SPRING BASICS

LESSON 6

SPRING

- Spring is the most popular application development framework for enterprise Java. Millions of developers around the world use Spring Framework to create high performing, easily testable, and reusable code.

- Spring makes it easy to create Java enterprise applications. It provides everything you need to embrace the Java language in an enterprise environment, with support for Groovy and Kotlin as alternative languages on the JVM, and with the flexibility to create many kinds of architectures depending on an application’s needs.

DESIGN PHILOSOPHY

- Provide choice at every level. Spring lets you defer design decisions as late as possible. For example, you can switch persistence providers through configuration without changing your code. The same is true for many other infrastructure concerns and integration with third-party APIs.
- Accommodate diverse perspectives. Spring embraces flexibility and is not opinionated about how things should be done. It supports a wide range of application needs with different perspectives.

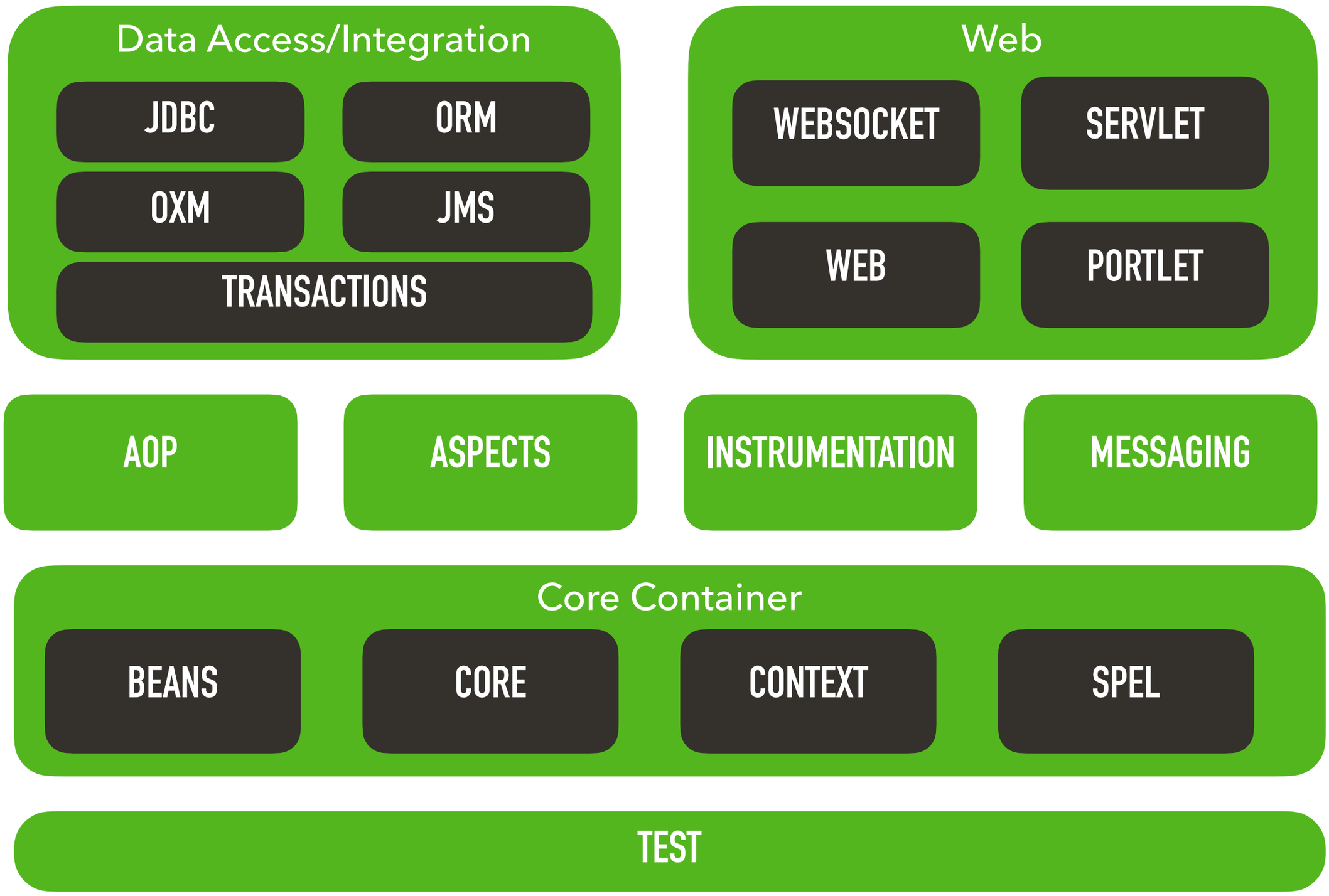
- Maintain strong backward compatibility. Spring's evolution has been carefully managed to force few breaking changes between versions. Spring supports a carefully chosen range of JDK versions and third-party libraries to facilitate maintenance of applications and libraries that depend on Spring.
- Care about API design. The Spring team puts a lot of thought and time into making APIs that are intuitive and that hold up across many versions and many years.
- Set high standards for code quality. The Spring Framework puts a strong emphasis on meaningful, current, and accurate javadoc. It is one of very few projects that can claim clean code structure with no circular dependencies between packages.

DEPENDENCY INJECTION

- The technology that Spring is most identified with is the **Dependency Injection (DI)** flavor of Inversion of Control. The **Inversion of Control (IoC)** is a general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control.
- When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

ASPECT ORIENTED PROGRAMMING (AOP)

- One of the key components of Spring is the **Aspect Oriented Programming (AOP)** framework. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, caching, etc.
- The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. DI helps you decouple your application objects from each other, while AOP helps you decouple cross-cutting concerns from the objects that they affect.
- The AOP module of Spring Framework provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.


Data Access/Integration
JDBC
ORM
OXM
JMS
TRANSACTIONS
Web
WEBSOCKET
SERVLET
WEB
PORTLET
AOP
ASPECTS
INSTRUMENTATION
MESSAGING
Core Container
BEANS
CORE
CONTEXT
SPEL
TEST

CORE CONTAINER

- The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
- The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

DATA ACCESS/INTEGRATION

- The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service **JMS** module contains features for producing and consuming messages.
- The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

WEB

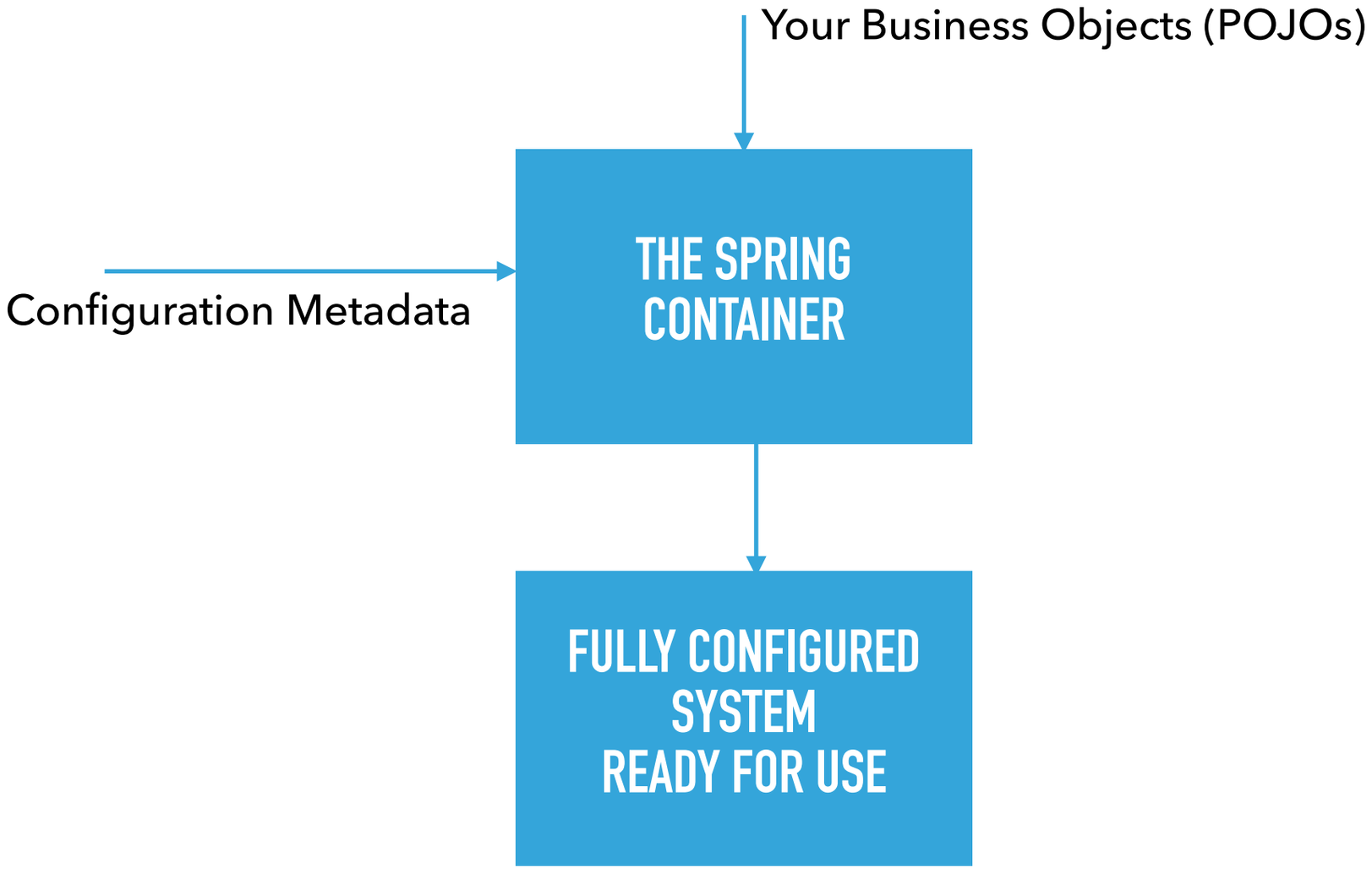
- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

MISCELLANEOUS

- The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

SPRING IOC CONTAINERS

- The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application.
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.


Your Business Objects (POJOs)
Configuration Metadata
THE SPRING CONTAINER
FULLY CONFIGURED SYSTEM READY FOR USE

- As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.

- Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.

- The **org.springframework.beans** and **org.springframework.context** packages are the basis for Spring Framework's IoC container. The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of **object.ApplicationContext** is a sub-interface of **BeanFactory**
- In short, the **BeanFactory** provides the configuration framework and basic functionality, and the **ApplicationContext** adds more enterprise-specific functionality.

XML-BASED CONFIGURATION

- **XML-based** metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. These days, many developers choose **Java-based configuration** for their Spring applications.

ARE ANNOTATIONS BETTER THAN XML?

- The introduction of annotation-based configuration raised the question of whether this approach is "better" than XML. The short answer is "it depends." The long answer is that each approach has its pros and cons, and, usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer **POJOs** and, furthermore, that the configuration becomes decentralized and harder to control.

```xml
<?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="..." class="..."> (1) (2)
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

(1) The `id` attribute is a string that identifies the individual bean definition.

(2) The `class` attribute defines the type of the bean and uses the fully qualified classname.

ANNOTATION-BASED CONFIGURATION

- The central artifacts in Spring's new Java-configuration support are **@Configuration**-annotated classes and **@Bean**-annotated methods.
- The **@Bean** annotation is used to indicate that a method instantiates, configures, and initializes a new object to be managed by the Spring IoC container. For those familiar with Spring's **<beans/>** XML configuration, the **@Bean** annotation plays the same role as the **<bean/>** element. You can use **@Bean**-annotated methods with any Spring **@Component**. However, they are most often used with **@Configuration** beans.

- Annotating a class with **@Configuration** indicates that its primary purpose is as a source of bean definitions. Furthermore, **@Configuration** classes let inter-bean dependencies be defined by calling other **@Bean** methods in the same class. The simplest possible **@Configuration** class reads as follows:

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

The preceding AppConfig class is equivalent to the following Spring <beans/> XML:

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

Using the @Bean annotation

- **@Bean** is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports some of the attributes offered by <bean/>, such as: * init-method * destroy-method * **autowiring** * name.

- You can use the **@Bean** annotation in a **@Configuration**-annotated or in a **@Component**-annotated class.

Declaring a Bean

- To declare a bean, you can annotate a method with the **@Bean** annotation. You use this method to register a bean definition within an ApplicationContext of the type specified as the method's return value. By default, the bean name is the same as the method name. The following example shows a **@Bean** method declaration:

JAVA

```java
@Configuration
public class AppConfig {

    @Bean
    public TransferServiceImpl transferService() {
        return new TransferServiceImpl();
    }
}
```

BEAN NAMING CONVENTIONS

- The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter and are camel-cased from there. Examples of such names include **accountManager**, **accountService**, **userDao**, **loginController**, and so forth.
- Naming beans consistently makes your configuration easier to read and understand. Also, if you use Spring AOP, it helps a lot when applying advice to a set of beans related by name.

Customizing Bean Naming

- By default, configuration classes use a **@Bean** method's name as the name of the resulting bean. This functionality can be overridden, however, with the name attribute, as the following example shows:

```
@Configuration
public class AppConfig {

    @Bean(name = "myThing")
    public Thing thing() {
        return new Thing();
    }
}
```

Using the @Configuration annotation

- **@Configuration** is a class-level annotation indicating that an object is a source of bean definitions. @Configuration classes declare beans through public @Bean annotated methods. Calls to @Bean methods on @Configuration classes can also be used to define inter-bean dependencies

Injecting Inter-bean Dependencies

- When beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another, as the following example shows:

```
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        return new BeanOne(beanTwo());
    }

    @Bean
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

@Component and Further Stereotype Annotations

- Spring provides further stereotype annotations: **@Component**, **@Service**, and **@Controller**. **@Component** is a generic stereotype for any Spring-managed component. **@Repository**, **@Service**, and @Controller are specializations of **@Component** for more specific use cases (in the persistence, service, and presentation layers, respectively).

- Therefore, you can annotate your component classes with @Component, but, by annotating them with **@Repository**, **@Service**, or **@Controller** instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. **@Repository**, **@Service**, and **@Controller** can also carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice.

Automatically Detecting Classes and Registering Bean Definitions

- To autodetect these classes and register the corresponding beans, you need to add **@ComponentScan** to your **@Configuration** class, where the **basePackages** attribute is a common parent package for the two classes.

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
     ...
}
```

The following alternative uses XML:

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

    <context:component-scan base-package="org.example"/>

</beans>
```

Using the @Autowired annotation

- The **@Autowired** annotation tells Spring where an injection needs to occur. If you put it on a method or field it understands that a bean needs to be injected. In the second scan, Spring searches for a bean of type class, and if it finds such bean, it injects it to this method. If it finds two such beans you will get an Exception.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

You can apply the @Autowired annotation to constructors, as the following example shows:

```
public class ConsoleUI {

    private final TaskService taskService;

    @Autowired
    public ConsoleUI(TaskService taskService) {
        this.taskService = taskService;
    }
```

As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean defines only one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.

Constructor-based or setter-based DI?

- The Spring team generally advocates constructor injection, as it lets you implement application components as immutable objects and ensures that required dependencies are not **null**. Furthermore, constructor-injected components are always returned to the client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a bad code smell, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

- Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through JMX MBeans is therefore a compelling use case for setter injection.

Instantiating the Spring Container

- In much the same way that Spring XML files are used as input when instantiating a ClassPathXmlApplicationContext, you can use @Configuration classes as input when instantiating an AnnotationConfigApplicationContext. This allows for completely XML-free usage of the Spring container, as the following example shows:

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

MIGRATION

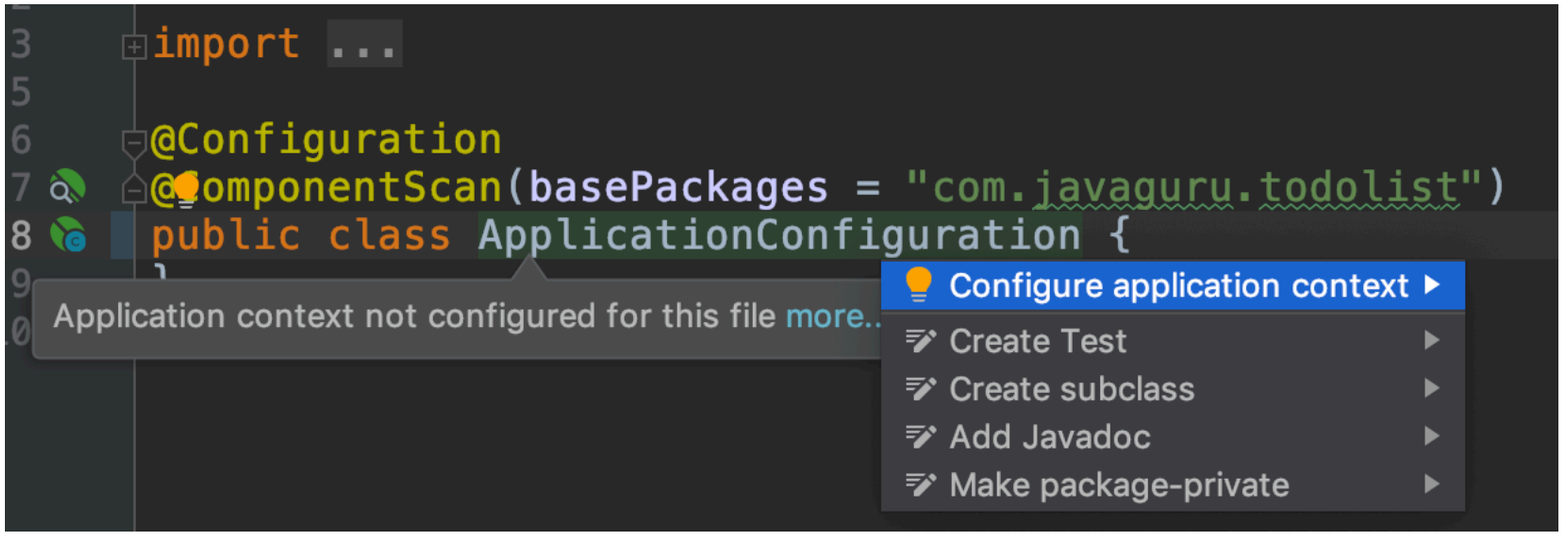
+1 dependency

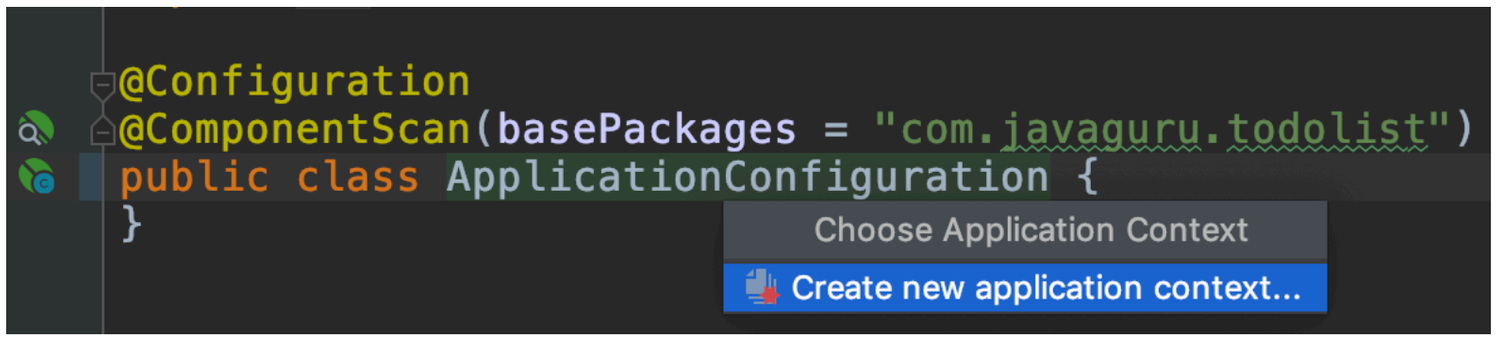
```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.22.RELEASE</version>
</dependency>
```

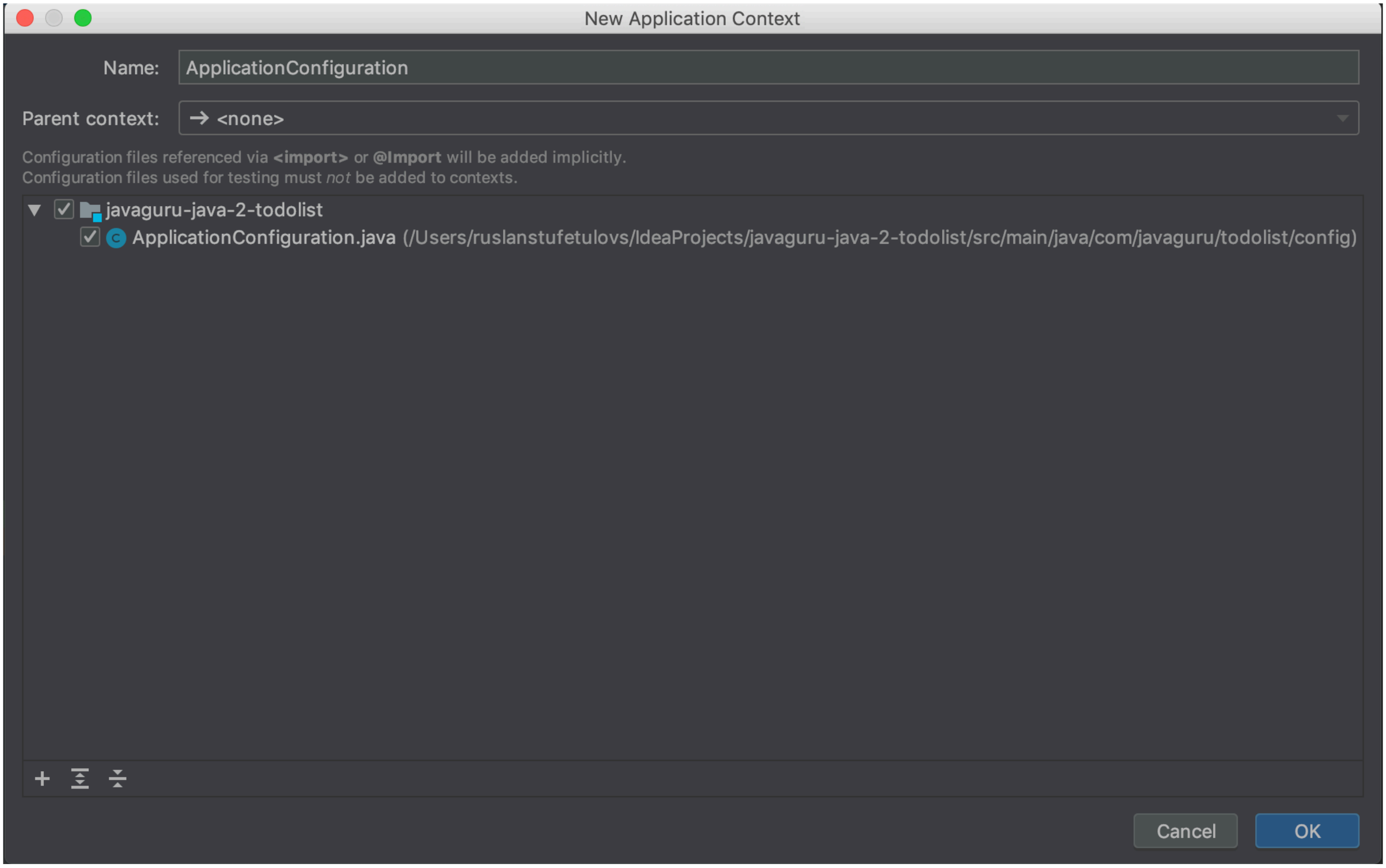
CORE CONFIGURATION

- After spring dependency added we can create our configuration class “ApplicationConfiguration”.
- Then we can annotate ApplicationConfiguration class with @Configuration to inform SpringContext about class that will be responsible for configurations and @ComponentScan to say where Spring need to search for another Beans.

```
@Configuration
@ComponentScan(basePackages = "com.javaguru.todolist")
public class ApplicationConfiguration {
}
```


import ...
@Configuration
@ComponentScan(basePackages = "com.javaguru.todolist")
public class ApplicationConfiguration {
Application context not configured for this file more...
Configure application context
Create Test
Create subclass
Add Javadoc
Make package-private


@Configuration
@ComponentScan(basePackages = "com.javaguru.todolist")
public class ApplicationConfiguration {
}
Choose Application Context
Create new application context...


New Application Context
Name:
ApplicationConfiguration
Parent context:
<none>
Configuration files referenced via <import> or @Import will be added implicitly.
Configuration files used for testing must not be added to contexts.
javaguru-java-2-todolist
ApplicationConfiguration.java (/Users/ruslanstufetulovs/IdeaProjects/javaguru-java-2-todolist/src/main/java/com/javaguru/todolist/config)
Cancel
OK