Getting Started with Spring Framework 5.0 and Design Patterns

CHAPTER 1

PRESENTATION BY:
RMN

spring

Chapter One

We will cover these skills

- Basic concepts Introduction of the Spring Framework
- Simplifying application development using Spring and its pattern
  - Using the power of the POJO pattern
  - Injecting dependencies
  - Applying aspects to address cross-cutting concerns
  - Applying a template pattern to eliminate boilerplate code
- Creating a Spring container for containing beans using the Factory pattern
  - Creating a container with an application context
  - The life of a bean in the container
  - Spring modules
- New Features in Spring Framework 5.0

Introducing
Spring Framework

In the early days of Java, there were lots of heavier enterprise Java technologies for enterprise applications that provided enterprise solutions to programmers.

Spring provided a very simple, leaner, and lighter programming model compared with other existing Java technologies.

The Spring Framework is an open-source application framework and a Java-based platform that provides comprehensive infrastructure support for developing enterprise Java applications.

The Spring Inversion of Control (IoC) container is the heart of the entire framework.

Simplifying Application Development

Spring uses the following strategies to make java development easy and testable:

- Spring uses the power of the POJO pattern for lightweight and minimally invasive development of enterprise applications.
- It uses the power of the dependency injection pattern (DI pattern) for loose coupling and makes a system interface oriented.
- It uses the power of the Decorator and Proxy design pattern for declarative programming through aspects and common conventions.
- It uses the power of the Template Design pattern for eliminating boilerplate code with aspects and templates.

Using the Power of the POJO Pattern

Spring allows you to do programming with very simple non-Spring classes, which means there is no need to implement Spring-specific classes or interfaces, so all classes in the Spring-based application are simply POJOs.

That means you can compile and run these files without dependency on Spring libraries; you cannot even recognize that these classes are being used by the Spring Framework.

This is the beauty of Spring's non-invasive programming model.

Example

```
package com.packt.chapter1.spring;

public class HelloWorld {

 public String hello() {

 return "Hello World";

 }

}
```

Injecting Dependencies Between POJOs

Dependency injection is a design pattern that promotes loose coupling between the Spring components--that is, between the different collaborating POJOs.

So, by applying DI to your complex programming, your code will become simpler, easier to understand, and easier to test.

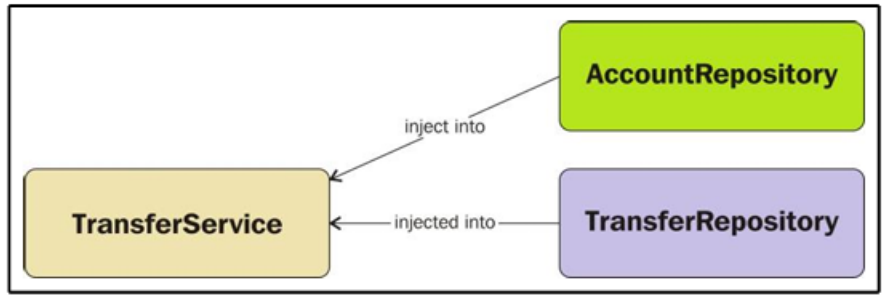
Injecting dependency between the working components helps you to unit test every component in your application without tight coupling.

Using DI Pattern
for Dependent Components

According to the DI pattern, dependent objects are given their dependencies at the time of the creation of the objects by some factory or third party.

This factory coordinates each object in the system in such a way that each dependent object is not expected to create their dependencies.

This means that we have to focus on defining the dependencies instead of resolving the dependencies of collaborating objects in the enterprise application.



Dependency injection between the different collaborating components in the application

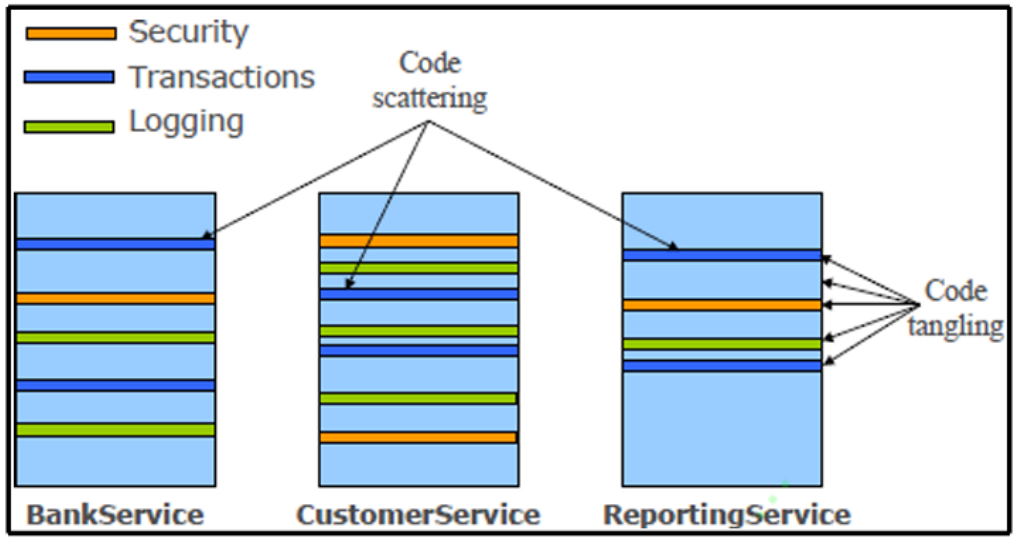
Using DI Pattern
for Dependent Components

Example

```
package com.packt.chapter1.bankapp;

public class TransferServiceImpl implements TransferService {
 private TransferRepository transferRepository;
 private AccountRepository accountRepository;

 public TransferServiceImpl(TransferRepository transferRepository, AccountRepository
accountRepository) {
 this.transferRepository = transferRepository; //TransferRepository is injected
 this.accountRepository = accountRepository; //AccountRepository is injected
 }

 public void transferMoney(Long a, Long b, Amount amount) {
 Account accountA = accountRepository.findByAccountId(a);
 Account accountB = accountRepository.findByAccountId(b);
 transferRepository.transfer(accountA, accountB, amount);
 }
}
```

Applying Aspects
for Cross Cutting Concerns

Aspect-Oriented Programming in Spring (Spring AOP) enables you to capture common functionalities that are repetitive throughout your application.



If you put these components with your core functionalities, thereby implementing cross-cutting concerns without modularization, it will have two major problems:

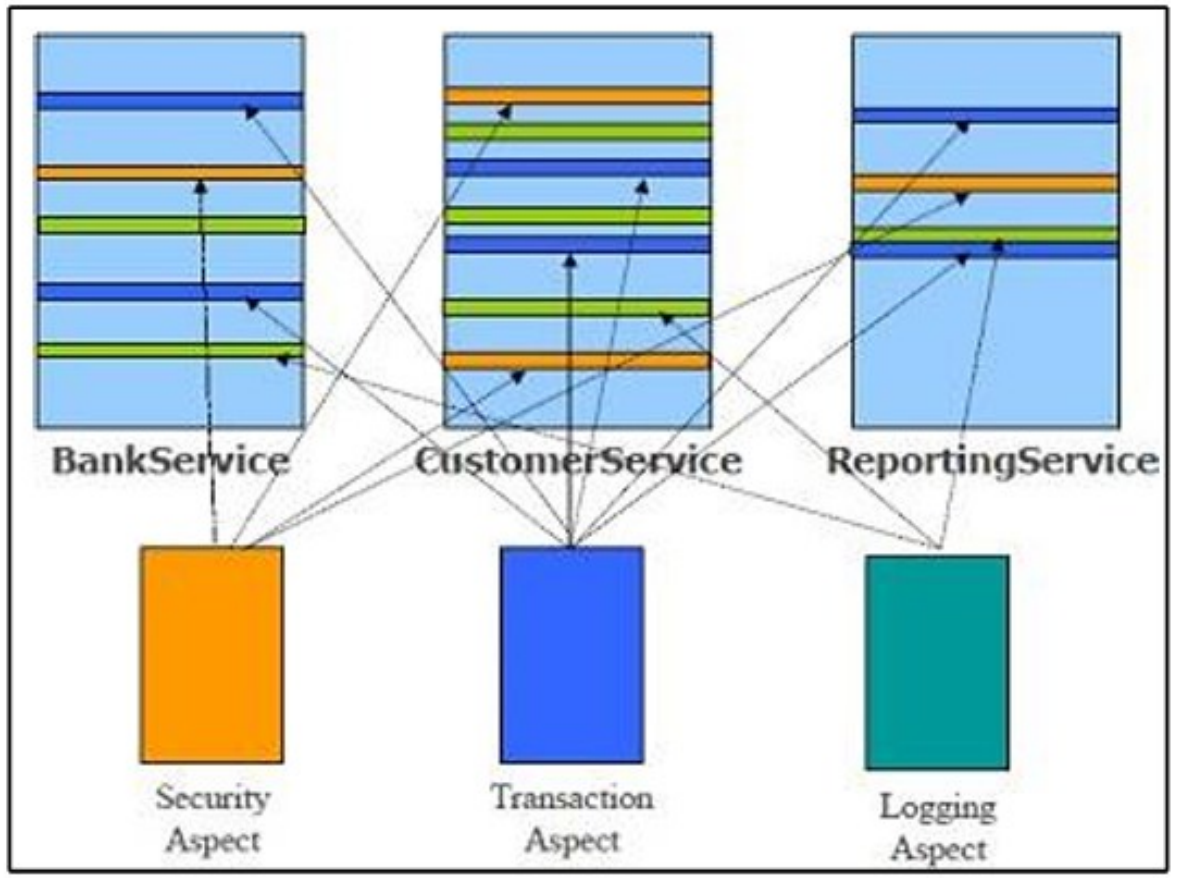
- **Code tangling**: A coupling of concerns means that a cross-cutting concern code, such as a security concern, a transaction concern, and a logging concern, is coupled with the code for business objects in your application.
- **Code scattering**: Code scattering refers to the same concern being spread across modules. This means that your concern code of security, transaction, and logging is spread across all modules of the system. In other words, you can say there is a duplicity of the same concern code across the system.

How Spring AOP Works

Implement your mainline application logic: Focusing on the core problem means that, when you are writing the application business logic, you don't need to worry about adding additional functionalities, such as logging, security, and transaction, between the business codes- Spring AOP takes care of it.

Write aspects to implement your cross-cutting concerns: Spring provides many aspects out of the box, which means you can write additional functionalities in the form of the aspect as independent units in Spring AOP. These aspects have additional responsibilities as cross-cutting concerns beyond the application logic codes.

Weave the aspects into your application: Adding the cross-cutting behaviors to the right places.



Applying
the Template Pattern

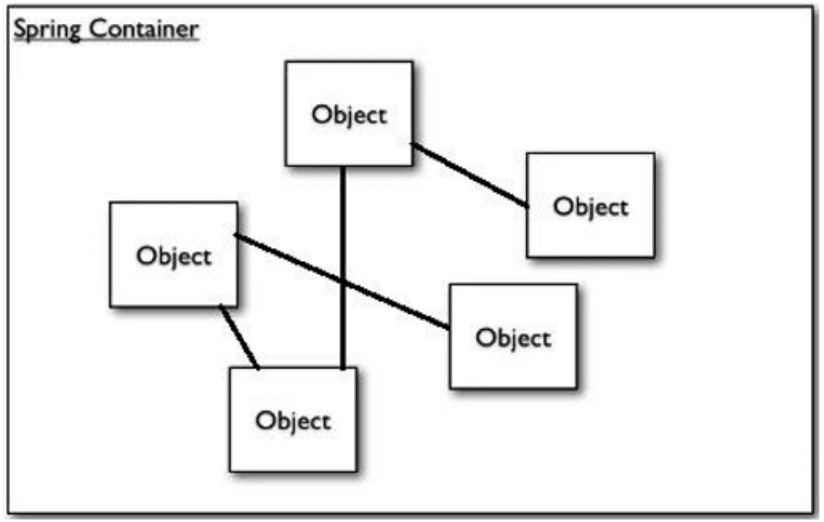
Spring JDBC solves the problem of boilerplate code by using the Template Design pattern, and it makes life very easy by removing the common code in templates.

This makes the data access code very clean and prevents nagging problems, such as connection leaks, because the Spring Framework ensures that all database resources are released properly.

The Template Design pattern in Spring:

- Define the outline or skeleton of an algorithm.
- Leave the details for specific implementations until later.
- Hide away large amounts of boilerplate code.

Using
a Spring Container



- Spring provides us with a container, and our application objects live in this Spring container.
- This container is responsible for creating and managing the objects.
- The Spring Container also wires the many Object together according to its configuration.

Basically, there are two distinct types of Spring container:

- **Bean factory**
  - The bean factory is merely an object pool where objects are created and managed by configuration.
- **Application contexts**
  - It is simply a wrapper of the bean factory, providing some extra application context services, such as support for AOP and, hence, declarative transaction, security, and instrumentation support.

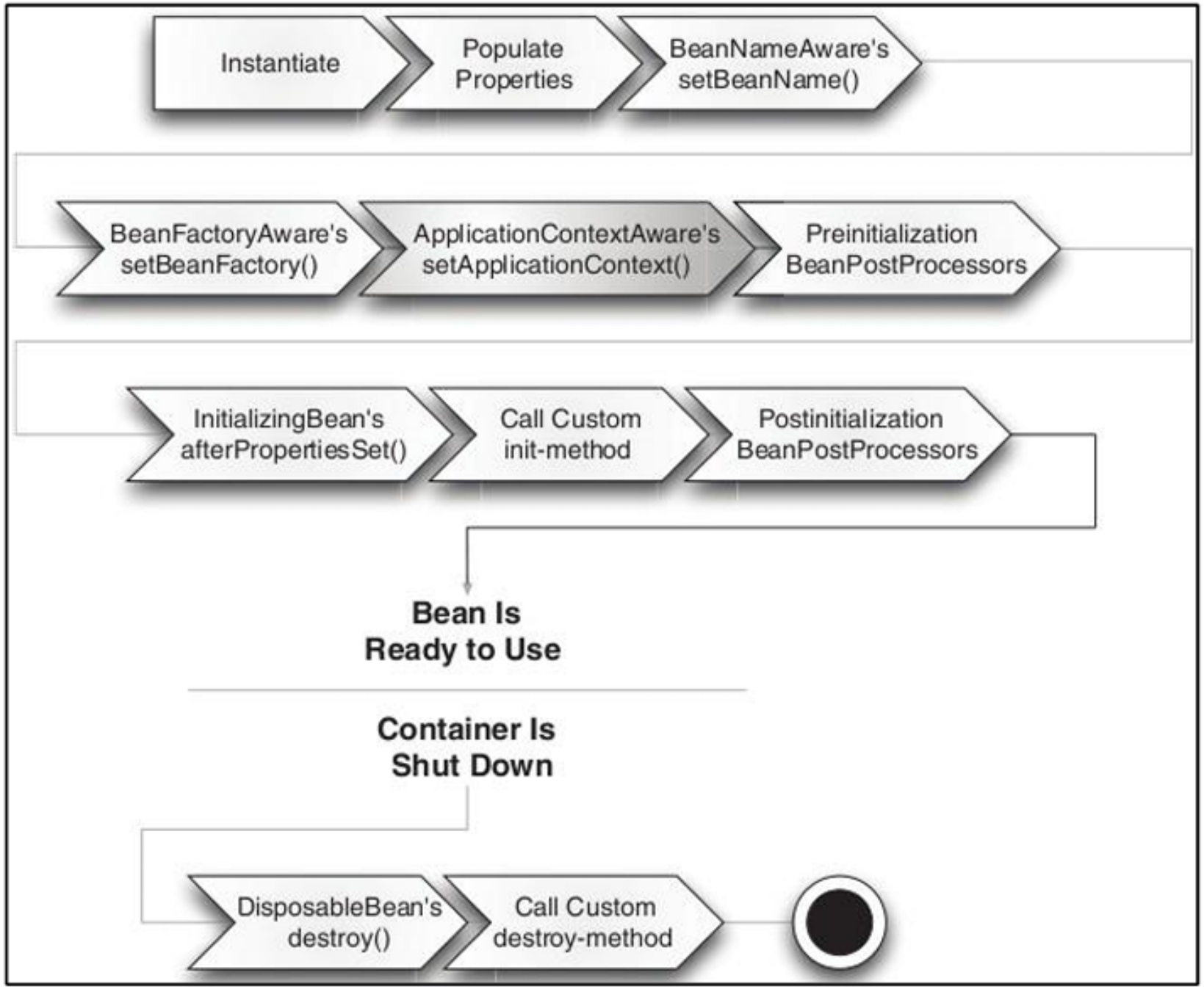
Life of a Bean
in the Container

The Spring application context uses the Factory method design pattern to create Spring beans in the container in the correct order according to the given configuration.

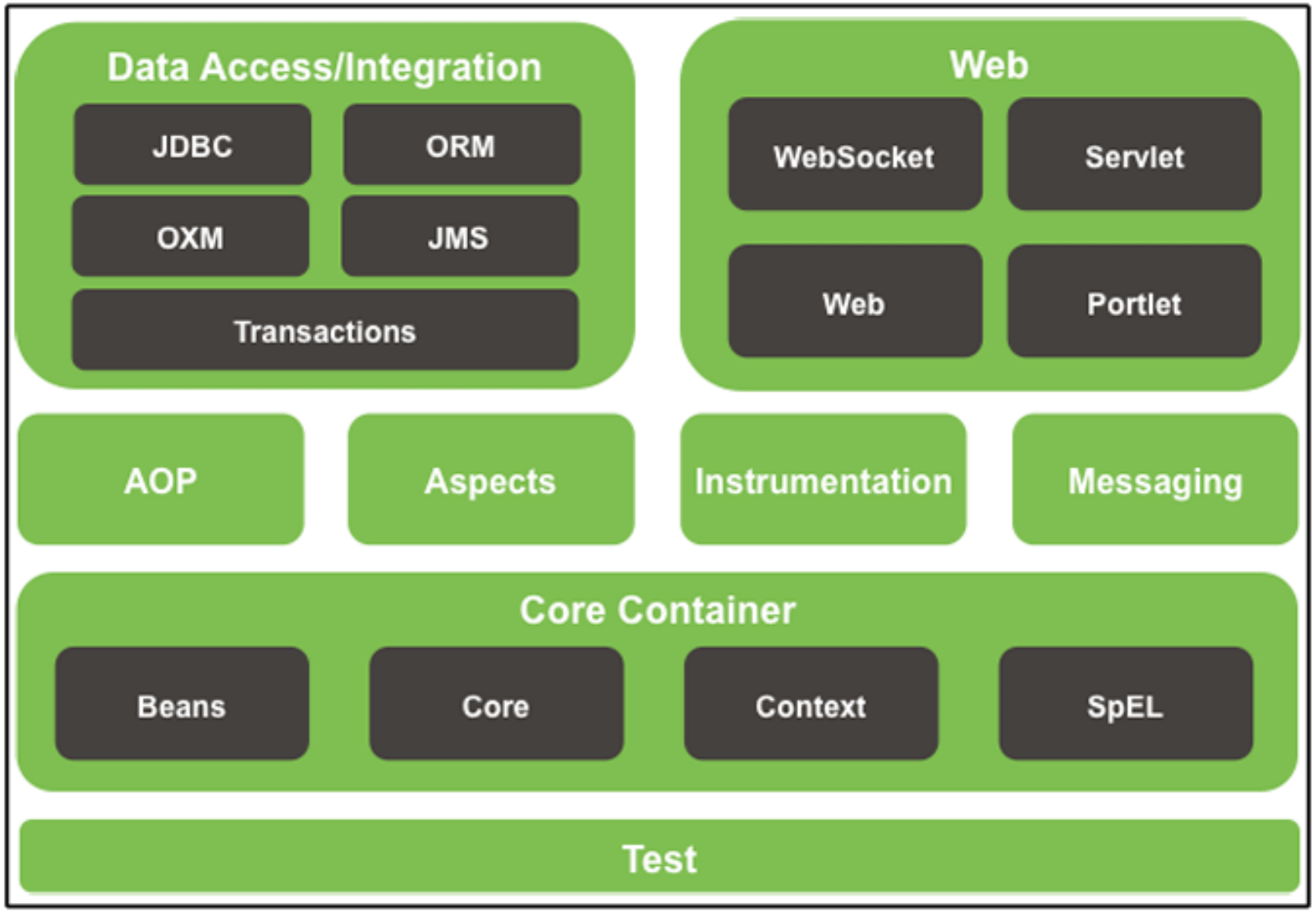
So, the Spring container has the responsibility of managing the life cycle of the bean from creation to destruction.

In the normal java application, Java's new keyword is used to instantiate the bean, and it's ready to use. Once the bean is no longer in use, it's eligible for garbage collection. But in the Spring container, the life cycle of the bean is more elaborate.

Life cycle of a Spring bean in the Spring container



Spring Modules



New Features
in Spring Framework 5.0

Support for JDK 8+ and Java EE 7 Baseline: Spring 5 supports Java 8 as a minimum requirement, as the entire framework codebase is based on Java 8.

Adding the new reactive programming model:

- Spring 5 introduced the Spring-core module DataBuffer and encoder/decoder abstractions with non-blocking semantics into the reactive programming model.
- Spring 5.0 also introduced a new WebClient with reactive support on the client side to access services.

Thank you

RMN