Berner
Fachhochschule

Spring Framework

Stephan Fischli

Spring

Overview

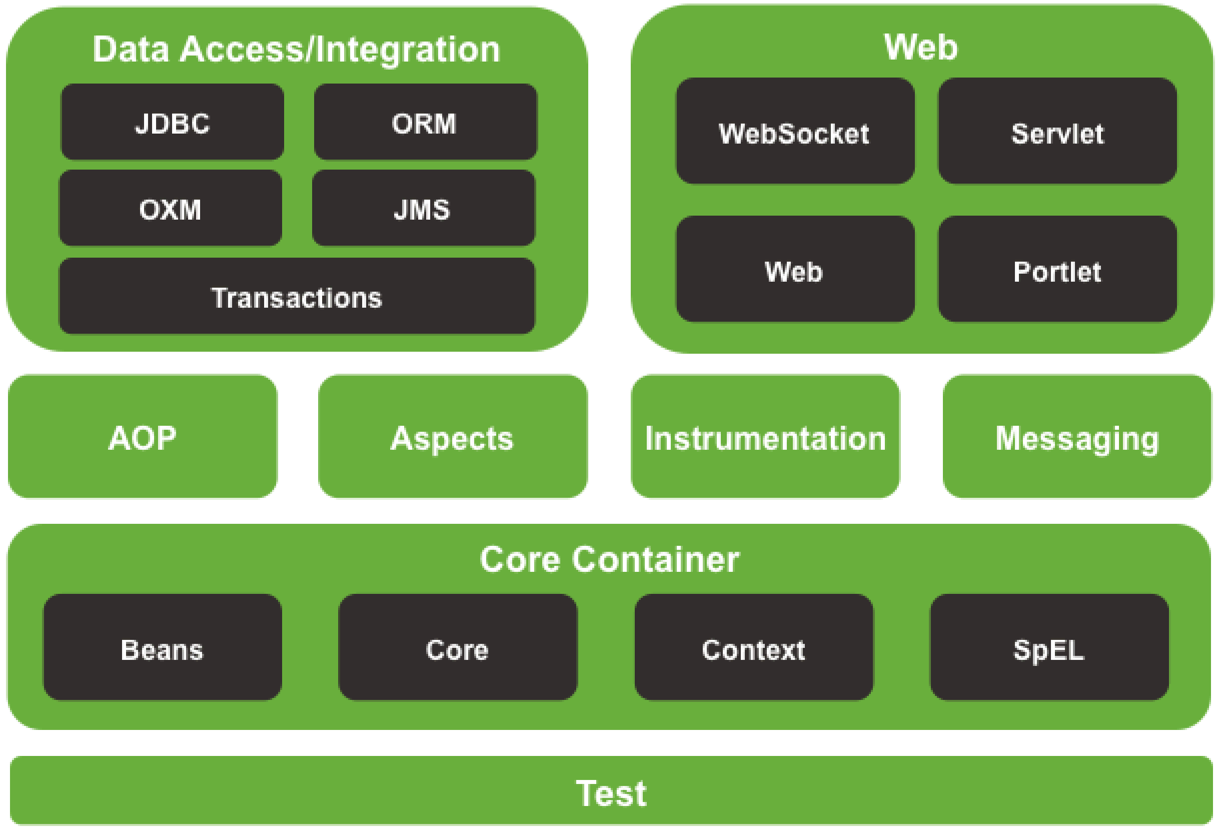
- Spring makes it easy to create Java enterprise applications
- Spring supports a wide range of application scenarios
    - applications that run as WAR on an application server
    - applications that run as single JAR with the server embedded (possibly in the cloud)
    - standalone applications (such as batch workloads)

Spring Projects

- Spring is a family of projects including
    - Spring Framework
    - Spring Boot
    - Spring Data
    - Spring Security
    - Spring Cloud
    - Spring AI
    - etc.

Spring Framework

- The Spring framework is divided into modules
- The core modules include a *configuration model* and a *dependency injection* mechanism



Spring vs Java EE

- Rod Johnson created Spring in 2003 as a response to the complexity of the early J2EE specifications
- Spring integrates with specifications from the Java EE platform (such as JPA and JMS)
- Spring has a different programming model as the Java EE platform

> *In the early days, applications were created to be deployed to an application server. Today, applications are created in a DevOps- and cloud-friendly way, with the Servlet container embedded.*

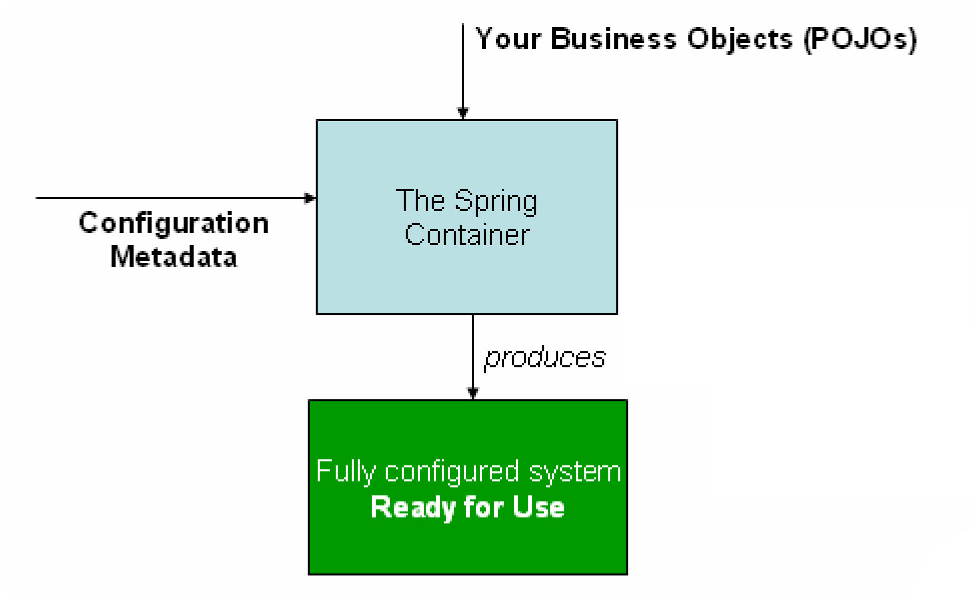
Spring Container

Introduction

- The core of the Spring framework is the Spring container (a.k.a. IoC container)
- The Spring container is the runtime environment that is responsible for
    - the lifecycle and configuration of beans (components)
    - the injection of dependencies

Inversion of Control (IoC)

- Beans define their dependencies through fields, setter methods, or constructor arguments
- But the beans do not control the instantiation or location of their dependencies
- The IoC container injects the dependencies when it creates the beans



Configuration Metadata

- The Spring container knows which beans to instantiate and how to configure and assemble them using configuration metadata
- The configuration metadata can be represented in XML, as Java annotations, or Java code
- The traditional XML-based configuration may be helpful when migrating legacy Spring applications

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ...>
    <bean id="recommender" class="org.example.MovieRecommender">
        <constructor-arg ref="catalog"/>
    </bean>
    <bean id="catalog" class="org.example.MovieCatalog"/>
    ...
</beans>
```

Component Scan

- If annotations are used, Spring can automatically detect components by scanning the classpath
- The annotation `@ComponentScan` on a configuration class is used to activate the component scan
- By default, the current package and all its subpackages are scanned, but the base packages to be scanned can also be specified

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

Using the Container

- The Spring container is represented by the `ApplicationContext` interface
- Depending on the configuration type, different implementations of the interface can be used
- The application context maintains a registry of beans and their dependencies
- The application context can be used directly, but this is usually not necessary

```
ApplicationContext context =
    new AnnotationConfigApplicationContext(ApplicationConfig.class);
MovieRecommender recommender = context.getBean(MovieRecommender.class);
...
```

Spring Beans

Introduction

- Spring beans are represented in the application context as `BeanDefinition` objects which contain
    - a name as unique identifier
    - the fully-qualified class name
    - behavioral elements (scope, lifecycle callbacks)
    - references to other beans (collaborators or dependencies)
    - other settings (e.g. the size limit of a connection pool)

Bean Annotations

- Spring provides different bean annotations
  - `@Component` is a generic annotation for Spring-managed components
  - `@Repository`, `@Service`, and `@Controller` are specializations for the persistence, service, and presentation layer which can carry additional semantics
- The annotations have a `value` attribute that defines the bean's name
- By default, the simple class name with the first letter in lower case is used

```
@Service("recommender")
public class MovieRecommender {
    ...
}
```

```
@Repository("catalog")
public class MovieCatalog {
    ...
}
```

Bean Scopes

- The scope of a bean determines where it can be used and how long it lives

Type	**Scope / Lifetime**
`singleton`	single instance for each container (default)
`prototype`	any number of object instances
`request`	lifecycle of a single HTTP request (only web)
`session`	lifecycle of an HTTP Session (only web)
`application`	lifecycle of a servlet context (only web)
`websocket`	lifecycle of a web socket (only web)

- The singleton scope should be used for stateless, the prototype scope for stateful beans

The Singleton Scope

- With a singleton bean, the container creates exactly one instance of the bean
- The instance is stored in the application context and returned with each request
- Singleton beans should not have client-specific state

```
@Service
@Scope("singleton")
public class MovieRecommender {
    ...
}
```

The Prototype Scope

- With a prototype bean, the container creates a new instance each time it is retrieved from the application context
- If a prototype bean is injected into a singleton bean, the bean is instantiated only once
- On a prototype bean, no destruction lifecycle callbacks are called

```
@Repository
@Scope("prototype")
public class MovieCatalog {
    ...
}
```

Lifecycle Callbacks

- A bean can implement callback methods to interact with the container's lifecycle management
    - `PostConstruct` lets a bean perform initialization work after the container has injected the dependencies
    - `PreDestroy` lets a bean perform clean-up work before the container is destroyed

```
@Service
public class MovieRecommender {

    @PostConstruct
    public void init() { ... }
    @PreDestroy
    public void cleanup() { ... }
    ...
}
```

Dependency Injection

Introduction

- With dependency injection, code becomes cleaner, decoupling is more effective, and classes are easier to test
- Dependencies can be injected into a bean through fields, setter methods, or constructor arguments
- The resolution mechanism is based firstly on the component type and secondly on the component name

Field Injection

- With field injection, the container sets the value of a bean's field using reflection
- Field injection is discouraged because dependencies are hidden and the bean cannot be instantiated without reflection

```
@Service
public class MovieRecommender {

    @Autowired
    private MovieCatalog catalog;
    ...
}
```

Setter Injection

- With setter injection, the container calls setter methods on the bean after its instantiation
- Setter injection should only be used for optional dependencies that can be assigned reasonable default values

```java
@Service
public class MovieRecommender {
    private MovieCatalog catalog;

    @Autowired
    public void setMovieCatalog(MovieCatalog catalog) {
        this.catalog = catalog;
    }
    ...
}
```

Constructor Injection

- With constructor injection, the container invokes a constructor with arguments representing the dependencies
- Constructor injection allows beans to be implemented as immutable objects and ensures that required dependencies are not null
- If the bean has only one constructor, the `@Autowired` annotation can be omitted

```
@Service
public class MovieRecommender {
    private final MovieCatalog catalog;

    @Autowired
    public MovieRecommender(MovieCatalog catalog) {
        this.catalog = catalog;
    }
    ...
}
```

Autowiring with Qualifiers

- When using interfaces, autowiring by type may lead to multiple injection candidates
- Qualifiers can be used to narrow the set of possible matches

```java
@Repository
@Qualifier("action")
public class ActionMovieCatalog implements MovieCatalog {
    ...
}
```

```java
@Service
public class MovieRecommender {
    private MovieCatalog catalog;

    public MovieRecommender(@Qualifier("action") MovieCatalog catalog) {
        this.catalog = catalog;
    }
    ...
}
```

Custom Qualifiers

- Instead of using the standard `@Qualifier` annotation, a custom qualifier annotation can be defined

```
@Qualifier
@Target({FIELD, PARAMETER, TYPE})
@Retention(RUNTIME)
public @interface Action {}
```

```
@Repository
@Action
public class ActionMovieCatalog implements MovieCatalog { ... }
```

```
@Service
public class MovieRecommender {
    private MovieCatalog catalog;

    public MovieRecommender(@Action MovieCatalog catalog) {
        this.catalog = catalog;
    }
    ...
}
```