework.beans.factory.DisposableBean;

public class ComplexBean implements implements InitializingBean,
     DisposableBean {
    private Logger logger = LoggerFactory.getLogger(ComplexBean.class);

    private SimpleBean simpleBean;

    private AnotherSimpleBean anotherSimpleBean;

    public ComplexBean() {
        logger.info("Stage 1: Calling the constructor.");
    }

    @Autowired
    public void setSimpleBean(SimpleBean simpleBean) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean = simpleBean;
    }
```

[26] *DisposableBeanAdapter* is an internal infrastructure bean type that performs various destruction steps on a given bean instance. Its code is available here: https://github.com/spring-projects/spring-framework/blob/master/spring-beans/src/main/java/org/springframework/beans/factory/support/DisposableBeanAdapter.java.

```
    @Override
    public void afterPropertiesSet() throws Exception {
        logger.info("Stage 3: Calling the afterPropertiesSet.");
        long ct = System.currentTimeMillis();
        if (ct % 2 == 0) {
            anotherSimpleBean = new AnotherSimpleBean();
        }
    }

    @Override
    public void destroy() throws Exception {
        logger.info("Stage 4: Calling the destroy method.");
    }
}
```

The difference between the `destroy()` method being declared with `@PreDestroy` and by implementing the `DisposableBean` can be seen in the console log.

```
INFO  c.a.c.i.ComplexBean - Stage 1: Calling the constructor.
TRACE o.s.b.f.a.AutowiredAnnotationBeanPostProcessor - Autowiring by type
from bean
   name 'complexBean' to bean named 'simpleBean'
...
INFO  c.a.c.i.ComplexBean - Stage 2: Calling the setter.
TRACE o.s.b.f.s.DefaultListableBeanFactory - Invoking afterPropertiesSet on
bean
    with name 'complexBean'
INFO  c.a.c.i.ComplexBean - Stage 3: Calling the afterPropertiesSet.
...
TRACE o.s.b.f.s.DefaultListableBeanFactory - Destroying singletons in
    ..DefaultListableBeanFactory@14dd7b39: defining beans
[...,complexBean,simpleBean,
     anotherSimpleBean]; root of factory hierarchy
TRACE o.s.b.f.s.DisposableBeanAdapter - Invoking destroy on bean with
   name 'complexBean'
INFO  c.a.c.i.ComplexBean - Stage 4: Calling the destroy method.
```

Also, if declaring a destroy method with the `@PreDestroy` annotation is not an option, there is the analogous attribute of the `@Bean` annotation used for configuring a destroy method, named `destroyMethod`.

```
 package com.apress.cems.im;
 import org.springframework.context.annotation.Bean;
 ...
@Configuration
@ComponentScan(basePackages = {"com.apress.cems.im"} )
public class AcbConfig {

    @Bean(initMethod = "beanInitMethod", destroyMethod = "beanDestroyMethod")
    AnotherComplexBean anotherComplexBean(){
        return new AnotherComplexBean();
    }

    @Bean
    SimpleBean simpleBean(){
        return new SimpleBean();
    }
}
```

When a context is created with the bean declared as in the previous code snippet, in the log, you see that the job of calling the `destroy()` method is delegated to the `DisposableBeanAdapter` as in the case where the method is declared by annotating it with `@PreDestroy`.

```
INFO  c.a.c.i.AnotherComplexBean - Stage 1: Calling the constructor.
TRACE o.s.b.f.a.AutowiredAnnotationBeanPostProcessor - Autowiring by type
from bean
   name 'anotherComplexBean' to bean named 'simpleBean'
...
INFO  c.a.c.i.AnotherComplexBean - Stage 2: Calling the setter.
...
TRACE o.s.b.f.s.DefaultListableBeanFactory - Destroying singletons in
    ..DefaultListableBeanFactory@14dd7b39: defining beans [...,another
    ComplexBean,simpleBean,
     anotherSimpleBean]; root of factory hierarchy
```

```
TRACE o.s.b.f.s.DisposableBeanAdapter - Invoking destroy method
        'beanDestroyMethod' on bean with name 'anotherComplexBean'
INFO  c.a.c.i.AnotherComplexBean - Stage 4: Calling the beanDestroyMethod.
```

! Since this is the end of a big section, there is an exercise for you. In `chapter02/config-practice`, there is a class named `FunBean` that has a comment `TODO 12` on it. Complete this class by adding a method annotated with `@PostConstruct`, one with `@PreDestroy`, then implement `InitializingBean` and `DisposableBean` and add an initialization and a destroy method do be configured with the `@Bean` annotation.

In the `FunBeanConfig` class, there is `TODO 13`, which requires you to declare a bean of type `FunBean` using the `@Bean` annotation and declare an initialization and a destroy method.

Add logging messages with numbers to each method to reflect the order of the execution for the methods based on the information provided to you so far.

Use the `FunBeanConfigPracticeTest` class to test your configuration.

A proposed solution is provided in the `chapter02/config` project.

Bean Declaration Inheritance

Since classes are templates for creating objects and they can be extended, the same is true for classes that are used to declare beans. A hierarchy of classes can create different beans, and injected values can be inherited or overwritten depending on the needs. Also, subclasses can declare their own fields that can be initialized separately from the parent bean declaration. Using abstract types is a recommended practice in Spring applications, but declaring different beans using classes in the same hierarchy requires for these classes to be instantiated, so we cannot have an abstract class on top of the hierarchy. This is why the following code snippet declares a superclass named `ParentBean` that has two `String` properties named `familyName` and `surname` which are both annotated with `@Value`, only one property value is initialized using the constructor, and one via a constructor.

```
package com.apress.cems.beans.inheritance;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class ParentBean {

    @Value("Smith")
    protected String familyName;

    protected String surname;

    public ParentBean(@Value("John")String surname) {
        this.surname = surname;
    }
    ...
}
```

A class named ChildBean extends this class. It is annotated with @Component and declares its own constructor that initializes the property inherited from the parent and its property named adult (which is set to false because a child is obviously not an adult).

```
package com.apress.cems.beans.inheritance;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class ChildBean extends ParentBean {

    private Boolean adult;

    public ChildBean(@Value("Lil' John") String surname,
           @Value("false")Boolean adult) {
        super(surname);
        this.adult = adult;
    }
...
}
```

The two bean declarations will be picked up by the Spring container and two beans will be created within the application context. Both beans will have the value of the `familyName` field equal to *Smith* the value injected in the `ParentBean` class. The bean of type `ChildBean` will have the value of its `surname` field set to *Lil' John*, as declared in the `ChildBean` class. This value overwrites the value of *John* declared in the `ParenBean` constructor. To test that the beans were initialized as we intended, a test class can be created that tests the equality of their fields, but that also prints all the fields of every bean.

The configuration class for this example is named `FamilyAppConfig` and is just an empty configuration class annotated with `@Configuration` and `@ComponentScan`.

```
 package com.apress.cems.beans.inheritance;

...

import static org.junit.jupiter.api.Assertions.*;

public class FamilyAppConfigTest {

    private Logger logger =
        LoggerFactory.getLogger(FamilyAppConfigTest.class);

    @Test
    void testSimpleBeans() {
        ConfigurableApplicationContext ctx =
                  new AnnotationConfigApplicationContext(FamilyAppConfig.
                  class);

        ParentBean parentBean = ctx.getBean("parentBean",ParentBean.class);
        assertNotNull(parentBean);

        ChildBean childBean = ctx.getBean("childBean", ChildBean.class);
        assertNotNull(childBean);

        assertEquals(parentBean.getFamilyName(),
                childBean.getFamilyName());
        assertNotEquals(parentBean.getSurname(),
                childBean.getSurname());
```

```
        logger.info(parentBean.toString());
        logger.info(childBean.toString());
        ctx.close();
    }
}
```

The fact that this class test executes correctly is all the proof we need that the beans were created exactly as we intended, but if we really want to make sure we can also check the logs.

```
DEBUG Refreshing ..AnnotationConfigApplicationContext
...
DEBUG Creating shared instance of singleton bean 'familyAppConfig'
DEBUG Creating shared instance of singleton bean 'childBean'
DEBUG Creating shared instance of singleton bean 'parentBean'
DEBUG FamilyAppConfigTest - ParentBean{ familyName='Smith', surname='John'}
DEBUG FamilyAppConfigTest - ChildBean{ familyName='Smith', surname='Lil' John'
        , isAdult=false}
DEBUG Closing ..AnnotationConfigApplicationContext
```

Bean declaration inheritance is not really such a big thing when configuration is done using annotations, because to share commons parts, the classes must be related. In XML, the classes do not really have to be in a hierarchy, they just have to share field names and configuration values can be inherited. Also, in XML if beans have the same type, the only bean that has to declare the type is the parent bean, and any bean that inherits from it can override it.

Another advantage of using XML is that abstract classes can provide a template for beans of types that extend the abstract class to be created. Let's assume that the CEMS[27] application must run in production on a cluster of three servers. Each server has its own database, and the configuration must mention them all. All of them are Oracle databases, so by using XML, the configuration file for connecting to the database would look like the following.

[27]Criminal Evidence Management System - the name is too long, using the acronym from now on 😁

```
<beans ...>
   <bean id="abstractDataSource" class="oracle.jdbc.pool.OracleDataSource"
                abstract="true">
        <property name="user" value="admin"/>
        <property name="loginTimeout" value="300"/>
    </bean>

      <bean id="dataSource-1" parent="abstractDataSource">
        <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:
        PET"/>
    </bean>

    <bean id="dataSource-2" parent="abstractDataSource">
        <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:
        PET"/>
    </bean>

    <bean id="dataSource-3" parent="abstractDataSource"
                    class="com.apress.CustomizedOracleDataSource">
        <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:
        PET"/>
        <property name="loginTimeout" value="100"/>
    </bean>
</beans>
```

By declaring a `<bean ../>` element with the attribute `abstract` set to `true`, we tell the Spring IoC container that this bean is not meant to be instantiated itself, but it is just serving as parent for concrete child bean definitions. And since this bean is not meant to be instantiated, the `class` attribute can even be set to an abstract class.

Writing all that configuration using Java would be a pain, so although XML is not a topic for the certification exam, there are cases when it would be more suitable to use XML configuration files.

Injecting Dependencies That Are Not Beans

Dependency injection with Spring is a vast subject, and this book was written with the intention of covering all the cases you might need in enterprise development. It was mentioned previously in the book that `scalar` values can be injected into

bean configurations, which opens the discussion about type conversion. In the previous examples, we injected `boolean`, `String`, and `Integer` values, and the Spring container knew to take the text value from the configuration file and convert it to the property type defined in the bean type definition. The Spring container knows how to do this for all primitive types and their reference wrapper types. `String` values, Booleans, and numeric types are supported by default. Just keep in mind that for Booleans and decimal types, however, the syntax typical for these types must be respected[28]; otherwise, the default Spring conversion will not work.

Spring also knows to automatically convert `Date` values before injection if the syntax matches any date pattern with "/" separators (e.g., dd/MM/yyyy, yyyy/MM/dd, and dd/yyyy/MM). Take care, because even invalid numerical combinations are accepted and converted (e.g., 25/2433/23), which leads to unexpected results.

To support any other type, the Spring container must be told how to convert the value of the attribute to the type that the constructor, setter, or instance variable requires. The most common method until Spring 3 was to use a property editor. In Spring 3 a `core.convert` package was introduced that provides a general type conversion system that defines an SPI [29] that you have to implement to provide the desired conversion behavior. The interface is named `Converter<S,T>` and its code is depicted in the following snippet.[30]

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
    T convert(S source);
}
```

This is a Spring concept that describes a component used to handle the transformation between any two types of objects. To make such a conversion possible, you have to define an implementation for the `org.springframework.core.convert.converter.Converter` interface and register it in the Spring context. Out of the box, Spring provides a number of `Converter` implementations for commonly used types, in the `core.convert.support` package.

[28]For example, values "0" and "1" cannot be used as values for a boolean field.

[29]An application programming interface (API) describes what a class/method does; a Service Provider Interface (SPI) describes what you need to extend and implement.

[30]Full code and copyright notices on GitHub: https://github.com/spring-projects/spring-framework/blob/master/spring-core/src/main/java/org/springframework/core/convert/converter/Converter.java

Let's assume we want to define a Person bean as follows.

```
package com.apress.cems.beans.scalars;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.time.LocalDate;

@Component
public class PersonBean implements Creature {

    private LocalDate birthDate;
    private String name;

    @Autowired
    public PersonBean(@Value("1977-10-16") LocalDate birthDate,
        @Value("John Mayer") String name) {
        this.birthDate = birthDate;
        this.name = name;
    }

    @Override
    public LocalDate getBirthDate() {
        return birthDate;
    }

    @Override
    public void setBirthDate(LocalDate birthDate) {
        this.birthDate = birthDate;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Spring provides more than one way to tell the Spring IoC container how the value `1977-10-16` can be converted to a `LocalDate`. An implementation of the JDK `java.beans.PropertyEditor` provided out of the box by Spring, the `org.springframework.beans.propertyeditors.CustomDateEditor` was used in the previous edition of this book, when the `java.util.Date` was used for the `birthDate` field. This is how you use this class: you create an instance of `CustomDateEditor` and provide as a parameter a `SimpleDateFormat` instance, register the `CustomDateEditor` instance with a `PropertyEditorRegistrar` that is then used to create a bean of type `CustomEditorConfigurer`.[31]

But since the `Converter` SPI was mentioned, and our type is the new and improved JDK `LocalDate`, an implementation is required.

In order for our converter to be used at runtime the following things must be done.

- Define a class that implements the `org.springframework.core.convert.converter.Converter` and implement the `convert` method to provide a way to convert `String` values to `LocalDate` values.

```
package com.apress.cems.beans.scalars;

import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

@Component
public class StringToLocalDate implements Converter<String,
LocalDate> {

    private DateTimeFormatter formatter =
           DateTimeFormatter.ofPattern("yyyy-MM-dd");

    @Override
    public LocalDate convert(String source) {
        return LocalDate.parse(source, formatter);
    }
}
```

[31] If you are curious about this implementation just clone this GitHub repository https://github.com/Apress/pivotal-certified-pro-spring-dev-exam and take a look at the '02-ps-container-01-solution' project.

- Add a bean declaration of type `org.springframework.core.convert.ConversionService`, named `conversionService`. The name is mandatory, as this is an infrastructure bean, and add the `StringToLocalDate` converter to its list of supported converters. When using XML, the configuration is pretty straightforward, even if to create the bean a factory bean is used. Spring supports instantiating beans using factory beans as well.

```
<bean id="conversionService"
    class="org.springframework.context.support.
    ConversionServiceFactoryBean">
    <property name="converters">
        <set>
            <bean class="com.apress.cems.beans.scalars.
            StringToLocalDate"/>
        </set>
     </property>
</bean>
```

When using Java configuration, the bean declaration looks a little different from what you've seen so far in the chapter. The `configurationService` bean is not directly instantiated, but created by a factory bean of type `ConversionServiceFactoryBean`. The `getObject()` method is called explicitly to obtain an instance of type `org.springframework.core.convert.ConversionService`.

```
package com.apress.cems.beans.scalars;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support.ConversionServiceFactoryBean;
import org.springframework.core.convert.ConversionService;
```

```
import java.util.Set;

@Configuration
@ComponentScan(basePackages = {"com.apress.cems.beans.scalars"} )
public class AppCfg {

    @Autowired
    StringToLocalDate stringToLocalDateConverter;

    @Bean
    ConversionService conversionService(
              ConversionServiceFactoryBean factory){
        return factory.getObject();
    }

    @Bean
    ConversionServiceFactoryBean conversionServiceFactoryBean() {
        ConversionServiceFactoryBean factory = new
        ConversionServiceFactoryBean();
        factory.setConverters(Set.of(stringToLocalDateConverter,
        stringToDate));
        return factory;
    }
}
```

Because of its syntax and its definition, `@Autowired` cannot be used to autowire primitive values, which is quite logical, considering that there is an annotation named `@Value` that specializes in this exactly. The `@Value` annotation can insert scalar values or used with placeholders and SpEL[32] to provide flexibility in configuring a bean.

SpEL is the Spring Expression language. It is quite powerful, since it supports querying and manipulating an object graph at runtime. This means the language supports getting and setting property values, method invocation, and usage of arithmetic and logic operators and bean retrieval from the Spring IoC container. The SpEL is

[32]Spring Expression Language

inspired from WebFlow EL,[33] a superset of Unified EL,[34] and it provides considerable functionality, such as

- method invocation
- access to properties, indexed collections
- collection filtering
- Boolean and relational operators

and many more.[35]

There are several extensions of SpEL (OGNL, MVEL, and JBoss EL), but the best part is that it is not directly tied to Spring and can be used independently.

! An example bean with fields of various types was provided in `chapter02/beans`. The class is called `com.apress.cems.beans.scalars.MultipleTypesBean`, and it can be tested by running the test class called `com.apress.cems.beans.scalars.AppConvertersCfgTest`. The configuration class for this example is named `AppConvertersCfg` and is just an empty configuration class annotated with `@Configuration` and `@ComponentScan`.

```
package com.apress.cems.beans.scalars;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.util.Date;
```

[33] An expression language used to configure web flows: http://docs.spring.io/spring-webflow/docs/current/reference/html/el.html.

[34] Unified EL is the Java expression language used to add logic in JSP pages: https://docs.oracle.com/javaee/6/tutorial/doc/bnahq.html.

[35] The full list of capabilities is not in the scope of this book. If you are interested in SpEL, the official documentation is the best resource: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html.

```
@Component
public class MultipleTypesBean {
    private int noOne;
    private Integer noTwo;

    private long longOne;
    private Long longTwo;

    private float floatOne;
    private Float floatTwo;

    private double doubleOne;
    private Double doubleTwo;

    private boolean boolOne;
    private Boolean boolTwo;

    private char charOne;
    private Character charTwo;

    private Date date;

    @Autowired void setNoOne(@Value("1") int noOne) {
        this.noOne = noOne;
    }

    @Autowired void setNoTwo(@Value("2")Integer noTwo) {
        this.noTwo = noTwo;
    }

    @Autowired void setFloatOne(@Value("5.0")float floatOne) {
        this.floatOne = floatOne;
    }

    @Autowired void setFloatTwo(@Value("6.0")Float floatTwo) {
        this.floatTwo = floatTwo;
    }

    @Autowired void setDoubleOne(@Value("7.0")double doubleOne) {
        this.doubleOne = doubleOne;
    }
```

```
    @Autowired void setDoubleTwo(@Value("8.0")Double doubleTwo) {
        this.doubleTwo = doubleTwo;
    }

    @Autowired void setLongOne(@Value("3")long longOne) {
        this.longOne = longOne;
    }

    @Autowired void setLongTwo(@Value("4")Long longTwo) {
        this.longTwo = longTwo;
    }

    @Autowired void setBoolOne(@Value("true")boolean boolOne) {
        this.boolOne = boolOne;
    }

    @Autowired void setBoolTwo(@Value("false")Boolean boolTwo) {
        this.boolTwo = boolTwo;
    }

    @Autowired void setCharOne(@Value("1")char charOne) {
        this.charOne = charOne;
    }

    @Autowired void setCharTwo(@Value("A")Character charTwo) {
        this.charTwo = charTwo;
    }

    @Autowired void setDate(@Value("1977-10-16") Date date) {
        this.date = date;
    }
...
}
```

Run the test in debug mode, as depicted in Figure 2-18, and set a breakpoint on line 55, right after the `mtb` bean is returned by the application context. In the Debug console (visible at the bottom of the figure), you can see the values for all the fields of the `mtBean`.

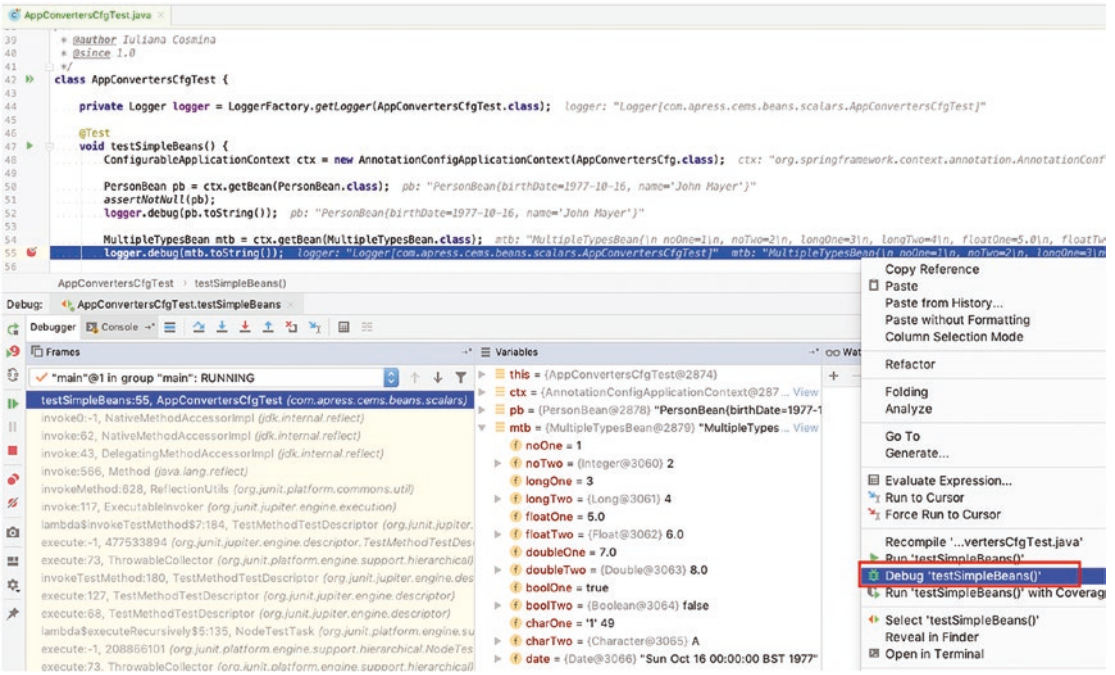

Figure 2-18. *Running a test in debug mode to inspect the fields of a bean of type MultipleTypesBean*

The fields with names postfixed with One are primitives, because Spring knows out of the box how to convert `String` values to primitives, when the text represent valid values; the others are objects. Notice how they were all initialized correctly, and the values in the bean definition were converted to the appropriate types.

Collections are a type of object often used. To make using them with beans easier, the Spring development team created the `util` namespace. But since the focus of this book is not on XML configuration, let's see how we can use collections as beans using Java configuration. Consider the `CollectionHolder` class defined in the following code snippet.

```
package com.apress.cems.beans.scalars;

import com.apress.cems.beans.ci.SimpleBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
import java.util.List;
import java.util.Map;
import java.util.Set;

public class EmptyCollectionHolder {

    private List<SimpleBean> simpleBeanList;
    private Set<SimpleBean> simpleBeanSet;
    private Map<String, SimpleBean> simpleBeanMap;

    public void setSimpleBeanList(List<SimpleBean> simpleBeanList) {
        this.simpleBeanList = simpleBeanList;
    }

    public void setSimpleBeanSet(Set<SimpleBean> simpleBeanSet) {
        this.simpleBeanSet = simpleBeanSet;
    }

    public void setSimpleBeanMap(Map<String, SimpleBean> simpleBeanMap) {
        this.simpleBeanMap = simpleBeanMap;
    }

     /**
     * This method was implemented just to verify the collections injected
     *  into this bean
     */
    @Override
    public String toString() {
        return "CollectionHolder{" +
                "simpleBeanList=" + simpleBeanList.size() +
                ", simpleBeanSet=" + simpleBeanSet.size()  +
                ", simpleBeanMap=" + simpleBeanMap.size()  +
                '}';
    }
}
```

The state of the collectionHolder bean will be tested using the AppConvertersCfgTest:

```
package com.apress.cems.beans.scalars;

import org.junit.jupiter.api.Test;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

import static org.junit.jupiter.api.Assertions.assertNotNull;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class AppConvertersCfgTest {

    private Logger logger = LoggerFactory.getLogger(AppConvertersCfgTest.
    class);

    @Test
    void testSimpleBeans() {
        ConfigurableApplicationContext ctx =
        new AnnotationConfigApplicationContext(AppConvertersCfg.class);

        EmptyCollectionHolder collectionHolder =
                    ctx.getBean(EmptyCollectionHolder.class);
        (*) logger.debug(collectionHolder.toString());

        ctx.close();
    }
```

The collections to be injected can be created just like normal beans by annotating methods in configuration classes with @Bean. This will result in beans of collection types being created and injected where necessary. Next, you can see a few examples of collection beans being declared.

- inject empty collections

```
package com.apress.cems.beans.scalars;

import com.apress.cems.beans.ci.SimpleBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import java.util.*;

@Configuration
public class AppConvertersCfg {
    ...

    @Bean
    List<SimpleBean>  simpleBeanList(){
        return new ArrayList<>();
    }

    @Bean
    Set<SimpleBean>  simpleBeanSet(){
        return new HashSet<>();
    }

    @Bean
    Map<String, SimpleBean> simpleBeanMap(){
        return new HashMap<>();
    }
}
```

If you run the `AppConvertersCfgTest` class aside from the test passing, the `logger.debug(..)` statement in the line marked with (*) in the test class will print the following: `CollectionHolder{simpleBeanList=0, simpleBeanSet=0, simpleBeanMap=0}`

We injected empty collections here because their content is not really important at the moment. In XML configuration injecting empty collections is even easier and readable.

```
 <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
       beans
        http://www.springframework.org/schema/beans/spring-beans.
        xsd">
```

```
  <bean id="collectionHolder"
         class="com.ps.beans.others.EmptyCollectionHolder">
        <property name="simpleBeanList">
            <list/>
        </property>

        <property name="simpleBeanSet">
            <set/>
        </property>

        <property name="simpleBeanMap">
            <map/>
        </property>
    </bean>
</beans>
```

If you are wondering why the `util` namespace is not used in the previous example, this is because the collections are declared as values for the properties of the `collectionHolder` bean and this can be done by using configuration elements like the ones underlined which are declared in the `beans` namespace.

- inject a property of type List with `SimpleBean` elements. Using Java configuration, the only thing that changes is the declaration of the collection beans, instead of creating an empty collection, we just have to make sure to add some instances in it. We can even add `null`, although best practices say that you should avoid using `null` objects as much as possible.

```
 package com.apress.cems.beans.scalars;

import com.apress.cems.beans.ci.SimpleBean;
import com.apress.cems.beans.ci.SimpleBeanImpl;
import java.util.*;
...

@Configuration
public class AppConvertersCfg {
    ...
```

```
    @Bean
      SimpleBean simpleBean(){
          return new SimpleBeanImpl();
      }

      @Bean
      List<SimpleBean>  simpleBeanList(){
        return List.of(simpleBean());
      }

      @Bean
      Set<SimpleBean>  simpleBeanSet(){
          return Set.of(simpleBean());
      }

      @Bean
      Map<String, SimpleBean> simpleBeanMap(){
          return Map.of("simpleBean", simpleBean());
      }
}
```

We can either add existing beans as in the previous example, or create them on the spot, in the `simpleBeanList`. Or both, it really does not matter. Everything mentioned can be done in XML too, and working code can be found in the repository for the previous edition of the book. If you do not want to scan the previous GitHub repo, the following is an XML configuration sample in which the list to be injected in the `collectionHolder` bean is populated with an existing bean named `simpleBean`, an instance of `SimpleBeanImpl` created on the spot, and a `null` element, configured using a `<null />` element. Pretty intuitive, right?

```
<bean id="simpleBean" class="com.apress.cems.beans.ci.SimpleBeanImpl"/>

<bean id="collectionHolder"
    class="com.apress.cems.beans.scalars.CollectionHolder">
       <property name="simpleBeanList">
           <list>
               <ref bean="simpleBean"/>
               <bean class="com.apress.cems.beans.ci.SimpleBeanImpl"/>
```

```
                <null/>
            </list>
        </property>
</bean>
```

The `logger.debug(..)` statement in the line marked with (*) in the test class will print something similar to the following sample.

```
CollectionHolder{simpleBeanList=3, ... }
```

And since I mentioned SpEL, the `@Value` annotation can also be used to inject values from other beans by making use of the SpEL language in the following case of a `Properties` object.

```
@Configuration
public class DataSourceConfig1 {

    @Bean
    public Properties dbProps(){
        Properties p = new Properties();
        p.setProperty("driverClassName", "org.h2.Driver");
        p.setProperty("url", "jdbc:h2:~/sample");
        p.setProperty("username", "sample");
        p.setProperty("password", "sample");
        return p;
    }

    @Bean
    public DataSource dataSource(
            @Value("#{dbProps.driverClassName}")String driverClassName,
            @Value("#{dbProps.url}")String url,
            @Value("#{dbProps.username}")String username,
            @Value("#{dbProps.password}")String password) throws
            SQLException {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driverClassName);
        ds.setUrl(url);
```

```
        ds.setUsername(username);
        ds.setPassword(password);
        return ds;
    }
}
```

The `dbProps` is an object that was instantiated and initialized, and it is treated as a bean in the application because of the `@Bean` annotation. There is no equivalent for `@Value` in any JSR.

Using Bean Factories

In the previous section, when we created a bean of type `ConversionService` we did so by using a bean of type `ConversionServiceFactoryBean`. This is an approach that makes use of the *Factory Pattern*. A pattern in the software world is a list of steps, an algorithm, or a model for a solution that solves a class of problems.[36] Alongside Singleton, the *Factory pattern*[37] is one of the most used patterns for writing Java applications. The Factory pattern is a creation pattern that provides a practical way to create objects of various types that implement the same interface by hiding the creation logic and exposing only a minimal API. It comes in two flavors: the *Factory Method pattern* and the *Abstract Factory pattern*. Spring supports creating beans using implementations following both patterns by using the `@Bean` annotation. The same annotation can be used when creating a bean of a singleton class (a class that can only be instantiated once). Before the introduction of this annotation, different attributes were required in XML to create beans.

Let's create a simple singleton class[38] and use the `@Bean` annotation to declare a bean of its type. We'll call the class `SimpleSingleton` to really make it obvious what we're doing.

[36]In the Java world there is a book considered the bible of patterns named *Design Patterns: Elements of Reusable Object-Oriented Software* written by four authors known as the *Gang of Four*; it describes various development techniques and covers 23 object-oriented java patterns.

[37]More information about this pattern can easily be found on the Internet using a simple search on Google, but here is a quick good source: https://en.wikipedia.org/wiki/Factory_method_pattern.

[38]The overview of the Singleton pattern

```
 package com.apress.cems.beans.misc;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SimpleSingleton {
    private Logger logger = LoggerFactory.getLogger(SimpleSingleton.class);

    private static SimpleSingleton instance = new SimpleSingleton();

    private SimpleSingleton() {
        logger.info(">> Creating single instance.");
    }

    public static synchronized SimpleSingleton getInstance(){
        return instance;
    }
}
```

The use of synchronized is the worst approach to apply the Singleton pattern, indicate that for academic or simplicity purposes you are using this approach.

! In the previous example, the keyword `synchronized` is used on the `getInstance()` method. This is to respect the Singleton design pattern, which requires to ensure that even in a multithreaded environment a single instance is created. This causes a bottleneck in accessing this instance, but is a drawback it can be lived with. But in a Spring environment for using the `SimpleSingleton` class as a template for a bean factory synchronization is not necessary.

This class is not annotated with `@Component` because we want more control over the creation of the bean. The class declares a private static field of the class type that is instantiated on the spot, and it is returned by the `getInstance()` method.

The `@Bean` annotation is used on a method creating and returning a bean that is declared inside a configuration class annotated with `@Configuration`. The class and bean declaration are depicted in the next code snippet and you can see that the bean is created by calling the `getInstance()` method.

```
package com.apress.cems.beans.misc;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MiscAppCfg {

    @Bean
    SimpleSingleton simpleSingleton(){
        return SimpleSingleton.getInstance();
    }
}
```

And this is all there is. And if you want to test that only one bean is created, you can run the `com.apress.cems.beans.misc.MiscAppCfgTest` class. This class retrieves two beans of type `SimpleSingleton` using two different methods and testing that the instances retrieved are equal.

```
package com.apress.cems.beans.misc;

import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplicationContext;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

public class MiscAppCfgTest {
    private Logger logger = LoggerFactory.getLogger(MiscAppCfgTest.class);

    @Test
    void testSimpleBeans() {
        ConfigurableApplicationContext ctx =
            new AnnotationConfigApplicationContext(MiscAppCfg.class);

        SimpleSingleton simpleSingleton = ctx.getBean(SimpleSingleton.class);
        assertNotNull(simpleSingleton);
```

```
        SimpleSingleton simpleSingleton2 =
             ctx.getBean("simpleSingleton", SimpleSingleton.class);
        assertNotNull(simpleSingleton2);

        assertEquals(simpleSingleton, simpleSingleton2);
        ctx.close();
    }
}
```

Now that this is covered, let's jump to the next one. Let's implement a class that uses a factory method to create a few types of beans. To show you how the factory method is used, let's create a method that depending of a text value, will return an instance of a class implementing the interface `com.apress.cems.beans.misc.TaxFormula` that groups together a hierarchy of classes that can calculate taxes. The hierarchy and the class containing the factory method are depicted in Figure 2-19.

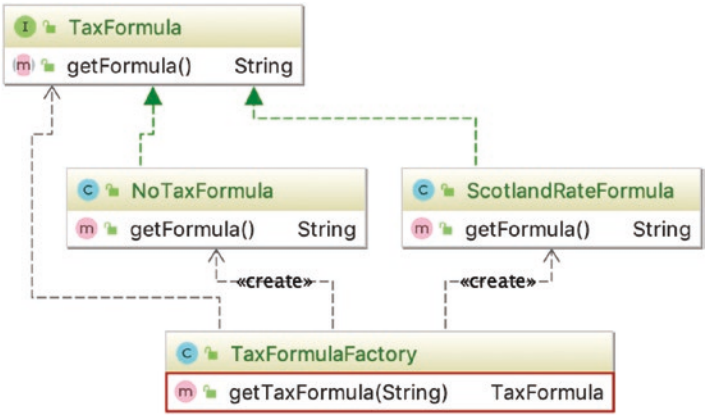

Figure 2-19. *Class hierarchy making use of the Factory Method pattern*

The implementations are naive, and the `TaxFormula` interface and the `TaxFormulaFactory` class are the only one relevant for this example.

```
// TaxFormula.java
package com.apress.cems.beans.misc;

public interface TaxFormula {
     String getFormula();
}
```

```
// TaxFormulaFactory.java
package com.apress.cems.beans.misc;

public class TaxFormulaFactory {

    public TaxFormula getTaxFormula(final String taxPlanCode){
        if(taxPlanCode == null){
            return null;
        }

        switch (taxPlanCode) {
            case "S":
                return new ScotlandRateFormula();
            case "NT":
                return new NoTaxFormula();
        }
        return null;
    }
}
```

The `getTaxFormula()`method creates beans of either a `ScotlandRateFormula` or `NoTaxFormula` type. To use it, we have to first instantiate the `TaxFormulaFactory`. The declaration of a bean using an instance of the previous class is depicted in the following code snippet.

```
 package com.apress.cems.beans.misc;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MiscAppCfg {

    @Bean
    SimpleSingleton simpleSingleton(){
        return SimpleSingleton.getInstance();
    }
```

```
    @Bean
    TaxFormula taxScotlandFormula(){
        return new TaxFormulaFactory().getTaxFormula("S");
    }
}
```

In the previous code snippet you might have noticed that the bean type is declared to be `TaxFormula`. This allows future modifications to the body of the `taxScotlandFormula()` method and return a different implementation, without causing compile time errors in the code using the created beans. Interfaces are contracts, a description of a set of methods and classes implementing them should provide a concrete behavior for them. Using interface types for references is practical for larger systems, because no matter how much the system evolves as long as the interface contract is respected, dependencies can be swapped at runtime without fear of the system malfunctioning. The previous example is quite succinct, and the factory class is instantiated in place inside the method declaring the bean.

Another version of implementing this is to declare the factory bean and use it whenever necessary. Although it seems somewhat redundant, creating beans in this way might be useful when one is using third-party libraries that only allow creating objects using a factory class. And this method connects existing components to provide a certain behavior, such as helping Spring to convert `String` values to `Date` objects (remember how the `conversionService` bean was created) using a format pattern different from the default one.

When writing your own bean factories, you might want to follow the guideline provided by Spring and implementing the `org.springframework.beans.factory.FactoryBean<T>` interface. This interface is used by many Spring implementations to simplify configuration, including the `ConversionServiceFactoryBean` used earlier in the chapter. Even if implementing your factories this way ties your code to Spring components (which is usually not recommended), this method is practical when XML configuration is needed because it simplifies it a lot. The class named `TaxFormulaFactoryBean` creates a singleton bean of type `TaxFormula` and implements the Spring interface. The class name follows the recommended Spring name template: `[ClassName]FactoryBean`.

```
  package com.apress.cems.beans.xml.misc;

import com.apress.cems.beans.misc.ScotlandRateFormula;
import com.apress.cems.beans.misc.TaxFormula;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.FactoryBean;

public class TaxFormulaFactoryBean implements FactoryBean<TaxFormula> {

    private Logger logger = LoggerFactory.getLogger(TaxFormulaFactoryBean.
    class);
    private TaxFormula taxFormula = new ScotlandRateFormula();

    public TaxFormulaFactoryBean() {
        logger.info(">> Look ma, no definition!");
    }

    @Override getObject() throws Exception {
        return this.taxFormula;
    }

    @Override
    public Class<?> getObjectType() {
        return ScotlandRateFormula.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

If the previous class did not implement the `FactoryBean<T>` interface, but just had a method named `getObject()` that returns the instance being created, the XML declaration would look like the following.

```
    <bean id="taxFormulaFactoryBean"
           class="com.apress.cems.beans.xml.misc.TaxFormulaFactoryBean" />
    <bean id="taxScotlandFormula" factory-bean="taxFormulaFactoryBean" />
```

Yes, as you probably suspected, the method name is important, and it can be recognized by Spring as a factory method even if it is not part of class implementing `FactoryBean`. If the method is named differently, let's say `getInstance()`, then the XML configuration has to be adjusted to tell Spring the name of the factory method. And this can be done by adding the `factory-method` attribute and set the method name as its value.

```
<bean id="taxFormulaFactoryBean"
       class="com.apress.cems.beans.xml.misc.TaxFormulaFactoryBean" />
<bean id="taxScotlandFormula" factory-bean="taxFormulaFactoryBean"
   factory-method="getInstance" />
```

But because a Spring interface is used, the configuration can be reduced to the following.

```
<bean id="scotlandTaxFactory" class="com.apress.cems.beans.xml.misc.
TaxFormulaFactoryBean" />
```

And although the `class` attribute value is `com.apress.cems.beans.xml.misc.TaxFormulaFactoryBean<T>`, the bean instance named `scotlandTaxFactory` will be of type `com.apress.cems.beans.misc.ScotlandRateFormula`.

If you are curious about the Spring classes that implement `org.springframework.beans.factory.FactoryBean`, IntelliJ IDEA can help you there. The class depicted in the previous code snippet is part of `02-ps-container-01-practice`. Just open that class, click the interface name, and press CTRL (Command in macOS)+ALT+B, and a list of classes implementing it will be displayed, as depicted in Figure 2-20.

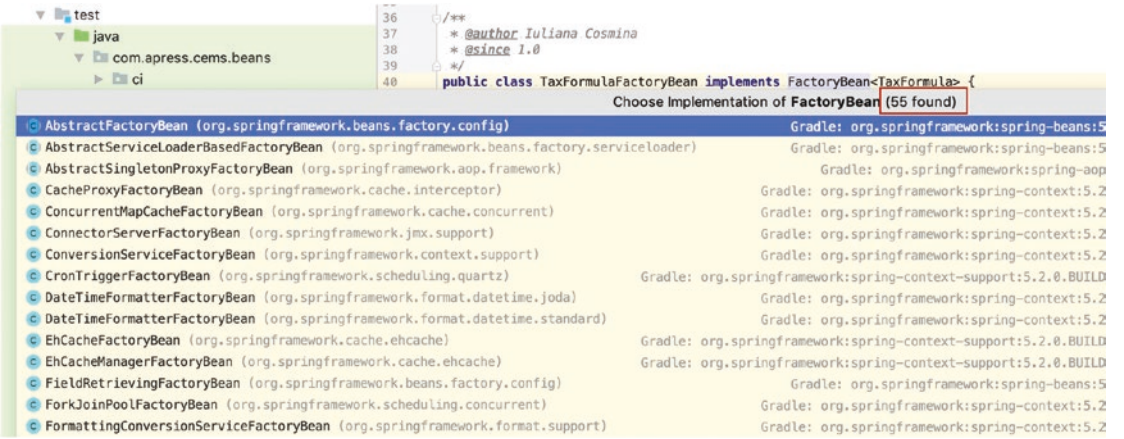

Figure 2-20. *Classes implementing the FactoryBean interface*

Spring factory bean classes provide assistance for configuring data access using Hibernate and JPA, for handling transactions, for configuring caching with EhCache, and so on. A few of them will be used in the code samples for this book, so you will have occasion to see them in action.

More About Autowiring

The `@Autowired` annotation is equivalent to JSR 330's `@Inject`. The infrastructure bean that is responsible for detecting and processing these annotations is named `org.springframework.context.annotation.internalAutowiredAnnotationProcessor` of type `AutowiredAnnotationBeanPostProcessor`[39] and implements the `org.springframework.beans.factory.config.BeanPostProcessor` interface, which means this type of bean modifies bean instances, which should be obvious since this annotation declares the dependencies to be injected.

The `@Autowired` annotation requires the dependency to be mandatory, but this behavior can be changed by setting the `required` attribute to `false`.

```
@Autowired(required=false)
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}
```

In the previous case, if a bean of type `DataSource` is not found within the context the value that is injected is `null`. The `@Autowired` annotation works on methods. If Spring can identify the proper bean, it will autowire it. This is useful for development of special Spring configuration classes that have methods that are called by Spring directly and have parameters that have to be configured. The most common example is the setting method global security using classes annotated with `@EnableGlobalMethodSecurity`:

[39]The name of this bean and its purpose and a few others are part of the *org.springframework.context.annotation.AnnotationConfigUtils* class. You can see the code here https://github.com/spring-projects/spring-framework/blob/3a0f309e2c9fdbbf7fb2d348be861528177f8555/spring-context/src/main/java/org/springframework/context/annotation/AnnotationConfigUtils.java

```
@Configuration
@EnableGlobalMethodSecurity
public class MethodSecurityConfig {

// method called by Spring
  @Autowired
  public void registerGlobal(AuthenticationManagerBuilder auth)
      throws Exception {
    auth
        .inMemoryAuthentication()
          .withUser("john").password("test").roles("USER").and()
          .withUser("admin").password("admin").roles("USER", "ADMIN");
    }

  @Bean
  public MethodSecurityService methodSecurityService() {
    return new MethodSecurityServiceImpl()
  }
}
```

A more detailed version of this code snippet is explained in the "Spring Security" section in Chapter 6.

A strong feature for `@Autowired` added in Spring 4.x is the possibility to use generic types as qualifiers. This is useful when you have classes that are organized in a hierarchy and they all inherit a certain class that is generic, like the repositories in the project attached to the book, all of which extend `JdbcAbstractRepo<T>`. Let's see how this feature can be used.

```
// repository classes extending JdbcAbstractRepo
@Component
public class JdbcCriminalCaseextends JdbcAbstractRepo<CriminalCase>
  implements CriminalCaseRepo {...}

@Component
public class JdbcDetectiveRepo extends JdbcAbstractRepo<Detective>
   implements DetectiveRepo {...}
```

```
@Configuration
@ComponentScan(basePackages =  {"com.apress.cems.repos"})
public class RepositoryConfig {
        ...
}
 // Test class
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {RepositoryConfig})
public class GenericQualifierTest {

    @Autowired
    JdbcAbstractRepo<CriminalCase> criminalCaseRepo;

    @Autowired
    JdbcAbstractRepo<Detective> detectiveRepo;
....
}
```

This helps a lot, because in related classes, the `Qualifier` annotation is no longer needed to name different beans of related types to make sure that Spring does not get confused. However, if Spring cannot use generic types as qualifiers because this type is the same for two related implementations, the use of `@Qualifer` is necessary; otherwise, the application context cannot be created.

```
// repository classes extending JdbcAbstractRepo<CriminalCase>
@Component
public class JdbcCriminalCaseRepo extends JdbcAbstractRepo<CriminalCase>
  implements CriminalCaseRepo {...}

@Component
public class JPACriminalCaseRepo extends JdbcAbstractRepo<CriminalCase>
   implements CriminalCaseRepo {...}

@Configuration
@ComponentScan(basePackages =  {"com.apress.cems.repos"})
public class RepositoryConfig {
        ...
}
```

```
 // Test class
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {RepoConfig.class})
public class TestRepoCfg {

    @Qualifier("jdbcCriminalCaseRepo")
    @Autowired
    CriminalCaseRepo jdbcRepo;

    @Qualifier("JPACriminalCaseRepo")
    @Autowired
    CriminalCaseRepo jpaRepo;

    @Test
    void allGoodTest(){
        assertNotEquals(jdbcRepo,jpaRepo);
    }
}
```

In this example, we specified in the test class exactly which bean we want autowired by annotating the fields with `@Qualifier` and providing the bean name as an argument for it. Without this annotation, Spring cannot figure out which bean we want to inject, since both available bean types are subtypes of `CriminalCaseRepo`.

The JSR 250 and JSR 330 annotations are supported and can be used alongside Spring annotations, but they have quite a few common functionalities. In Table 2-3, you can see the correspondences and the differences between them.

Table 2-3. *Autowiring Annotations*

Spring	JSR	Comment
`@Component`	`@Named`	`@Named` can be used instead of all stereotype annotations except `@Configuration`
`@Qualifier`	`@Qualifier`	JSR `Qualifier` is a marker annotation used to identify qualifier annotations, like `@Named`, for example
`@Autowired`	`@Inject`	`@Inject` may apply to static as well as instance members
`@Autowired + @Qualifier`	`@Resource(name= "beanName")`	`@Resource` is useful because replaces two annotations.

Another interesting annotation is `@Lazy`, which starting with Spring 4.x can be used on injection points (wherever `@Autowired` is used) too. This annotation postpones the creation of a bean until it is first accessed by adding this annotation to the bean definition. So yes, as with `@Scope`, `@Lazy` can be used with a `@Component` or `@Bean` annotation. This is useful when the dependency is a huge object, and you do not want to keep the memory occupied with this object until it is really needed, but since it allows creating an application context without the full dependency tree being created when the application starts, it leaves room for configuration errors, so use it wisely.

```
@Component
@Lazy
public class SimpleBean { ... }

// or on a @Bean
@Configuration
public class RequestRepoConfig {

    @Lazy
    @Bean
    public RequestRepo anotherRepo(){
        return new JdbcRequestRepo();
    }
}
```

```
// on injection point
@Repository
public class JdbcPetRepo  extends  JdbcAbstractRepo<Pet>
implements PetRepo {
    ...
    @Lazy
    @Autowired(required=false)
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

In the previous example, you can see `@Lazy` used with `Autowired(required = false)`. This is the only correct usage that allows the dependency to be lazily initialized. Without `Autowired(required = false)`, the Spring IoC container will do its normal job and inject dependencies when the bean is created and ignoring the presence of `@Lazy`.

Aside from all the annotations discussed so far, developers can write their own annotations. You've already seen an example of a `@Scope` specialization earlier in the chapter. Many of the annotations provided by Spring can be used as meta-annotations in your own code. You can view a meta-annotation as a super class. A meta-annotation can annotate another annotation. All stereotype specializations are annotated with `@Component`, for example. Meta-annotations can be composed to obtain other annotations. For example, let's create a `@CustomTx` annotation that will be used to mark service beans that will use a different transaction.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.transaction.annotation.Transactional;

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("customTransactionManager", timeout="90")
public @interface CustomTx {
    boolean readOnly() default false;
}
```

Spring configuration using annotations is really powerful and flexible. In this book, Java configuration and annotations are covered together, because this is the way they are commonly used in practice. There definitely are old-style developers who still prefer XML and more grounded developers who like to *mix and match*. Configuration using annotations is practical, because it can help you reduce the number of resource files in the project, but the downside is that it is scattered all over the code. But being linked to the code, refactoring is a process that becomes possible without the torture of searching bean definitions in XML files, although smart editors help a lot with that these days. The annotations configuration is more appropriate for beans that are frequently changing: custom business beans such as services, repositories, and controllers. Imagine adding a new parameter for a constructor and then going hunting beans in the XML files, so the definition can be updated. Java configuration annotations should be used to configure infrastructure beans (data sources, persistence units, etc.).

There are advantages to using XML too. The main advantage is that the configuration is centralized in a few XML files and is decoupled from the code. You could just modify the XML configuration files and repackage the application without a full rebuild being necessary. But Java configuration can be done in a smart way too; nothing stops you from declaring all the configuration in a set of packages that can be wrapped together in a configuration library. When you change configuration details, you only need to rebuild those packages, not the full project. Then, XML is widely supported and known, and there are more than a few legacy applications out there. Ultimately, it is only the context that decides the most appropriate solution. If you are a Spring expert starting to work on your own startup application, you will probably go with Java configuration and annotations. And probably Spring Boot, which will make your work much easier by providing super-meta-annotations to configure much of the infrastructure needed for the project. If you are working on a big project with legacy code, you will probably have to deal with some XML configuration here and there. Whatever the case, respect good practices and read the manual, and you should be fine.

Using Multiple Configuration Classes

In a Spring test environment, the spring-test module provides the `@ContextConfiguration` to bootstrap a test environment using one or multiple configuration resources. This is an alternative to creating an application context by directly instantiating a specific context class. The following code depicts this scenario, and it provides a different way to test spring

applications by delegating the responsibility of creating the application context to the Spring Test Framework. (Chapter 3 is fully dedicated to testing Spring applications.)

```
package com.apress.cems.config;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringRunner;

import javax.sql.DataSource;

import static org.junit.jupiter.api.Assertions.assertNotNull;

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {TestDataSourceConfig.class})
public class BootstrapDatasourceTest {

    @Autowired
    DataSource dataSource;

    @Test
    public void testBoot() {
        assertNotNull(dataSource);
    }
}
```

Let's declare another configuration class that will pick up bean declarations of repositories using the `dataSource` bean referred previously.

```
 package com.apress.cems.config;

import com.apress.cems.pojos.repos.DetectiveRepo;
import com.apress.cems.repos.JdbcDetectiveRepo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
```

```
import javax.sql.DataSource;

@Configuration
@ComponentScan(basePackages =  {"com.apress.cems.repos"})
@Import(TestDataSourceConfig.class)
public class RepositoryConfig {

    @Autowired
    DataSource dataSource;

    @Bean
    DetectiveRepo detectiveRepo(){
        return new JdbcDetectiveRepo(dataSource);
    }
}
```

The repository implementations are not really important, but what is important is the fact that the `JdbcDetectiveRepo` class was not configured as a bean. A bean of type JdbcDetectiveRepo is declared using the `@Bean` annotation in the RepositoryConfig configuration class. This style of configuration gives me the opportunity to show you how to declare beans that depend on this bean in a different configuration class.

The `@Import` annotation imports the configuration from another class into the class annotated with it.

The repository classes identified using component scanning by the `RepositoryConfig` class in the previous example, have been declared as beans by annotating them with `@Repository` which is a specialization of `@Component`. This decision was made to make the purpose of these bean obvious. As example, this is the `JdbcEvidenceRepo` class and annotations that configure it as a bean declaration.

```
package com.apress.cems.repos;

...
import com.apress.cems.pojos.repos.EvidenceRepo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
```

```
@Repository
public class JdbcEvidenceRepo extends
          JdbcAbstractRepo<Evidence> implements EvidenceRepo {

    public JdbcEvidenceRepo(){
    }

    @Autowired
    public JdbcEvidenceRepo(DataSource dataSource) {
        super(dataSource);
    }

    ...
}
```

The previous configuration can be tested by writing a typical Spring test class that runs in a test context based on the `RepositoryConfig` configuration class.

```
package com.apress.cems.config;

import com.apress.cems.pojos.repos.DetectiveRepo;
import com.apress.cems.pojos.repos.EvidenceRepo;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringRunner;

import static org.junit.Assert.assertNotNull;

@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {RepositoryConfig.class})
public class MultipleResourcesTest {

    @Autowired
    EvidenceRepo evidenceRepo;

    @Autowired
    DetectiveRepo detectiveRepo;
```

```
    @Test
    public void testInjectedBeans(){
        assertNotNull(evidenceRepo);
        assertNotNull(detectiveRepo);
    }
}
```

When run, the previous test passes, because both `evidenceRepo` and `detectiveRepo` beans are created and were injected correctly in the test context. This works fine, because the `@Import(TestDataSourceConfig.class)` annotation was used to decorate the `RepositoryConfig` class. Without this annotation on the `TestDataSourceConfig`, both classes have to be used as arguments for the `classes` attribute in the `@ContextConfiguration` annotation.

The following test class configuration depicts exactly this scenario, when `@Import|` is not used on `TestDataSourceConfig`.

```
package com.apress.cems.config;

import com.apress.cems.pojos.repos.DetectiveRepo;
import com.apress.cems.pojos.repos.EvidenceRepo;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringRunner;

import static org.junit.Assert.assertNotNull;

@RunWith(SpringRunner.class)
@ContextConfiguration(classes =
        {RepositoryConfig.class, TestDataSourceConfig.class})
public class MultipleResourcesTest {
   ...
}
```

Also, if you want to write a test class that creates a context without using the `@ContextConfiguration` annotation, the `AnnotationConfigApplicationContext` class can be used with multiple arguments as well.

```
package com.apress.cems.config;
...
class DataSourceConfigTest {

    @Test
    void testMultipleCfgSource() {
        ApplicationContext ctx =
                new AnnotationConfigApplicationContext(TestDataSource
                Config.class,
                    RepositoryConfig.class);

        EvidenceRepo evidenceRepo = ctx.getBean(JdbcEvidenceRepo.class);
        DetectiveRepo detectiveRepo = ctx.getBean(JdbcDetectiveRepo.class);

        assertNotNull(evidenceRepo);
        assertNotNull(detectiveRepo);
    }
}
```

In the configuration examples we've done so far, we have declared bean definitions (infrastructure and business) by methods annotated with `@Bean`, which are always found under a class annotated with `@Configuration`. For large applications, you can imagine that this is quite impractical, which is why beans configurations should be done using stereotype annotations. The simplest way to define a bean in Spring is to annotate the bean class with `@Component` (or any of the stereotype annotations that apply) and enable component scanning. Of course, this is applicable only to classes that are part of the project. For classes that are defined in third-party libs like the `DataSource` in the examples presented so far, declaring beans using `@Bean` is the only solution.

Having multiple configuration classes is useful to separate beans based on their purpose; for example, to separate infrastructure beans from application beans, because infrastructure change between environments.[40]

```
ApplicationContext ctx =
                new AnnotationConfigApplicationContext(
                    TestDataSourceConfig.class,
                    RepositoryConfig.class);
```

[40]Most companies use three types of environments: development, testing, and production.

In the preceding example, the configuration for the storage layer is decoupled from the rest of the application configuration in `TestDataSourceConfig` configuration class. This makes the database configuration easily replaceable depending on the environment. The contents of the `TestDataSourceConfig` configuration class contain a `dataSource` bean definition that uses an in-memory H2 database; this type of database is often used in test environments. The configuration file containing the connection information and credentials for the in-memory database are contained in the `test-datasource.properties` file.

```
db.driverClassName=org.h2.Driver
db.url=jdbc:h2:~/sample
db.username=sample
db.password=sample
```

The production configuration is (usually) more complex than a test configuration (as it usually involves transaction management and connection pooling), and requires a different implementation of the `javax.sql.DataSource`. For this example, let's assume that the database used in production is Oracle. The connection information and credentials are contained in a `prod-datasource.properties` file and the class using that file to create a production configuration using an Oracle datasource is named `ProdDataSourceConfig`. Both are depicted in the next code listing.

```
 // prod-datasource.properties
driverClassName=oracle.jdbc.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:xe
username=prod
password=prod

// ProdDataSourceConfig.java
package com.apress.cems.config;

import com.apress.cems.ex.ConfigurationException;
import oracle.jdbc.pool.OracleDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.support.PropertiesLoaderUtils;
```

```
import javax.sql.DataSource;
import java.io.IOException;
import java.sql.SQLException;
import java.util.Properties;

@Configuration
public class ProdDataSourceConfig {

    @Bean("connectionProperties")
    Properties connectionProperties(){
        try {
            return PropertiesLoaderUtils.loadProperties(
                    new ClassPathResource("db/prod-datasource.properties"));
        } catch (IOException e) {
            throw new ConfigurationException(
                     "Could not retrieve connection properties!", e);
        }
    }

    @Bean
    public DataSource dataSource() {
        try {
            OracleDataSource ds = new OracleDataSource();
            ds.setConnectionProperties(connectionProperties());
            return ds;
        } catch (SQLException e) {
            throw new ConfigurationException(
                  "Could not configure Oracle database!", e);
        }
    }
}
```

The `org.springframework.core.io.support.PropertiesLoaderUtils` provides a set of convenient utility methods for loading of `java.util.Properties`, performing standard handling of input streams. It was used here because the `OracleDataSource` class can be instantiated using a `Properties` object containing a specific set of properties. It also makes a simple configuration class, don't you think?

Based on the following information, a test context should be created by calling.

```
ApplicationContext ctx =
     new AnnotationConfigApplicationContext(TestDataSourceConfig.class,
          RepositoryConfig.class);
```

A production context should be created by calling the following.

```
ApplicationContext ctx =
     new AnnotationConfigApplicationContext(ProdDataSourceConfig.class,
         RepositoryConfig.class);
```

The differences between the two environments are only represented here by the bean of the `DataSource` type being injected into the repository classes. And because we have control of both environments, we can modify a test class to use the `ProdDataSourceConfig` class as a part of the configuration, and use debugging to inspect the `dataSource` bean. You can see the differences between the two environments in Figure 2-21.

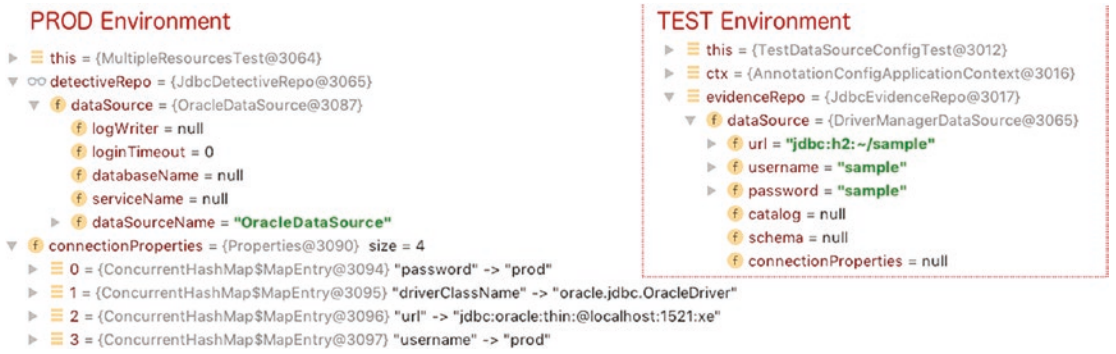

Figure 2-21. *Test and Production environments different configurations*

! Since this is the end of an important section, there is a task for you. In project `chapter02/config-practice` there is a configuration class that has gotten a little too big:

`com.apress.cems.config.FullConfig`. Split this file into one or more configuration files, and modify the test class `com.apress.cems.config.FullConfigTest` to use those files to load the context. The test must execute successfully to validate your configuration. In project `chapter02/config` you have an implementation which is quite closer to how your solution should look like.

Also, do not forget that after you have modified a class, you might want to build your project using `gradle build` in the command line or run the `compileJava` task from the IntelliJ IDEA interface to make sure you are running your test with the most recent sources. In Figure 2-22, the location of the `compileJava` task is evidentiated for you.

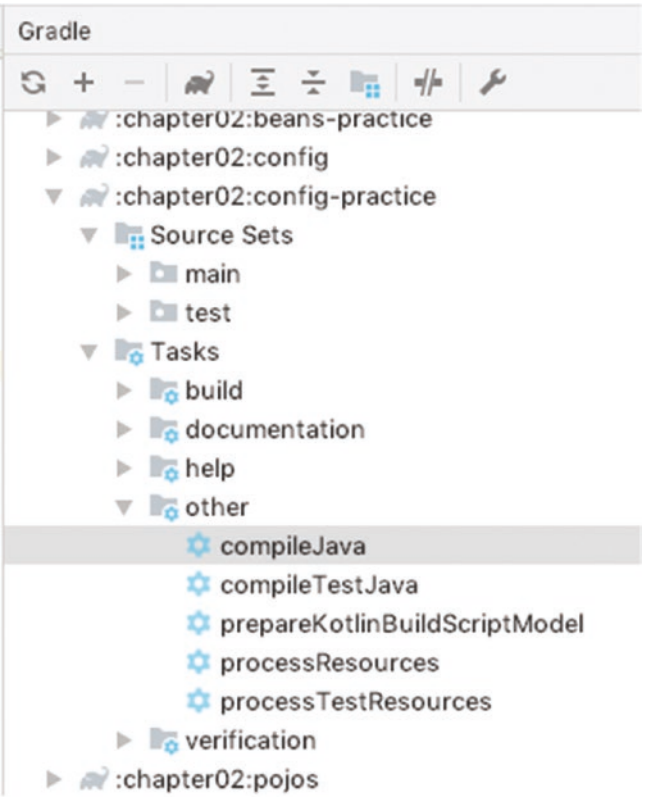

Figure 2-22. *The Gradle view in IntelliJ IDEA*

The `Environment` bean models two key aspects of an application environment: *properties* and *profiles*. We've already covered properties; it is only fair that *profiles* be covered as well. A profile is a logical group of bean definitions that is registered within the Spring IoC container when the profile is active. Similarly, profiles can group property files or property values that are supposed to be active only when the profile is. Using profiles within a Spring application is practical because it becomes easier to activate configuration for one environment or another. For the previous example we had the

following configuration classes: `TestDatasourceConfig`, `ProdDatasourceConfig`, and `RepositoryConfig`. And to bootstrap one environment or the other, we would couple the specific environment configuration class `TestDatasourceConfig` with the class declaring beans specific to both environments (e.g., `RepositoryConfig`). Using profiles, we can bootstrap an application that is configured using all the classes, but that is specific only to environment activated by the profile, because when using profiles Spring analyses the configuration classes available and creates all beans that are not annotated with `@Profile` and only the beans annotated with `@Profile` and specific to the activated profile. For this to work, fist we have to annotate each of the data source configuration classes with the `@Profile` annotation to make them specific to a profile.

```
 ...
 import org.springframework.context.annotation.Profile;

@Configuration
@PropertySource("classpath:db/test-datasource.properties")
@Profile("test")
public class TestProfileDataSourceConfig {
...
}

@Configuration
@Profile("prod")
public class ProdDataSourceConfig {
...
}
```

After doing this, we just have to bootstrap an application using a class annotated with `@ContextConfiguration`, adding all configuration classes as arguments and activating the profile by annotating it with `@ActiveProfiles`.

```
 package com.apress.cems.config;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.test.context.ActiveProfiles;
```

```
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringRunner;

import javax.sql.DataSource;

import static org.junit.Assert.assertNotNull;

@RunWith(SpringRunner.class)
@ActiveProfiles("test")
@ContextConfiguration(classes = {ProdDataSourceConfig.class,
      TestProfileDataSourceConfig, RepositoryConfig.class})
public class ProfilesTest {

    @Autowired
    DataSource dataSource;

    @Test
    public void testInjectedBeans(){
        assertNotNull(dataSource instanceof DriverManagerDataSource);
    }
}
```

The previous class activated the `test` profile which means the application context will be created only using configuration classes that are either annotated with `@Profile("test")` and classes that are not annotated with `@Profile`.

The line `assertNotNull(dataSource instanceof DriverManagerDataSource)` tests that the `dataSource` bean is of the type declared in the `TestProfileDataSourceConfig` class, the one specific to the `test` profile.

Spring Boot

Spring Boot is a set of preconfigured set of frameworks/technologies designed to reduce boilerplate configuration (infrastructure) and provide a quick way to have a Spring application up and running. Its goal is to allow developers to focus on implementation of the actual required functionality instead of how to configure an application by providing ready-to-use infrastructure beans. On the official Spring Boot page[41] a REST application

[41]Spring Boot official page: http://projects.spring.io/spring-boot/

is given as an example and almost 20 lines of code are needed to have a runnable application. Impressive, right? And this is only one of the advantages of using Spring Boot.

Spring Boot is built on top of the Spring Framework and is represented by a collection of libraries named *starters* which, when used as dependencies of a project provide a collection of preconfigured dependencies that allow starting an application with one click. Spring Boot might seem like one of those tools that seem to function *auto-magically* but really, under the hood provides the configuration of your application, so you don't have to because it is easier to change something that exists and customize it to match your needs than building it from scratch.

One thing that can be said about Spring Boot is that it is *opinionated*. It serves you with a configuration that a team of people from Pivotal thought it is suitable to what you are trying to build; so they basically turned their opinion into a configuration, they packed it up and decided to offer it to developers as a dependency.

For example, if you are trying to build a Spring web application that saves data into a database, no matter what form that data has you will most likely need the following dependencies: Spring MVC for the web layer; Spring Data JPA for the data access layer; and Spring Security if you want to restrict access to the data. You might need Spring Test to test it, and a database driver. To create a configuration like this, you first need to specify the dependencies in a build tool like Gradle and make sure that the versions are compatible. Then you have to declare configuration classes with the infrastructure beans that you need. Seems like a lot of work and very prone to errors. But since most applications of this type require almost the same dependencies and almost the same infrastructure beans, a template can be determined and provided out of the box.

Spring Boot makes it easier for developers to build stand-alone, production-level applications that are just ready to run. Spring Boot can create applications that are packed as a `jar` can be run with `java -jar` or typical `war`-packed projects that can be deployed on an application server. The new thing is that web archives (`.war` files) can be executed with `java -jar` as well.

Spring Boot focuses on the following.

- Setting up the infrastructure for a Spring project in a really short time.
- Providing a default, common infrastructure configuration, while allowing easy divergence from the default as the application grows (so any default configuration can easily be overridden).

- Providing a range of non-functional[42] features common to a wide range of projects (embedded servers, security, metrics, health checks, externalized configuration, etc.).
- Removing the need for XML and code configuration.

The stable version of Spring Boot when this book was written was 2.1.4.RELEASE, which works with JDK 11. In the book source, version 2.2.0-M2 was used. By the time this book is released, this version will have changed, since the sources are updated to work with the most recent versions of Spring and Java.

Configuration

Spring Boot can be used as any Java library. It can be added to a project as a dependency, because it does not require any special integration tools, and it can be used in any IDE. Spring Boot provides "starter" POMs to simplify the Maven configuration of a project. When using Maven, the project must have `spring-boot-starter-parent` so that the useful Maven defaults are provided. Since the sources attached to this book use Gradle, it is very suitable that Gradle is supported too. For Gradle there is no need to specify a parent, because the Spring Boot Gradle plugin makes sure that the dependencies specific to a certain type of application are provided. There are different starter dependencies, depending on the type of application; in the following sample, only a few are declared.[43] To reuse configuration declarations from the parent project, all Spring Boot components and versions will be declared in the `pivotal-certified-pro-spring-dev-exam-02/build.gradle` file.

```
//pivotal-certified-pro-spring-dev-exam-02/build.gradle configuration file
snippet
allprojects {
    group 'com.apress.cems'
    version '1.0-SNAPSHOT'
}

...
```

[42]The *non-functional* term means they are not critical for the functionality of the project (e.g., a project can function without metrics).

[43]The full list is available in the Spring Boot Reference documentation http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter

```
ext {
    springBootVersion = '2.2.0-M2'
...
    boot = [
    springBootPlugin:
        "org.springframework.boot:spring-boot-gradle-
        plugin:$springBootVersion",
    starterWeb      :
        "org.springframework.boot:spring-boot-starter-
        web:$springBootVersion",
    starterSecurity :
       "org.springframework.boot:spring-boot-starter-
       security:$springBootVersion",
    starterJpa      :
       "org.springframework.boot:spring-boot-starter-data-
       jpa:$springBootVersion",
    starterTest     :
       "org.springframework.boot:spring-boot-starter-
       test:$springBootVersion",
    actuator        :
       "org.springframework.boot:spring-boot-starter-
       actuator:$springBootVersion",
    starterWs       :
       "org.springframework.boot:spring-boot-starter-
       ws:$springBootVersion",
    devtools        :
        "org.springframework.boot:spring-boot-devtools:$springBootVersion"
    ]
        ...
}
```

In the Gradle configuration file of the parent project `pivotal-certified-pro-spring-dev-exam-02`, versions for main libraries or family of libraries are declared, as well as repositories where dependencies are downloaded from and version of Java used for compiling and running the project. The `chapter02/boot` is a subproject of the

pivotal-certified-pro-spring-dev-exam-02 project with its separate dependencies managed by the Spring Boot Gradle plugin. The chapter02/boot/build.gradle configuration file only contains the dependencies needed, without versions, because these are inherited from the parent configuration.

```
//pivotal-certified-pro-spring-dev-exam-02/chapter02/boot/build.gradle
buildscript {
    repositories {
        ...
    }

    dependencies {
     /* 1 */ classpath boot.springBootPlugin
    }
}

apply plugin: 'java-library'
/* 2 */ apply plugin: 'org.springframework.boot'

group 'com.apress.cems'
/* 3 */ext.moduleName = 'com.apress.cems.boot'

apply plugin: 'java'
apply plugin: 'idea'

dependencies {
    compile boot.starterWeb
    /* 4 */ testImplementation boot.starterTest
}
...
}
```

The marked lines have the following purposes within the configuration.

1. The springBootPlugin artifact is declared as a classpath dependency, because it is needed to build the Spring Boot dependency tree.
2. The Spring Boot Gradle Plugin is applied to the project to extend the project's capabilities, which is referred to by its name, org.springframework.boot.

3. Since we are using Java 11, the project uses modules; this line just tells Gradle what the module name of this project is.

4. This project is configured to use a very smart Gradle plugin named `java-library`[44] to manage dependencies that allows special types of dependencies, the `testImplementation` type means this dependency will be used to test the application, and that this dependency is internal to this project.

Each release of Spring Boot provides a curated list of dependencies it supports. The versions of the necessary libraries are selected so the API matches perfectly and this is handled by Spring Boot. The collection of libraries that end up in the application classpath is curated to the extreme so there is no API mismatch between versions. Therefore, the manual configuration of dependencies versions is not necessary. Upgrading Spring Boot ensures that those dependencies are upgraded as well. This can easily be proven by looking at the transitive dependencies of the `spring-boot-starter-web` in IntelliJ IDEA Gradle view, as depicted in Figure 2-23.

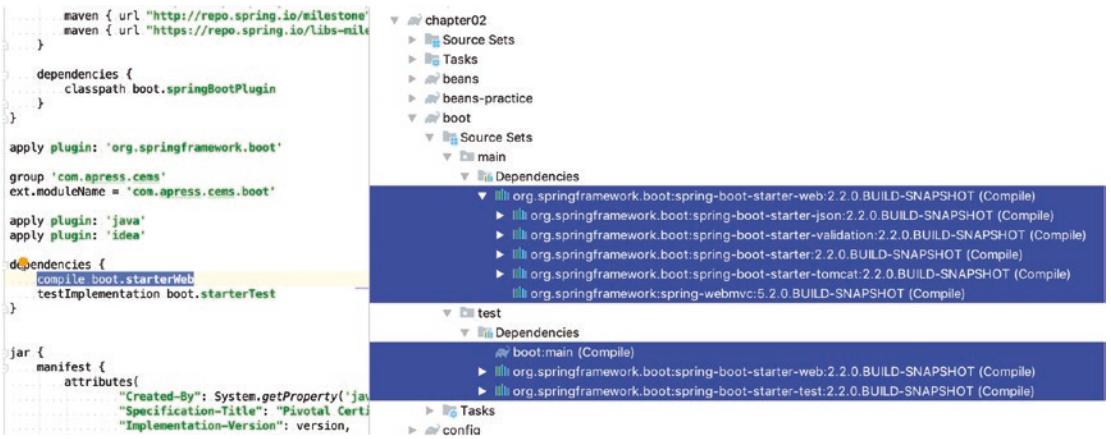

Figure 2-23. *chapter-2/boot: spring-boot-starter-web transitive dependencies in IntelliJ IDEA Gradle view*

The version for Spring Boot and project version is inherited from the `pivotal-certified-pro-spring-dev-exam-02` project. To have this project built and running, all we need is one class annotated with `@SpringBootApplication`.

[44]More info about it here: https://docs.gradle.org/current/userguide/java_library_plugin.html

This annotation is a top-level annotation designed to use only at the class level. It a convenience annotation that is equivalent to declaring the following three annotations on the same class.

- `@SpringBootConfiguration`, which is a specialization of `@Configuration` because the class is a configuration class and can declare beans with `@Bean`. Can be used as an alternative to the Spring's standard `@Configuration` annotation so that configuration can be found automatically.
- `@EnableAutoConfiguration` this is a specific Spring Boot annotation from package `org.springframework.boot.autoconfigure` that has the purpose to enable auto-configuration of the Spring ApplicationContext, attempting to guess and configure the beans that you are likely to need based on the specified dependencies and libraries found in the classpath.

  ! `@EnableAutoConfiguration` works well with Spring provided starter dependencies, but it is not directly tied to them, so other dependencies outside the starters can be used. For example, if there is a specific embedded server on the classpath this will be used unless there is another one configured for in the project (e.g., an external Tomcat server).

- `@ComponentScan`, because the developer will declare classes annotated with stereotype annotations that will become beans of some kind. By default, the base packages and its subpackages will be scanned. The attribute used to list the packages to scan with `@SpringBootApplication` is `scanBasePackages`. There are other attributes declared, such as `scanBasePackageClasses`, which allow you to specify a configuration class as a filter. The package containing it is scanned, thus providing a type-safe alternative scanning argument, because you might happen to write a package name wrong, but when specifying a class, the editor makes sure that you do not refer to a class that does not exist.

If no attribute is set relating to component scanning by default, the package where the class annotated with `@SpringBootApplication` and its subpackages will be scanned for bean annotations.

```
package com.apress.cems.boot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {Teiubesc!2
        SpringApplication.run(Application.class, args);
    }
}
```

The main method is the entry point of the application and follows the Java convention for an application entry point. This method calls the static `run` method from the `org.springframework.boot.SpringApplication` class that bootstraps the application and start the Spring IoC container, which starts the default embedded web server. Because we are creating a Spring Boot web application and we did not specify in the configuration the type of embedded server to use, the default is used, which is Tomcat. So, if you run this class in IntelliJ IDEA, or compile and build the application and execute the jar, the result will be the same: a Spring application will be started with a lot of infrastructure beans already configured with the default common configurations.

Now that we have a Spring application context, let's do something with it, like inspect all the beans. The easiest way to do this is to add a controller class that will display all of those beans. But declaring a controller means views also have to be resolved, so the simplest way is to use a REST controller. A REST controller is a controller class annotated with `@RestController`. This annotation is a combination of `@Controller`, the typical stereotype annotation marking a bean as a web component and `@ResponseBody` an annotation that basically tells spring that the result returned by methods in this class do not need to be stored in a model and displayed in a view. The `CtxController` depicted in the following code snippet contains one method that returns a simple HTML code containing a list of all beans in the application context.

```
package com.apress.cems.boot;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Arrays;

@RestController
public class CtxController {

    @Autowired
    ApplicationContext ctx;

    @GetMapping("/")
    public String index() {
        StringBuilder sb = new StringBuilder("<html><body>");

        sb.append("Hello there dear developer,
           here are the beans you were looking for: </br>");

        //method that returns all the bean names in the context of the
        application
        Arrays.stream(ctx.getBeanDefinitionNames()).sorted().forEach(
                beanName ->  sb.append("</br>").append(beanName)
        );
        sb.append("</body></htm>");
        return sb.toString();
    }
}
```

! The same thing that was done by the previous code is done more professionally by a Spring Boot Monitoring module named Spring Boot actuator. When this module is added as a dependency of a Spring Boot application, a number of endpoints are enabled for access under the path `/actuator`. The `/actuator/beans` must be activated explicitly by adding the following property to the `application.yml` file.

```
management:
endpoints:
  web:
    exposure:
      include: 'beans'
```

The Spring Boot Actuator is configured for the application corresponding to this chapter, and when opening `http://localhost:8081/boot/actuator/beans` in your browser, a complete list of all the Spring beans in this application and a few of their properties are displayed in JSON format. You can find out more about Spring Boot monitoring, including the actuator, in Chapter 9.

But, since we know how powerful the `@SpringBootApplication` annotation is, we can write the code in such a way that the functionality depicted previously is part of the `ApplicationThree` class. We know that this class is a configuration class that is treated as a bean by the Spring IoC container, so we can add the `@RestController` annotation on it, right? Yes, that is correct. We add the contents with a few modifications to this class, and we reduce our application to a single class.

```
 package com.apress.cems.boot3;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Arrays;
import java.util.function.Function;

@RestController
@SpringBootApplication
public class ApplicationThree.class {

    public static void main(String[] args) {
        SpringApplication.run(ApplicationThree.class, args);
    }
```

```
@Autowired
ApplicationContext ctx;

@GetMapping("/")
public String index() {
   return ctxController.apply(ctx);
}

Function<ApplicationContext, String> ctxController = ctx -> {
    StringBuilder sb = new StringBuilder("<html><body>");

    sb.append("Hello there dear developer,
        here are the beans you were looking for: </br>");

    //method that returns all the bean names in the context of the
    application
    Arrays.stream(ctx.getBeanDefinitionNames()).sorted().forEach(
            beanName -> sb.append("</br>").append(beanName)
    );
    sb.append("</body></htm>");
    return sb.toString();
};
```

If the application is made only from what it was listed up to this point, when running the main method and accessing `http://localhost:8080` the list of all the beans in the application context will be depicted like in Figure 2-24.

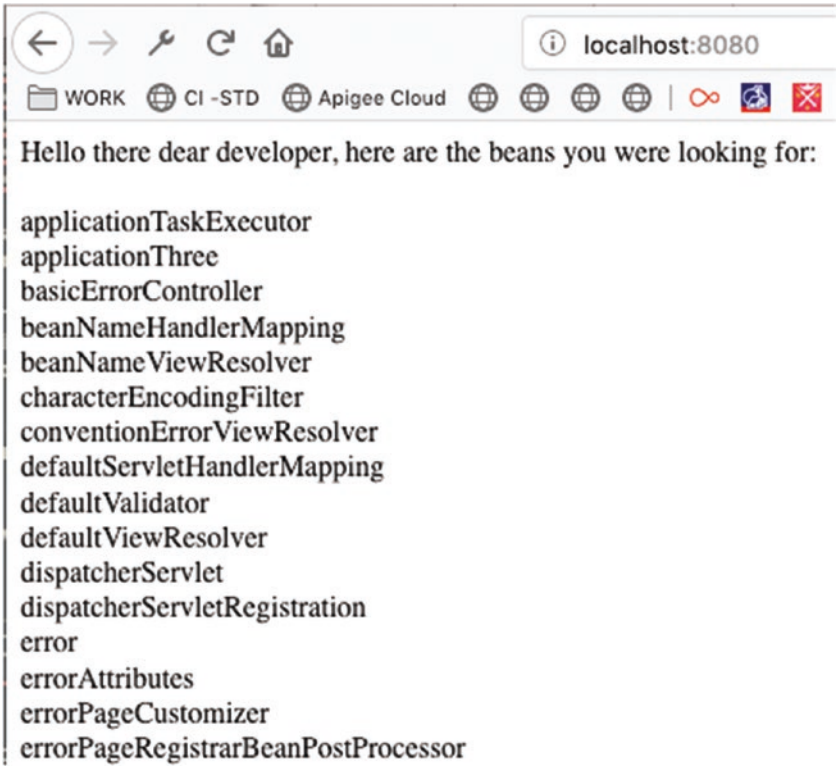

Figure 2-24. *Infrastructure beans of an application context created by Spring Boot*

To test the implementation, you can run the `ApplicationThree` class in the IDE, or you can run the application in the console. The Spring Boot Gradle Plugin provides a Gradle task named `bootJar` that packages the Spring Boot application as a runnable `jar`, just go to the `chapter02/boot` directory an execute the following command.

```
gradle clean build bootJar
```

The resulting `jar` is a web application that can be run by executing the following.

```
java -jar boot-1.0-SNAPSHOT.jar
```

If there are no exceptions thrown in the console, `http://localhost:8080` should return the expected output, exactly as before.

The application will run on an embedded Tomcat container that can be customized easily to use a different port and context for the application. There are multiple ways, but the easiest way is by adding a file named `application.properties` or `application.yml` (Spring Boot supports them out of the box) in your `resources` directory and setting some Spring Boot properties that are declared for these purposes with customized values.

```
 # application.properties
 server.port=8081
server.servlet.context-path=/boot

# application.yml
server:
  port: 8081
  servlet:
    context-path: /boot
```

Spring Boot property files could be filled with Spring Boot properties with customized values, but developer can add their own properties in there and injecting their values wherever needed. A list with all the properties that Spring Boot recognizes is found at https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

By using the preceding configuration, the web page with all the bean names within the context can be accessed now at `http://localhost:8081/boot/`, just make sure to rebuild the application and restart it, before accessing that URL.

To provide configuration for a Spring Boot application, you can use properties files, YAML files, environment variables, and command-line arguments. The mechanism of reading properties for Spring Boot involves the use of a very special `PropertySource` implementation that is designed to allow the overriding of values in a specific order. So, when running a Spring Boot application, the properties are considered in the following order.

- If you are running your application in development mode (boot-devtools library is on the classpath), properties from your home directory `~/.spring-boot-devtools.properties` are read first.
- If a `@TestPropertySource` annotation is found on your test classes, properties will be picked up from there.
- `properties` attribute on test classes on classes annotated with the @SpringBootTest annotation and the test annotations for testing a particular slice of your application.
- Properties and values can be provided using command-line arguments.

- Properties can also be provided in JSON format referred to by the `SPRING_APPLICATION_JSON` environment variable per system property
- `ServletConfig` init parameters.
- `ServletContext` init parameters.
- JNDI attributes from `java:comp/env`.
- Java System properties: accessible by calling `System.getProperties()`.
- Operating System environment variables.
- A `RandomValuePropertySource` that has properties only in random.*.
- Profile-specific application properties outside of your packaged jar (`application-{profile}.properties` or `application-{profile}.yml`).
- Profile-specific application properties packaged inside your jar (`application-{profile}.properties` or `application-{profile}.yml`).
- Application properties outside of your packaged jar (`application.properties` or `application.yml`)
- Application properties packaged inside your jar (`application.properties` and `application.yml`).
- `@PropertySource` annotations on `@Configuration` classes.
- Default properties (specified by calling `SpringApplication.setDefaultProperties`).

This list is ordered by precedence. The properties defined in the upper positions override the ones in lower positions. The `application.properties` file is provided when executing the application from the command line by using the `spring.config.location` argument.

```
java -jar ps-boot.jar --spring.config.location=
          /Users/iuliana.cosmina/temp/application.properties
```

The file name can be changed by using the `spring.config.name` in the command line.

```
java -jar sample-boot.jar --spring.config.name=my-boot.properties
```

There is more to be said about Spring Boot, but since each chapter contains a Spring Boot section that covers how to build one type of application, information is spread throughout the book for easier association with a topic and for learning that is more efficient.

And now let's have a little bit of fun customizing our Spring Boot application. While running the `ApplicationThree.class` when the application starts, the Spring Boot banner is depicted in the console. This can be replaced by creating a file named `banner.txt` under `src/main/resources` that contains the desired banner in ASCII format. The original Spring Boot banner and the Apress banner are depicted side by side in Figure 2-25.[45]

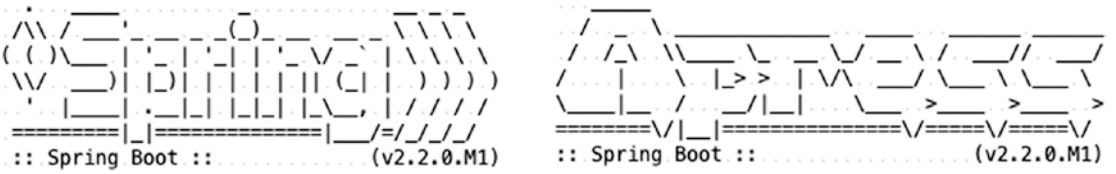

Figure 2-25. *Spring Boot Apress banner*

Logging

Spring Boot uses Logback by default, but leaves the underlying implementation open. Log4j2 and Java Util Logging are also supported. The application attached to this book makes used of the default: Logback. The starters use Logback by default. It is preconfigured to use the console as output, but it can be configured via the `logback.xml` file, which is located under `/src/main/resources`.

By default, ERROR, WARN, and INFO –level messages are logged. To modify this behavior and enable the writing of DEBUG messages for a category of core loggers (embedded container, Hibernate, and Spring Boot), the `application.properties` file must be edited and the `debug=true` property must be added.

Logging is initialized before the application context, so it is not possible to control logging from using `@PropertySources` in `@Configuration` classes. System properties and conventional Spring Boot external configuration files should be used. Depending on

[45]You can create your own ASCII banner using this site: `http://patorjk.com`

the logging system, Spring Boot looks for the specific configuration files in the following order.

- `logback-spring.xml`, `logback-spring.groovy`, `logback.xml`, `logback.groovy` for Logback
- `log4j2-spring.xml`, `log4j2.xml` for Log4j2
- `logging.properties` for Java Util Logging

The logfile name to use by Spring Boot is configured by using the `logging.file` Spring environment variable. There are other Spring environment variables that configure Spring Boot logging; the full list and purpose is available in the Spring Boot official reference documentation.[46]

Using file names postfixed with `-spring` is recommended because Spring cannot completely control log initialization when using standard configuration locations.

For now, this is all that can be said about Spring Boot. But no worries, there is a lot more to come. For this to make sense and to be easily assimilated, more knowledge about Spring is a prerequisite.

Spring Initializr

And since you've been introduced to Spring Boot that helps you build an application context with everything necessary in it in the most practical way. There is something that can make building a Spring application even quicker: **Spring Initializr**. This is a web application accessible at `https://start.spring.io/`. This application can help you start building a String application. On the web site, you can enter the desired configuration for your application, and it is generated for you with the build tool of your choice and using the language of your choice, because Spring applications can be written in Groovy and Kotlin. Let's take the site for a small run and enter the information for generating a Spring web application that makes use of JPA to save records into a H2 database and uses Spring Boot version 2.2.0-M2 and Java version 11.

In Figure 2-26, you can see a snapshot of the information entered into the Spring Initializr form.

[46]Spring Environment logging variables: `http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-log-configuration`

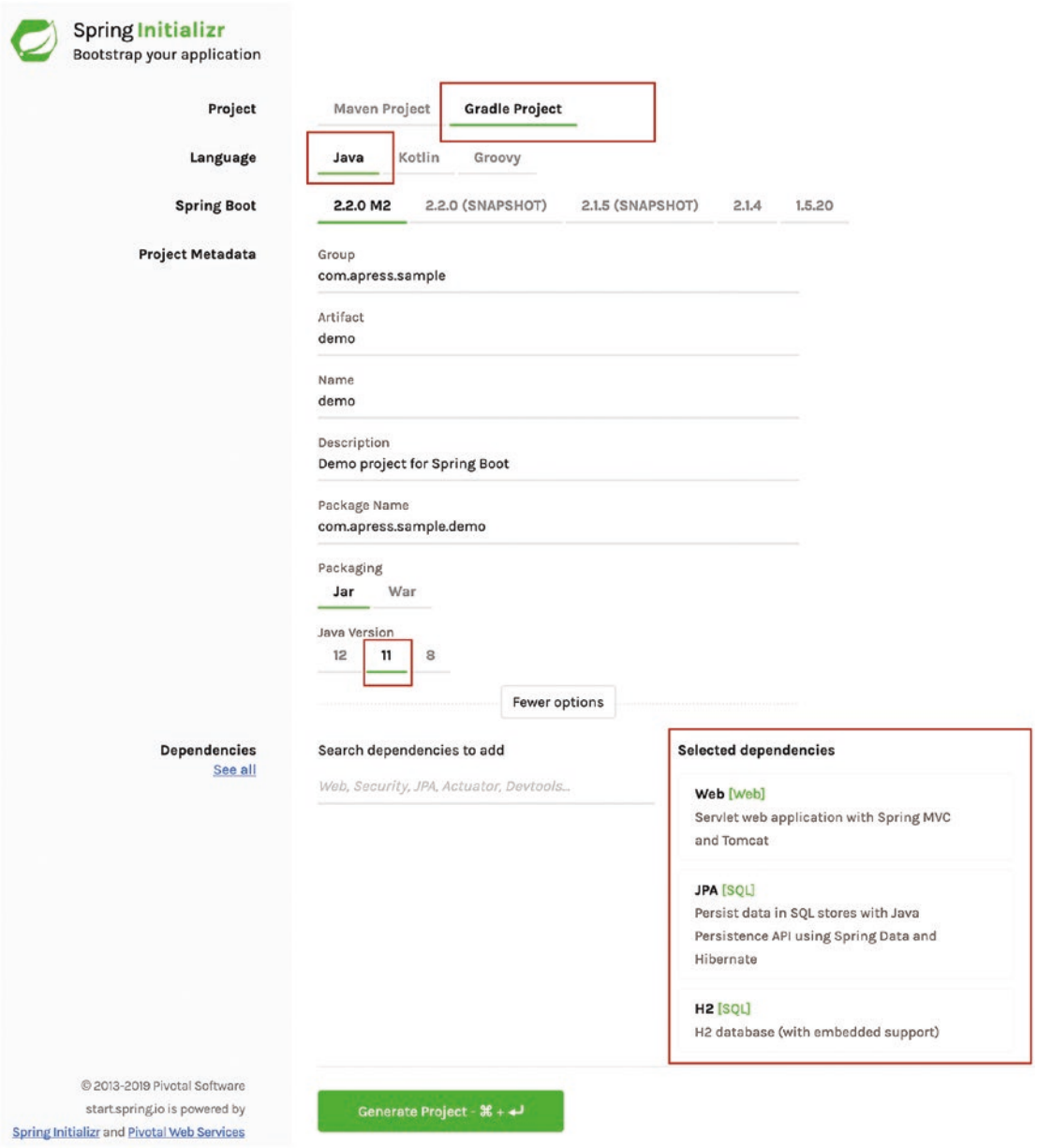

Figure 2-26. *The Spring Initializr interface with information to generate a Spring application*

When the Generate Project button is pressed, a browser pop-up window should appear asking you to save a file archived as **zip**. Save it and unpack it to inspect its contents. You should see something similar to the following output.

```
Downloads/app -  $ tree
.
|------ HELP.md
|------ build.gradle
|------ gradle
|       |------ wrapper
|               |------ gradle-wrapper.jar
|               |------ gradle-wrapper.properties
|------ gradlew
|------ gradlew.bat
|------ settings.gradle
|------ src
        |------ main
        |       |------ java
        |       |       |--- com
        |       |            |--- apress
        |       |                 |--- sample
        |       |                      |--- app
        |       |                           |--- AppApplication.java
        |       |------ resources
        |               |--- application.properties
        |               |--- static
        |               |--- templates
        |------ test
                |------ java
                        |--- com
                             |--- apress
                                  |--- sample
                                       |--- app
                                            |--- AppApplicationTests.java
18 directories, 10 files
```

The contents match the internal structure of a typical Gradle project. The `build.gradle` file is the Gradle configuration file generated by Spring Initializr and contains a compact and working Gradle single-module configuration for a Spring Boot web application.

```
plugins {
        id 'org.springframework.boot' version '2.2.0.M2'
        id 'java'
}

apply plugin: 'io.spring.dependency-management'

group = 'com.apress.sample'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

repositories {
        mavenCentral()
        maven { url 'https://repo.spring.io/snapshot' }
        maven { url 'https://repo.spring.io/milestone' }
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
        implementation 'org.springframework.boot:spring-boot-starter-web'
        runtimeOnly 'com.h2database:h2'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

You can use any type of terminal to build the project by executing

```
gradle clean build
```

Or if you do not have Gradle installed on your system, you can use the Gradle wrapper script that was generated for you.

```
./gradlew clean build
```

In your console, you should see an output looking quite similar to the following log listing.

```
 Downloads/app -  $ ./gradlew clean build
Downloading https://services.gradle.org/distributions/gradle-5.5-bin.zip
..........................................................................

Welcome to Gradle 5.5!

Here are the highlights of this release:
 - Define sets of dependencies that work together with Java Platform plugin
 - New C++ plugins with dependency management built-in
 - New C++ project types for gradle init
 - Service injection into plugins and project extensions

For more details see https://docs.gradle.org/5.5/release-notes.html

Starting a Gradle Daemon (subsequent builds will be faster)

> Task :test
2019-03-24 10:42:52.940  INFO 10244 --- [Thread-4] o.s.s.concurrent.
ThreadPoolTaskExecutor  :
   Shutting down ExecutorService 'applicationTaskExecutor'
2019-03-24 10:42:52.940  INFO 10244 --- [Thread-4] j.LocalContainerEntity
ManagerFactoryBean :
  Closing JPA EntityManagerFactory for persistence unit 'default'
2019-03-24 10:42:52.941  INFO 10244 --- [Thread-4] .SchemaDropperImpl$Delay
edDropActionImpl :
  HHH000477: Starting delayed evictData of schema as part of SessionFactory
  shut-down'
2019-03-24 10:42:52.942  INFO 10244 --- [Thread-4] com.zaxxer.hikari.
HikariDataSource       :
  HikariPool-1 - Shutdown initiated...
2019-03-24 10:42:52.946  INFO 10244 --- [ Thread-4] com.zaxxer.hikari.
HikariDataSource       :
  HikariPool-1 - Shutdown completed.

BUILD SUCCESSFUL in 50s
6 actionable tasks: 6 executed
```

The chances are slim that you would ever get a build failure, and most likely the cause would be local, like restricted rights on your computer; for example, sometimes a freshly installed Windows or macOS might not allow Java applications to be executed or ports being blocked by other applications. (Skype is known to use port 8080, the default port for a Spring Boot application.) If that ever happens, feel free to reach out and let me know, and I'll try to help.

The next step is to open this project using IntelliJ IDEA. Use either the `Import Project` from the initial pop-up window when opening IntelliJ IDEA without any project already loaded or the `Open` option from the `File` menu when you already have a project opened and choose the `app` directory. IntelliJ IDEA will guide you through the process of selecting the Gradle version and JDK for your project. After the project is loaded, you might need to select the project JDK manually in IntelliJ IDEA, because if you have multiple versions installed (as I do), version 8 might be selected by default. Just go to the `File` menu and select `Project Structure` and make sure the appropriate JDK is selected.

After that just run the `AppApplicationTests` class, if that passes you have a working application. Let's test that the application is really a web application by adding an `index.html` in the `src/main/resources/static` directory. The file can be as simple as the following.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index </title>
</head>
<body>

<h3>Sample App</h3>
</body>
</html>
```

After doing this, just run the `AppApplication` class and try to access `http://localhost:8080/`. It should be a simple web page with the *Sample App* text on it. This works because Tomcat looks for an `index.html` file to display.

At this point, you might be really enthusiastic. You have a web application, and all you did was add an HTML page to make sure the application works as intended. How

did this all happened? The simplest answer is **convention over configuration**. The web application that Spring Initializr generated is a typical Spring web application. It needs a web context that contains a typical set of infrastructure beans. By default, it uses Logback to write logs and an internal embedded Tomcat server that is started on port 8080. Any of these details can be changed by modifying the contents of the `application.properties`, which initially is empty, and by adding your own beans for that purpose.

Spring Initializr generated a fully working Gradle configuration file with dependencies on Spring Boot and the necessary plugins already configured, so you can start developing your beans right away. The configuration is quite succinct, specific to a single module project and it does not use Java modules either. That job is left to you because these configurations are quite peculiar and depend a lot on the developers' project design. But you must admit, it is nice and practical to have as a starting point when working on something new, a full working environment configuration.

There is not much that can be said about the Spring Initializr project. It is so easy to use and intuitive that the only thing that is important for a developer to know is that it can be accessed by using your browser at `https://start.spring.io/`.

Summary

After reading this chapter, you should possess enough knowledge to configure a Spring application using Java configurations and to harness the power of beans to develop small Spring applications.

Here is a little summary for you.

- Spring is built for dependency injection and provides a container for *IoC*, which takes care of injecting the dependencies according to the configuration.
- There are two flavors of configuration for Spring applications and they can also be mixed: XML-based when beans declarations are decoupled from code and Java configuration when bean declarations are in the code.
- Spring promotes the use of interfaces, so beans of types implementing the same interface can be easily interchanged.

- The Spring way of configuration promotes testability. Since beans can be interchanged, it is easy to use stubs and mocks to isolate components.
- The bean lifecycle can be controlled; behavior can be added at specific points during the bean lifecycle.
- Spring provides many ways to simplify configuration by respecting the convention over configuration principle. Bean names are precisely inferred in bean declarations that do not specifically specify one, beans are autowired by type, and there are a lot of specialized annotations available so you do not have to write your own for the most used cases.
- Bean definition sources can be coupled by importing them one into another, or by composing them to create an application context.
- Resources used to build an application context are selectable based on profiles.
- Spring Boot is the epitome of convention over configuration
- There are three terms that are specific to Spring Boot: auto-configuration, stand-alone, and opinionated.
- Spring Initializr is a Pivotal project that can generate complex Maven and Gradle Spring projects configuration.

Quiz

Question 1. What are the advantages of an application that is built making use of dependency injection? (Choose one.)

A. low coupling

B. high cohesion

C. high readability

D. easiness of testing

E. all of the above

Question 2. When should constructor injection be used? (Choose two.)

A. when the dependent bean can be created without its dependencies

B. when creating an immutable bean that depends on another bean

C. when the type of bean being created does not support other types of injection (e.g., legacy or third-party code)

Question 3. What is an application context? (Choose two.)

A. any instance of a class implementing interface ApplicationContext

B. the software representation of the Spring IoC Container

C. the means to provide configuration information for a Spring application

Question 4. Which of the following affirmations about component scanning is true? (Choose two.)

A. to enable component scanning a configuration class should be annotated with `@ComponentScan`

B. component scanning is enabled by the `@Configuration` annotation

C. `@ComponentScan` without arguments tells the Spring IoC container to scan the current package and all of its subpackages.

Question 5. We have the following bean declaration. What is the created bean's ID? (Choose one.)

```
@Bean
DataSource prodDataSource() {
    return new DriverManagerDataSource();
}
```

A. `dataSource`

B. `driverManagerDataSource`

C. `prodDataSource`

Question 6. What is the complete definition of a bean? (Choose one.)

A. a Plain Old Java Object

B. an instance of a class

C. an object that is instantiated, assembled, and managed by a Spring IoC Container

Question 7. What are the types of dependency injection supported by Spring IoC Container? (Choose all that apply.)

A. setter injection

B. constructor injection

C. interface-based injection

D. field-based injection

Question 8. The Spring IoC container by default tries to identify beans to autowire by type; if multiple beans are found, it chooses for autowiring the one with the name matching the `@Qualifier` value.

A. true

B. false

Question 9. What is the correct way to import bean definitions from a configuration class into another configuration class? (Choose one.)

A. `@Import(DataSourceConfig.class)`

B. `@Resource(DataSourceConfig.class)`

C. `@ImportResource(DataSourceConfig.class)`

Question 10. Spring Boot Question. `@SpringBootConfiguration` is a specialization of `@Configuration`? (Choose one.)

A. True

B. False

Question 11. Spring Boot Question. Which of the following annotations are used as meta annotations to declare the `@SpringBootApplication` annotation? (Choose three.)

A. `@ComponentScan`

B. `@Repository`

C. `@Import`

D. `@ContextConfiguration`

E. `@EnableAutoConfiguration`

F. `@SpringBootConfiguration`

The answers are in the appendix.