Java Spring Frameworks 6,7 5

Spring Architecture, Spring MVC Module, Life Cycle of Bean Factory, Spring Aspect of Object Oriented Concepts – Join Point and Point Cuts, Need of Spring Boot, Difference between Spring & Spring Boot, Building Spring Boot Application Rest Annotation with In Memory Database & CRUD Operations

POJO

•POJO in Java stands for Plain Old Java Object. **It is an ordinary object, which is not bound by any special restriction**. The **POJO file does not require any special classpath**. It i**ncreases the readability & re-usability** of a Java program.

•POJOs are now widely accepted due to their **easy maintenance**. They are easy to read and write. A POJO class **does not have any naming convention for properties and methods**. It is not tied to any Java Framework; any Java Program can use it.

•**a POJO class contains variables and their Getters and Setters**.

•The POJO classes are **similar to Beans as both** are used to define the objects to increase the readability and re-usability. The only difference between them that Bean Files have some restrictions but, the POJO files do not have any special restrictions.

Java Beans

- **Beans are special type of Pojos. There are some restrictions on POJO to be a bean.**
- It should have a no-arg constructor.
- It should be Serializable.
- It should provide methods to set and get the values of the properties, known as getter and setter methods
- it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

JavaBean Properties

A JavaBean property is a named feature that can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define.
A JavaBean property may be read, write, read-only, or write-only. JavaBean features are accessed through two methods in the JavaBean's implementation class:

Properties of setter Methods

1. It must be public in nature.
2. The return type a should be void.
3. The setter method uses prefix set.
4. It should take some argument.

Properties of getter Methods

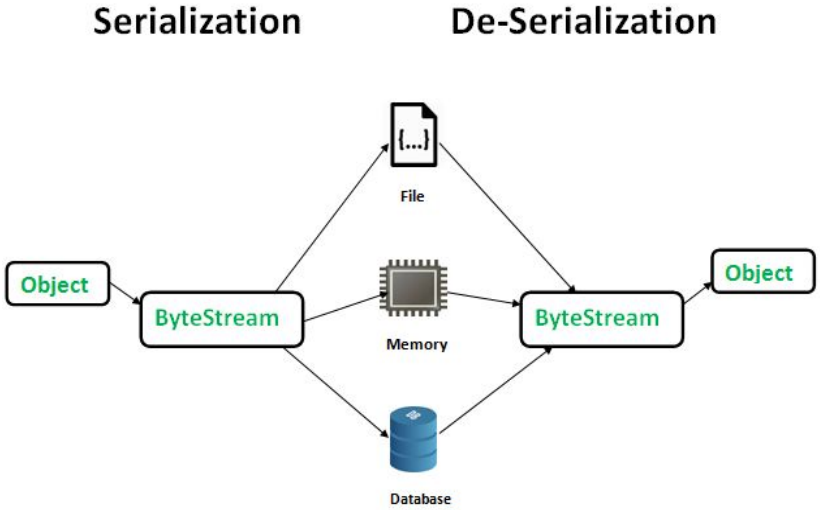
1. It must be public in nature.
2. The return type should not be void.
3. The getter method uses prefix get.
4. It does not take any argument.

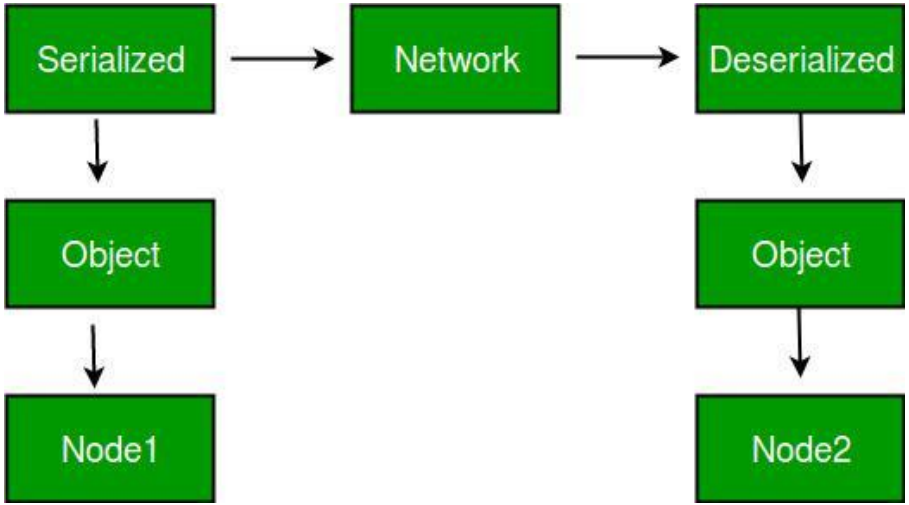
Life Cycle of Java Beans

1. Instantiation: The first stage in the life cycle of a JavaBean is instantiation. During this stage, the JavaBean is created, usually by calling a no-argument constructor. It is a simple creation step where the bean is allocated memory and its initial state is set up.
2. Customization (Optional): After a bean is instantiated, it may undergo customization. It involves configuring the bean's properties and other settings to suit specific needs. Customization typically happens in two ways:
   - Design Time: Using tools such as IDEs where beans can be configured using property sheets.
   - Runtime: Programmatically setting properties through setter methods or configuration files (XML, JSON, etc.).
3. Setting Property Values: Once the bean is customized, its properties are set. It is often done through setter methods defined in the bean. Property values can be set directly by the application that uses the bean, or by a framework or container that manages the bean.
4. Connection with Other Beans: JavaBeans can be connected to other beans. It can involve setting up event listeners and event sources, allowing beans to respond to events triggered by other beans. This step is crucial for the bean to interact within an application or environment, supporting the event-driven programming model.

Life Cycle of Java Beans

5. Activation (Optional): For beans that are serialized and later restored, the activation stage involves restoring the bean from its serialized state. Activation may involve re-establishing transient connections or resources that were not saved during serialization.
6. Introspection: Introspection is a process where the capabilities of a bean are examined dynamically. Tools and frameworks can inspect a bean to determine its properties, events, and methods. This allows for dynamic configuration and interaction with the bean.
7. Running: During this stage, the bean is in use and performing its intended tasks. It may handle data processing, respond to user input, or interact with other components and beans. This phase is generally where the bean spends most of its lifecycle.
8. Passivation (Optional): In environments where beans are managed (like EJB containers), passivation is a stage where a bean's state is temporarily stored to free resources. It usually happens when the bean is not currently needed, but its state needs to be preserved for later activation.
9. Destruction: The final stage in the lifecycle of a JavaBean is its destruction. Here, the bean is marked for garbage collection, and any resources it used are released. Before destruction, cleanup methods can be called to ensure that all resources are freed properly, such as closing database connections or releasing file handles.


Serialization
De-Serialization
Object
ByteStream
File
Memory
Database
ByteStream
Object


Serialized
Network
Deserialized
Object
Object
Node1
Node2

Serialization and Deserialization

Serialization is the conversion of the state of an object into a byte stream; deserialization does the opposite. Stated differently, serialization is the conversion of a Java object into a static stream (sequence) of bytes, which we can then save to a database or transfer over a network

The serialization process is instance-independent; for example, we can serialize objects on one platform and deserialize them on another. **Classes that are eligible for serialization need to implement a special marker interface,** *Serializable.* .

If you want a class object to be serializable, all you need to do it implement the java.io.Serializable interface. Serializable in java is a marker interface and has no fields or methods to implement. It's like an Opt-In process through which we make our classes serializable. Serialization in java is implemented by ObjectInputStream and ObjectOutputStream, so all we need is a wrapper over them to either save it to file or send it over the network.

Points to remember Serialization and Deserialization

1. If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true.
2. Only non-static data members are saved via Serialization process.
3. Static data members and transient data members are not saved via Serialization process.So, if you don't want to save value of a non-static data member then make it transient.
4. Constructor of object is never called when an object is deserialized.
5. Associated objects must be implementing Serializable interface.

Advantages of Serialization

1. To save/persist state of an object.
2. To travel an object across a network.

SerialVersionUID

The Serialization runtime associates a version number with each Serializable class, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization.
If the reciever has loaded a class for the object that has different UID than that of corresponding sender's class, the Deserialization will result in an **InvalidClassException**.
A Serializable class can declare its own UID explicitly by declaring a field name.
It must be static, final and of type long.
i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;
If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification.

However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

It is also recommended to use private modifier for UID since it is not useful as inherited member.

Spring J2EE Framework

J2EE stands for Java 2 Platform, Enterprise Edition.

J2EE is the standard platform for developing applications in the enterprise and is designed for enterprise applications that run on servers.

J2EE provides APIs that let developers create workflows and make use of resources such as databases or web services.

J2EE consists of a set of APIs. Developers can use these APIs to build applications for business computing.

Portability

Reusability

Security

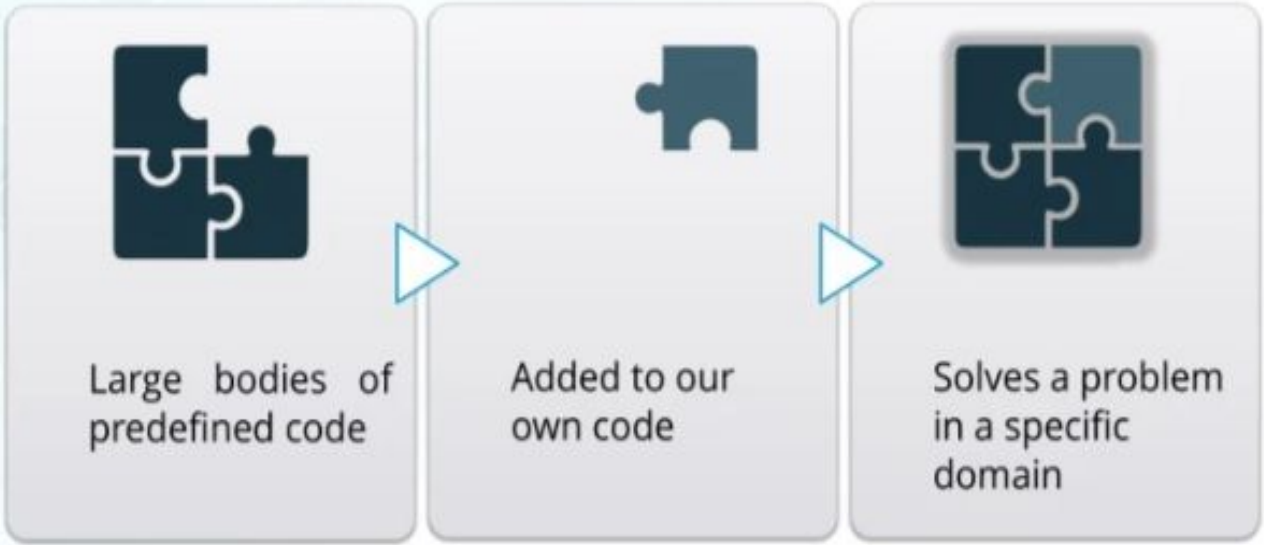
Scalability

Reliability

Why JAVA Frameworks

What Are Java Frameworks?

Frameworks are large bodies (usually many classes) of predefined code to which we can add to our own code to solve a problem in a specific domain.



Advantages of Java Framework

EFFICIENCY

SECURITY

EXPENSE

SUPPORT

Disadvantages of Java Framework

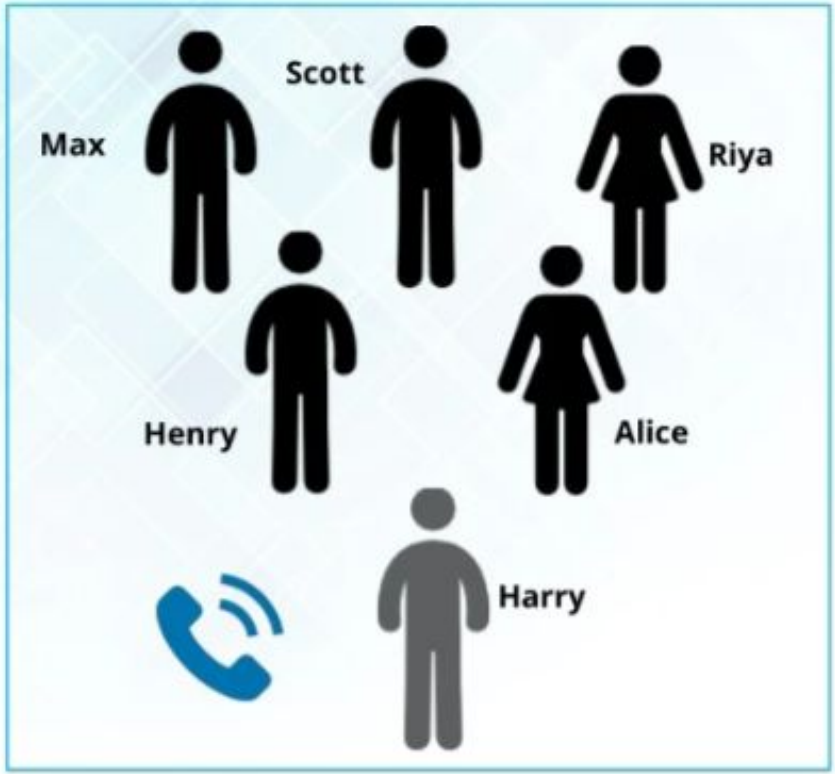
RESTRICTION

CODE IS PUBLIC

CUSTOM BUILT FEATURES

Different Java Frameworks

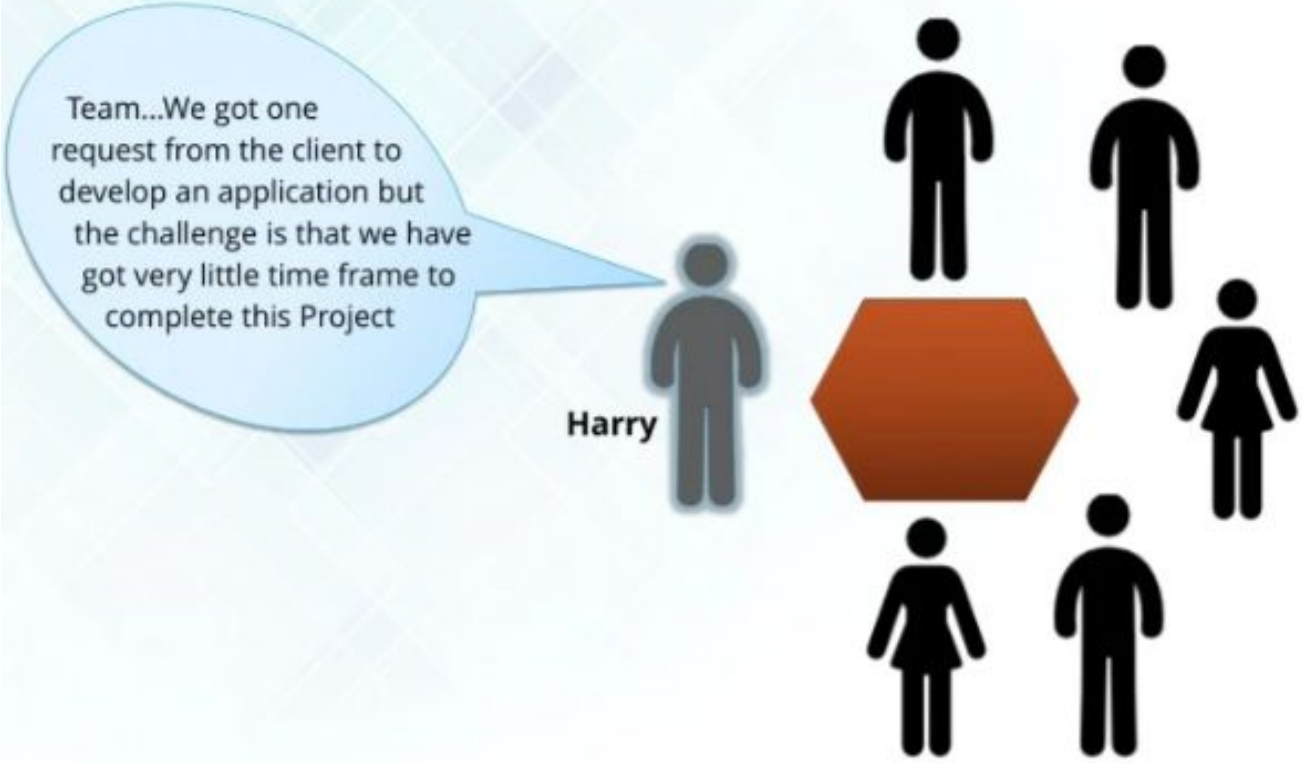
Spring



Web Development Team of XYZ company with a group of six members, headed by Harry.

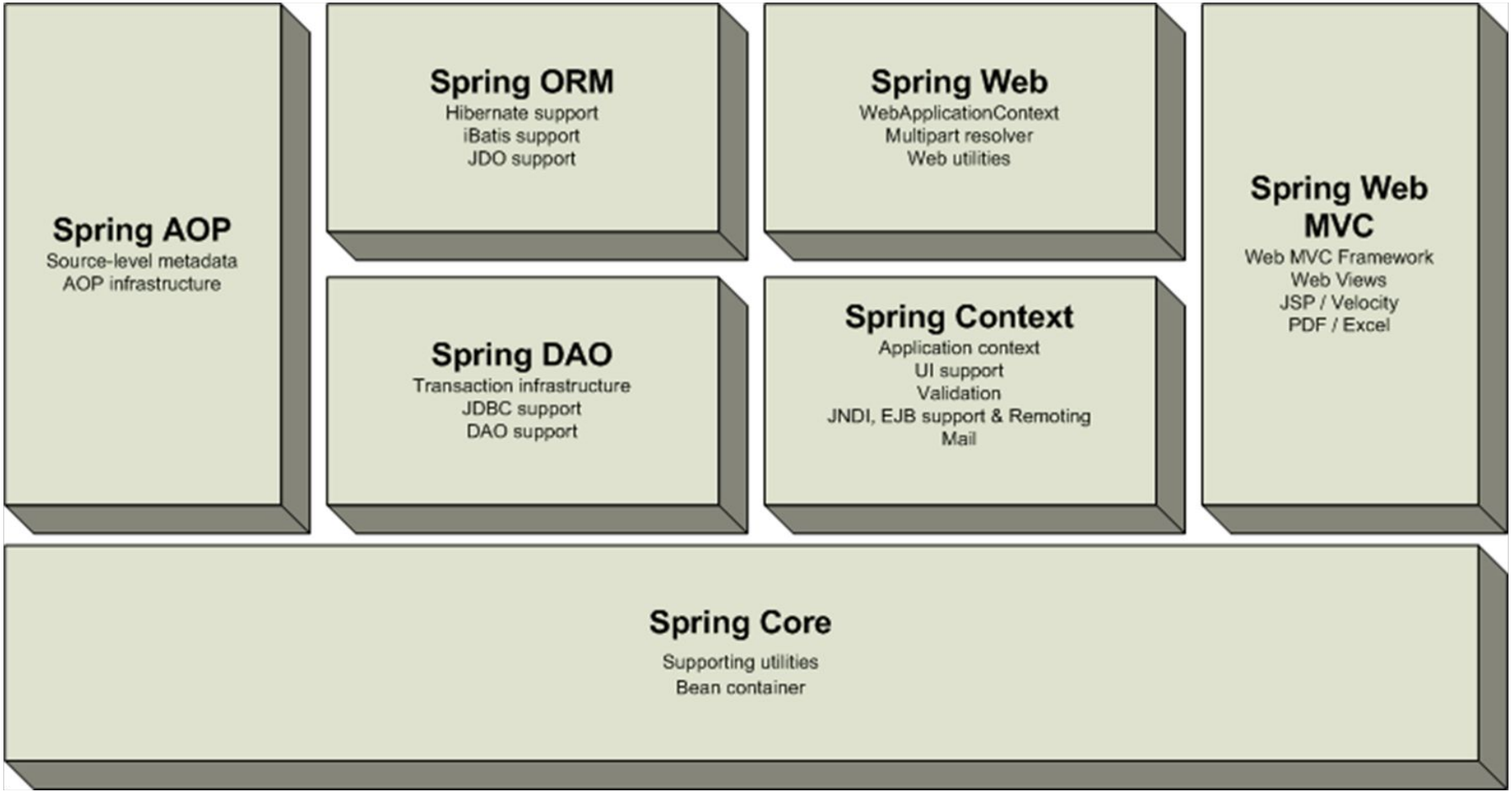
Harry got a call from the management team for one urgent requirement.

Spring


Team...We got one request from the client to develop an application but the challenge is that we have got very little time frame to complete this Project
Harry

Spring

- A lightweight non-intrusive framework which addresses various tiers in a J2EE application.
  - Presentation layer: Integrates with Struts to initialize action classes
  - Business layer: Lightweight IoC container with support for AOP-driven interceptors and transaction.
  - Persistence layer – DAO template support for Hibernate, SQLMaps and JDBC
  - Factory implementation to abstract and integrate various other facets of enterprise applications like E-mails, JMS, WebServices, etc.
- Helps integrates tiers together using XML configuration instead of hard-coding.
- Substantially reduces code, speeds up development, facilitates easy testing and improves code quality.


Spring AOP
Source-level metadata
AOP infrastructure
Spring ORM
Hibernate support
iBatis support
JDO support
Spring Web
WebApplicationContext
Multipart resolver
Web utilities
Spring Web MVC
Web MVC Framework
Web Views
JSP / Velocity
PDF / Excel
Spring DAO
Transaction infrastructure
JDBC support
DAO support
Spring Context
Application context
UI support
Validation
JNDI, EJB support & Remoting
Mail
Spring Core
Supporting utilities
Bean container

What is Spring Framework?

Reduces the overall complexity

Comprehensive tool

Lightweight application framework

Framework of frameworks

Powerful open source

Avails an array of resources

Spring Benefits

•POJO(Plain old Java Object)-based, non-invasive framework which **allows *ala carte* usage of its components**.

•Promotes **decoupling and reusability**

•**Reduces coding effort and enforces design discipline** by providing **out-of-box implicit pattern** implementations such as singleton, factory, service locator etc.

•**Removes common code issues** like leaking connections and more

•**Support for declarative transaction management**

•Easy **integration with third party tools and technologies**.

As an application developer, can use the Spring platform advantage:

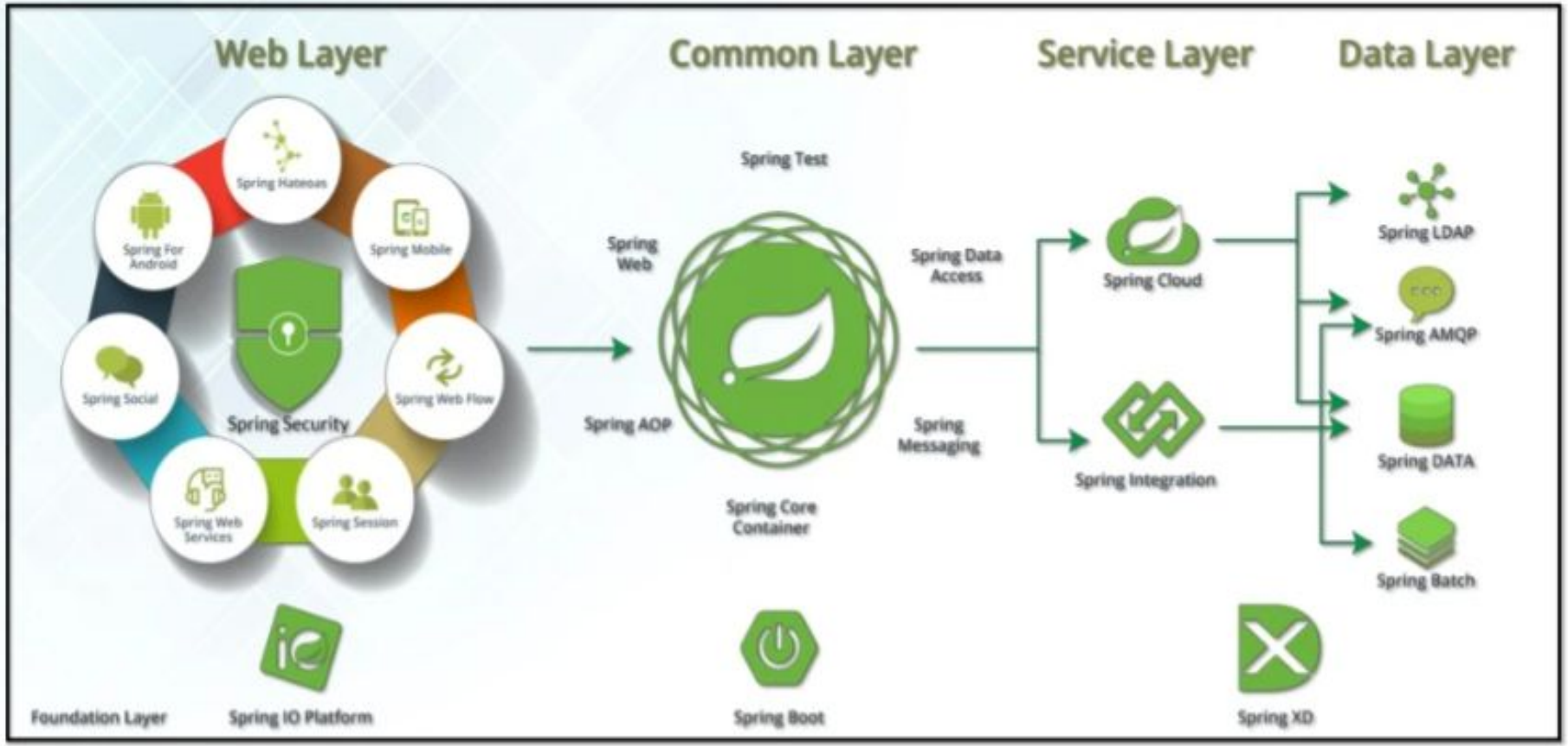
•Make a Java method execute in a database transaction without having to deal with transaction APIs.

•Make a local Java method a remote procedure without having to deal with remote APIs.

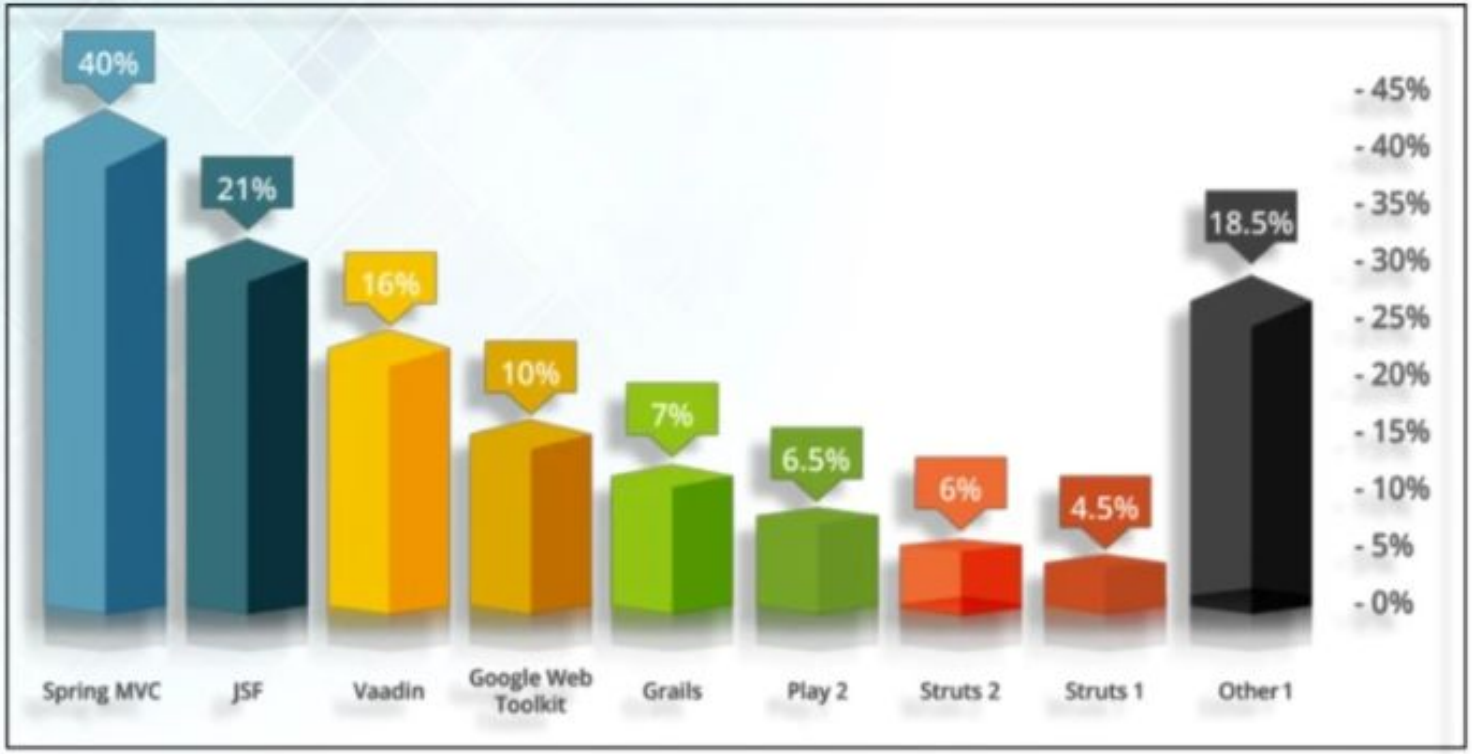
•Make a local Java method a management operation without having to deal with Java Management Extensions JMX API

•Make a local Java method a message handler without having to deal with Java Messaging Service (JMS) APIs.

Spring Framework Ecosystem



Uses Of Spring Over Other Frameworks

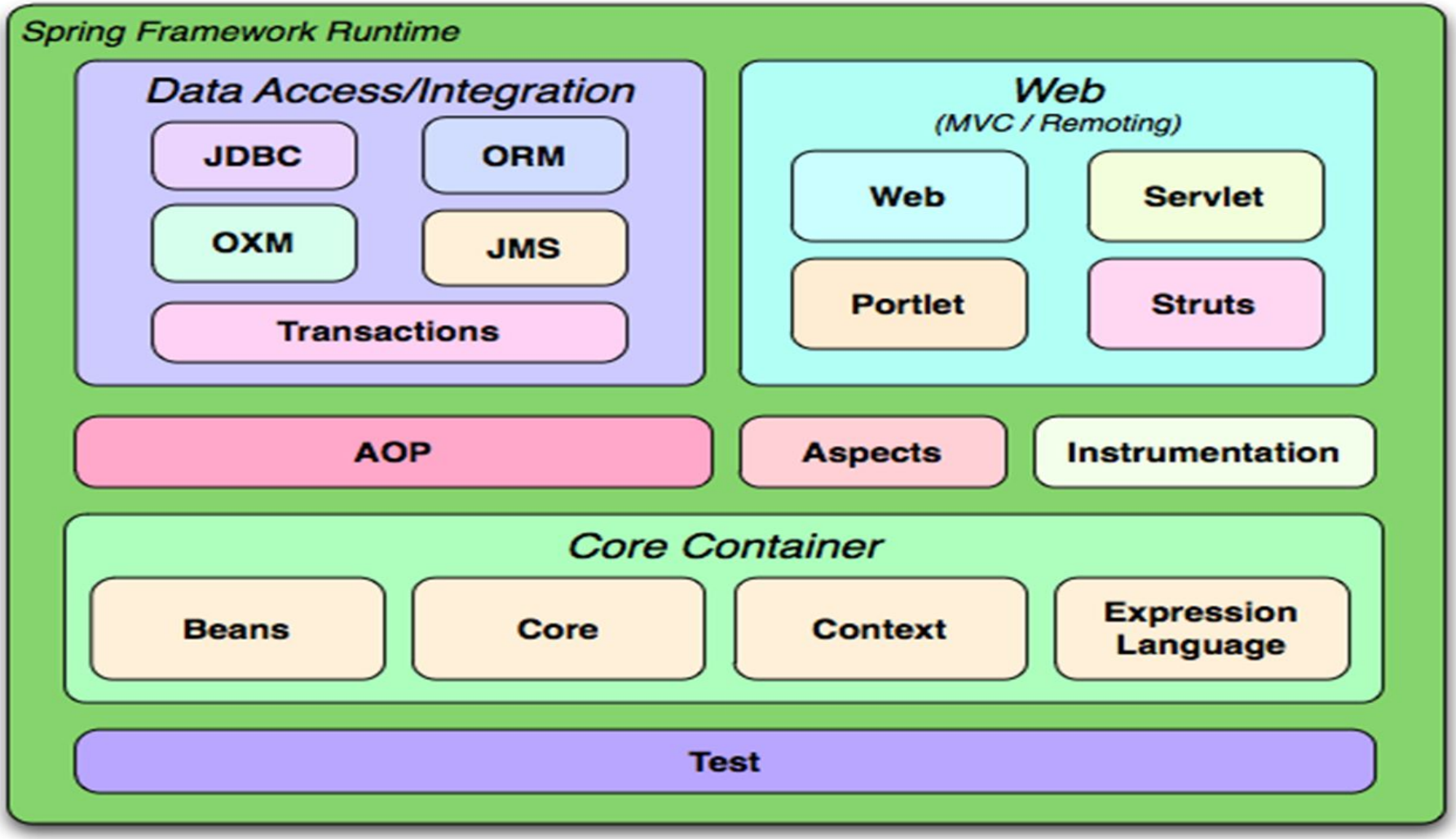


Why Spring is so popular?

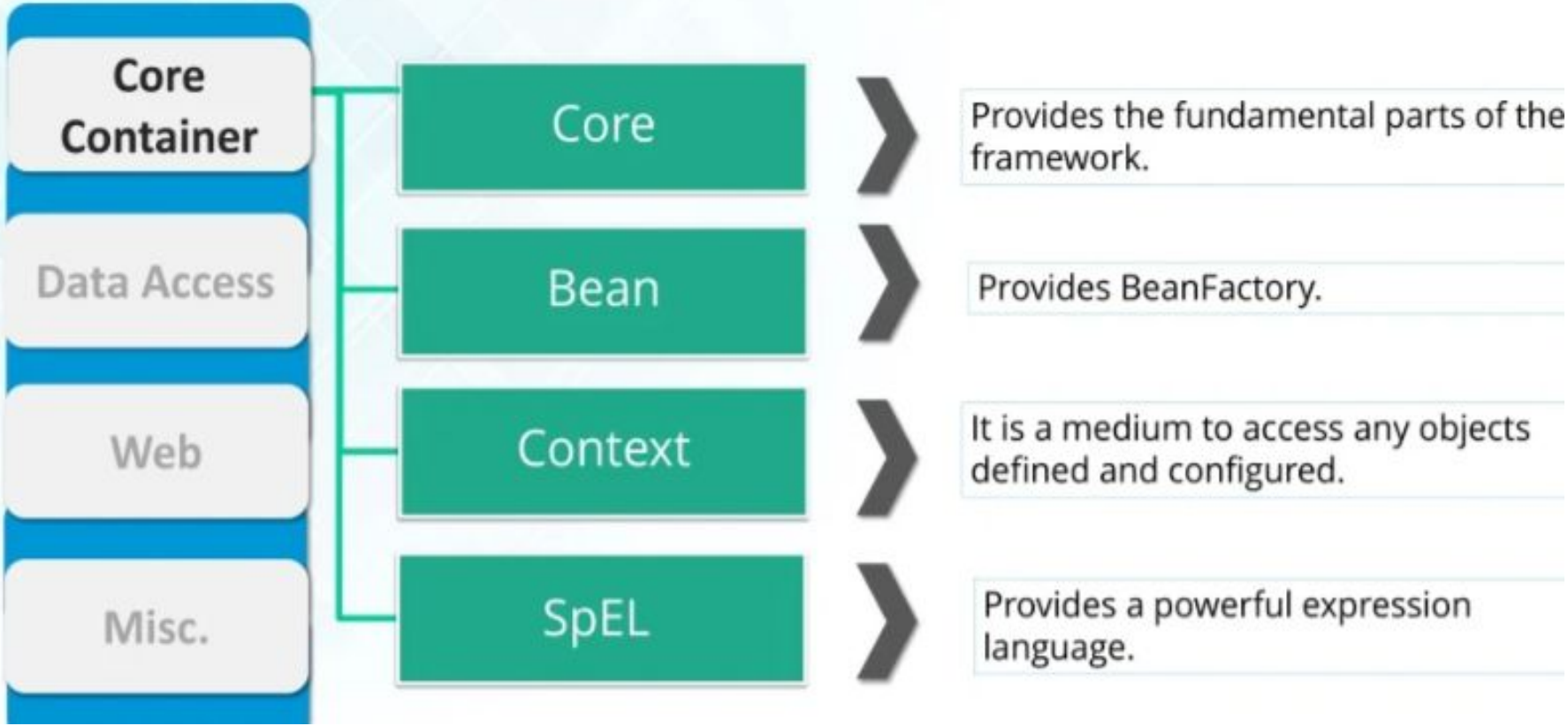
- Distinct division between JavaBean Models, Controllers and Views
- Spring's MVC is very flexible as it makes use of interfaces
- Spring's MVC web tiers are typically easier to test
- Well defined interface to business layer
- Spring Controllers are configured via IoC
- Offers better integration with view technologies other than JSP

Difference between Spring, Struts^2 and Hibernate

	Spring	Struts^2	Hibernate
Application Framework	✓	✗	✗
Light Weighted	✓	✗	✓
Layered Architecture	✓	✗	✓
Loose Coupling	✓	✗	✓
Tag Library	✗	✓	✓
Easy integration with ORM technologies	✓	✗	✓
Easy integration with client-side technologies	✗	✓	✗


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
(MVC / Remoting)
Web
Servlet
Portlet
Struts
AOP
Aspects
Instrumentation
Core Container
Beans
Core
Context
Expression Language
Test

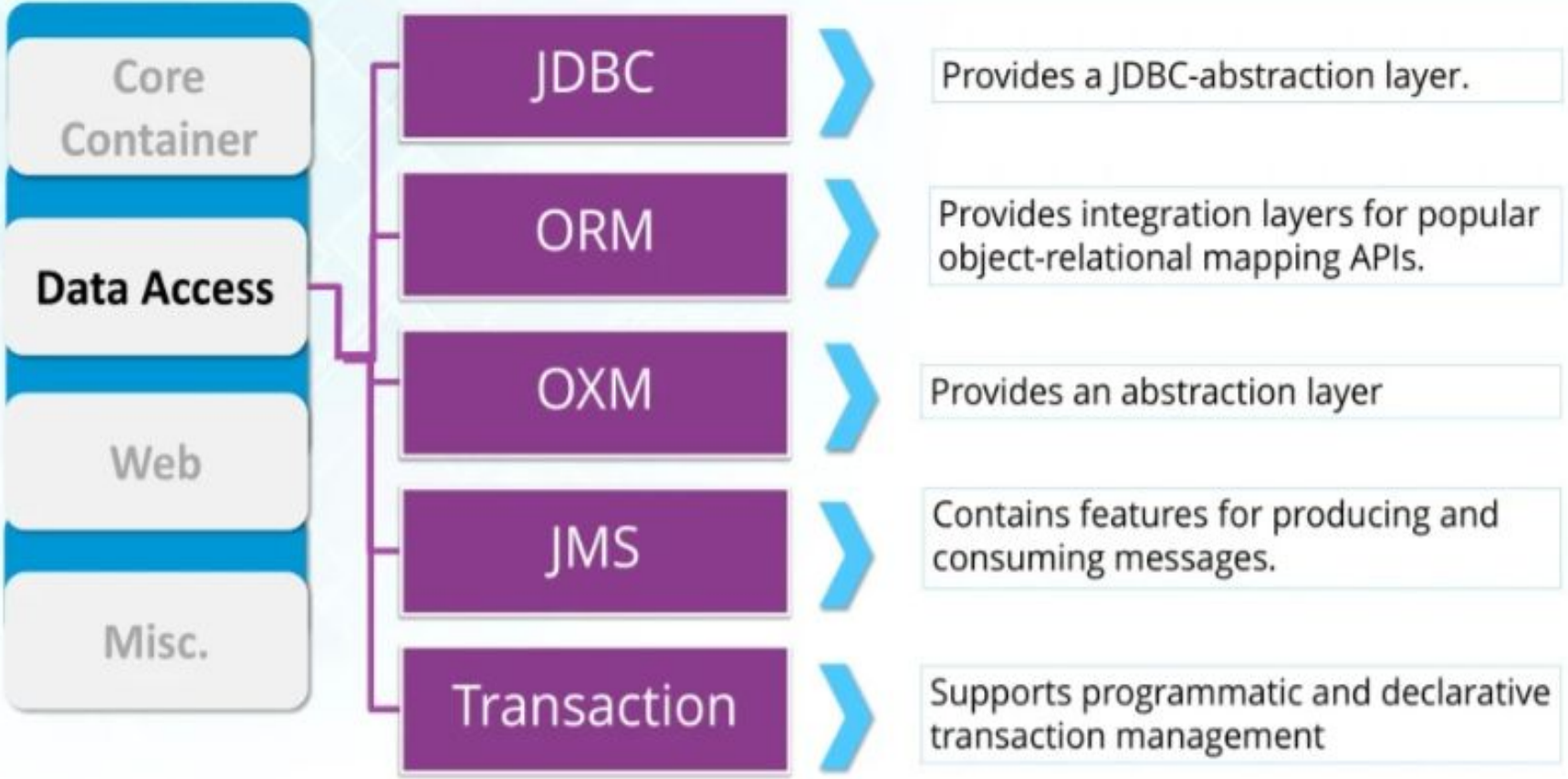
Core Module


Core Container
Data Access
Web
Misc.
Core
Bean
Context
SpEL
Provides the fundamental parts of the framework.
Provides BeanFactory.
It is a medium to access any objects defined and configured.
Provides a powerful expression language.

Core Container

- **The *Core and Beans* modules provides the IoC (Inversion of Control) and Dependency Injection features.** It **removes the need for programmatic singletons(only one instance of the class) and allows you to decouple the configuration and specification of dependencies** from your actual program logic. The BeanFactory is a sophisticated implementation of the factory pattern.
- The ***Context* module builds to access objects in a framework-style manner** that is similar to a JNDI (Java Naming and Directory Interface API) registry. The Context module **inherits its features from the Beans module and adds support for internationalization, event-propagation, resource-loading, and the transparent creation of context**. The Context module supports Java EE features such as EJB (Enterprise Java Bean), JMX (Java Management Extensions) ,and basic remoting. The ApplicationContext interface is the focal point of the Context module.
- The *Expression Language* module provides a **powerful expression language for querying and manipulating an object graph at runtime**. The language **supports setting and getting property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects** by name from Spring's IoC container. It also **supports list projection and selection** as well as common list aggregations.

Data Access/Integration



Data Access/Integration

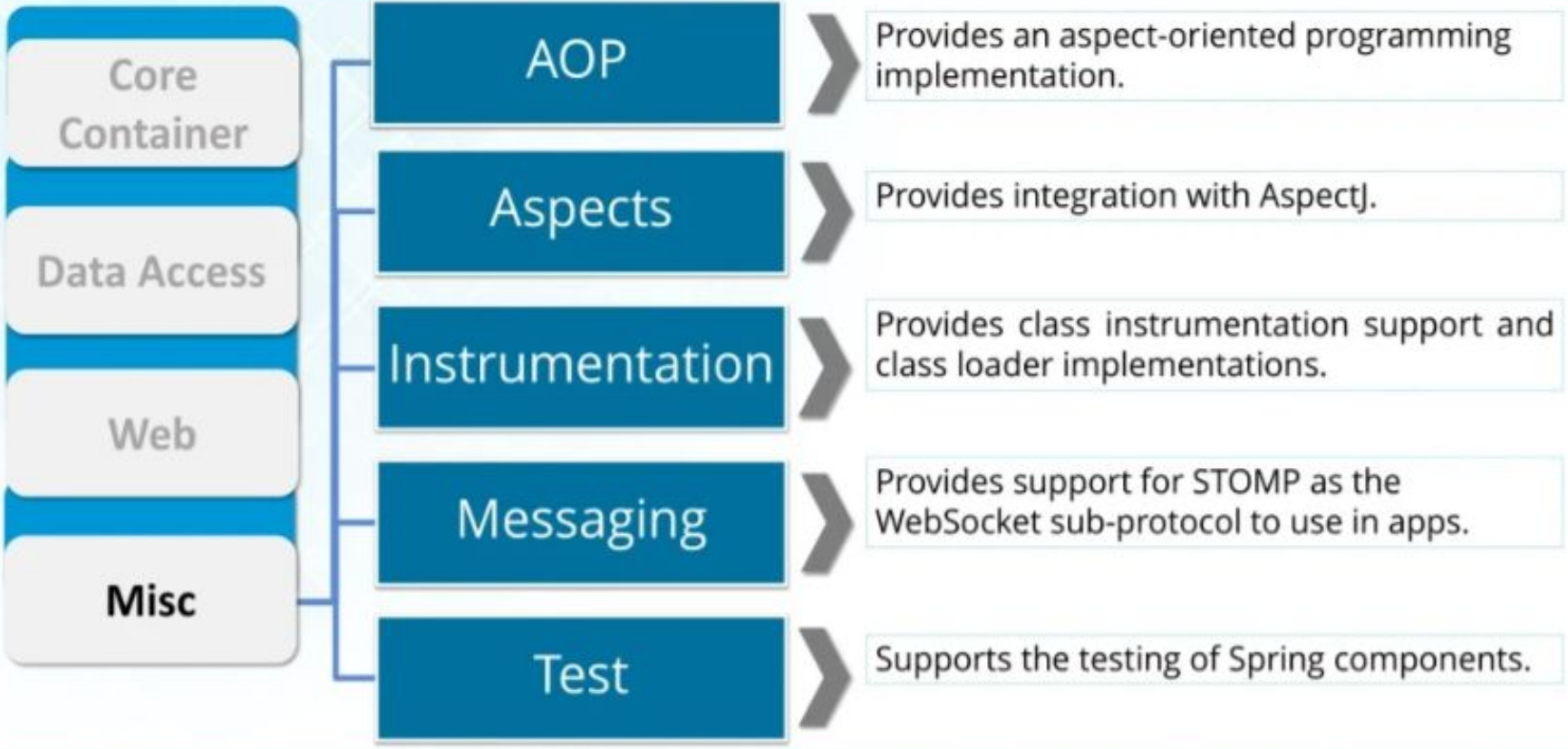
- The JDBC module **provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes**.
- The ***ORM*** **module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis**. Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.
- 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 (plain old Java objects)*.

Web(MVC/Remoting) Module

Web

- Spring's ***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-Servlet*** **module contains Spring's model-view-controller framework** provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.
- The ***Web-Struts*** **module contains the support classes** for integrating a classic Struts web tier within a Spring application.
- The ***Web-Portlet*** **module provides the MVC implementation** to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Miscellaneous

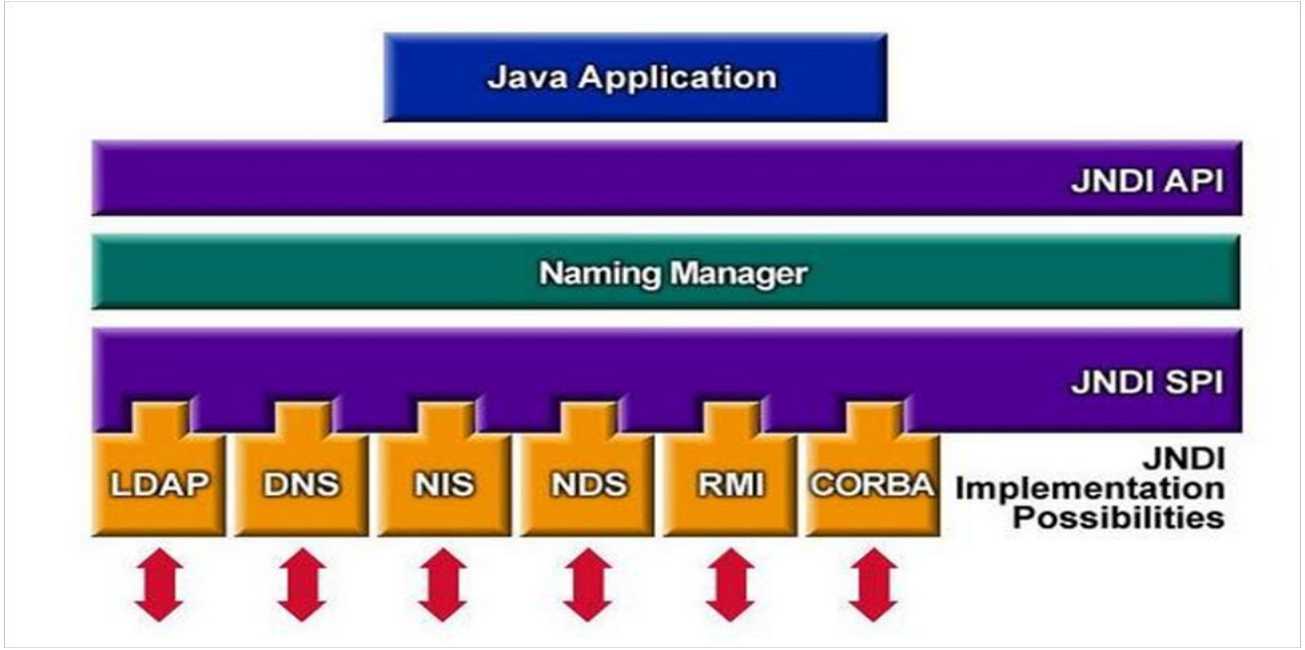


Test

- The *Test* module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

JNDI Registry

- JNDI is the acronym for the Java Naming and Directory Interface API. By making calls to this API, applications locate resources and other program objects. A resource is a program object that provides connections to systems, such as database servers and messaging systems.
- It **provides consistent use of naming and/or directory services as a Java API.** This interface can be used for binding objects, looking up or querying objects, as well as detecting changes on the same objects.
- Any work with JNDI requires **an understanding of the underlying service** as well as **an accessible implementation.** For example, a database connection service calls for specific properties and exception handling.
- However, JNDI's abstraction decouples the connection configuration from the application.



•The Java 2 SDK, v1.3 includes three service providers for the following naming/directory services:

1.Lightweight Directory Access Protocol (LDAP)

2.Common Object Request Broker Architecture (CORBA) Common Object Services (COS) name service

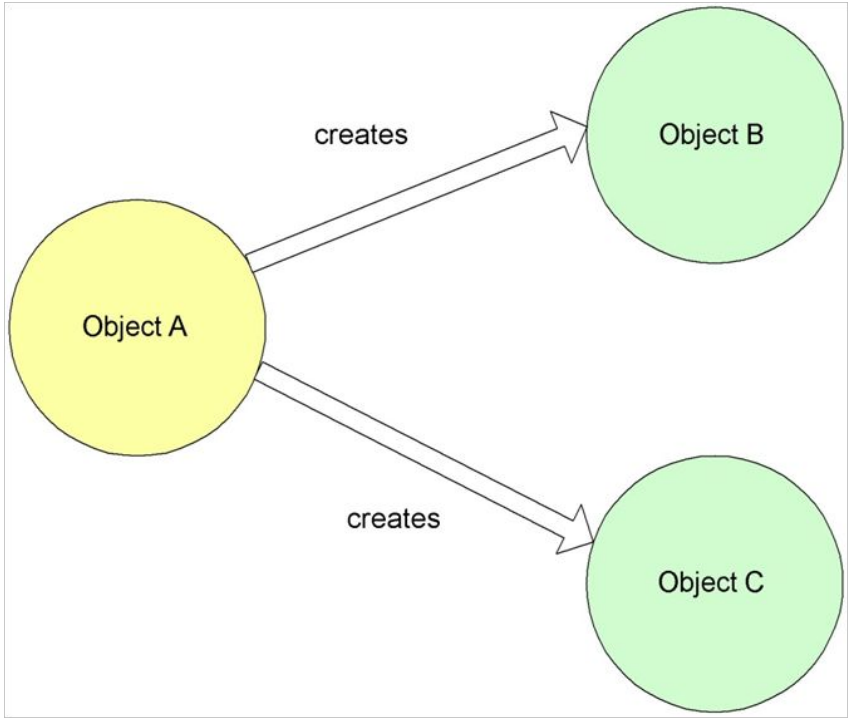
3.Java Remote Method Invocation (RMI) Registry

So basically you create objects and register them on the directory services which you can later do lookup and execute operation on.

Non-IoC / Dependency Injection

"Non-IoC" refers to a software architecture where classes are responsible for creating their own dependencies, rather than having those dependencies injected from an external source.

For example, a class might instantiate objects of other classes it needs to work with, or it might query a service locator for those objects. This means the class is tightly coupled to the implementation of its dependencies, making it difficult to switch to different implementations or test it in isolation.



Fundamentals Concepts

Spring largely built around

- Inversion of Control (IoC) or Dependency Management
- Aspect Oriented Programming (AOP)

IoC Container Features

Creating the objects

Wiring them together

Managing their complete life cycle

Configuring

Inversion of Control

•Instead of objects invoking other objects, the dependant objects are added through an external entity/container.

•Also known as the Hollywood principle – “don’t call me I will call you”

•Dependency injection

–Dependencies are “injected” by container during runtime.

–Beans define their dependencies through constructor arguments or properties

•Prevents hard-coded object creation and object/service lookup.

•Loose coupling

•Helps write effective unit tests.

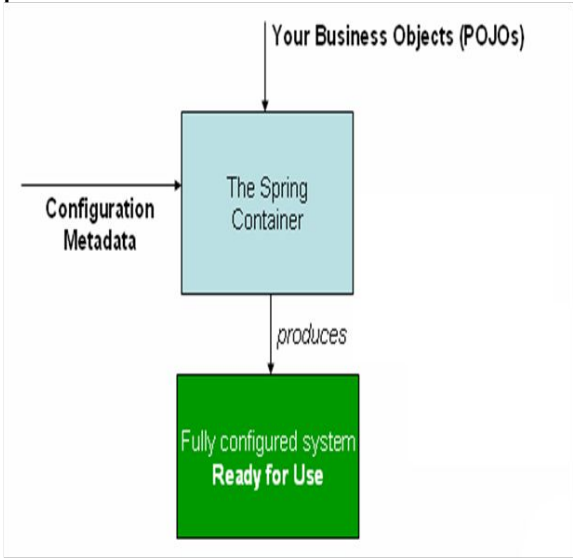
features of Spring IoC,

•Creating Object for us,

•Managing our objects,

•Helping our application to be configurable,

•Managing dependencies



```
// Java Program to Illustrate Sim Interface
public interface Sim
{
  void calling();
  void data();
}
```

```
// Java Program to Illustrate Airtel Class
public class Airtel implements Sim {
  @Override
public void calling()  {
  System.out.println("Airtel Calling");
  }
  @Override
public void data()  {
  System.out.println("Airtel Data");
  }
}
```

```
// Java Program to Illustrate Jio Class
public class Jio implements Sim{
  @Override
  public void calling() {
  System.out.println("Jio Calling");
  }
  @Override
  public void data() {
  System.out.println("Jio Data");
  }
}
```

beans.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
https://www.springframework.org/schema/beans/spring-beans.xsd">
<bean id="sim" class="Jio"></bean>
</beans>
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Mobile {
public static void main(String[] args) {
// Using ApplicationContext tom implement Spring IoC
ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml");
// Get the bean
Sim sim = applicationContext.getBean("sim", Sim.class);
// Calling the methods sim.calling(); sim.data();
}
}
```

Output:

Jio Calling Jio Data

The first step is to create an application context where we used framework API **ClassPathXmlApplicationContext()**. This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file.

The second step is used to get the required bean using **getBean()** method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.

```
<bean id="sim" class="Airtel"></bean>
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Mobile {
public static void main(String[] args) {
// Using ApplicationContext tom implement Spring IoC
ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml");
// Get the bean
Sim sim = applicationContext.getBean("sim", Sim.class);
// Calling the methods
sim.calling(); sim.data();
}
}
```

Output:

```
Airtel Calling Airtel Data
```

Dependency Injection

1. It is a design pattern which removes the dependency from the programming code, that makes the Application easy to manage and test.

2. Dependency Injection makes our programming code loosely coupled, which means change in implementation doesn't affects the user.

Types Of Dependency Injection

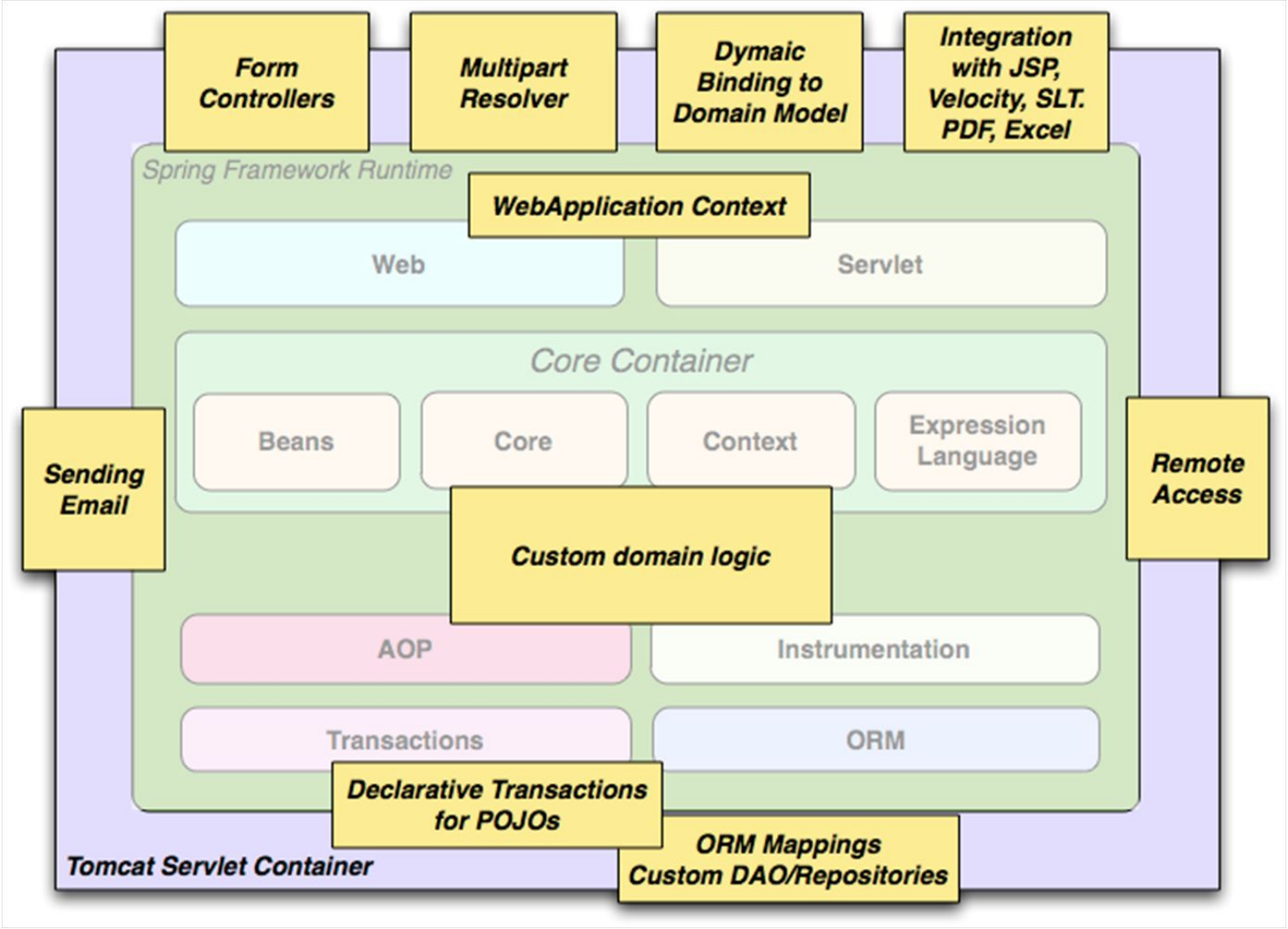
edu

Spring framework avails two ways to inject dependency :

By Constructor	1	The **<constructor-arg>** subelement of **<bean>** is used for constructor injection
By Setter method	2	The **<property>** subelement of **<bean>** is used for setter injection

Spring Usage Scenarios

- Following are the typical usage scenarios for Spring
  - Presentation layer
    - Integrates with Struts to initialize action classes and its dependencies.
    - Facilitates presentation-tier testing
  - Business layer
    - Integrates with EJBs
    - Provides integration with components using IoC.
    - Transaction (declarative and programmatic)
  - Persistence layer (for Data)
    - DAO pattern implementation
    - Template support for Hibernate, iBatis DataMapper and JDBC
    - Transaction management, Exception translation, connection management.
  - General
    - Email, JNDI, WebServices

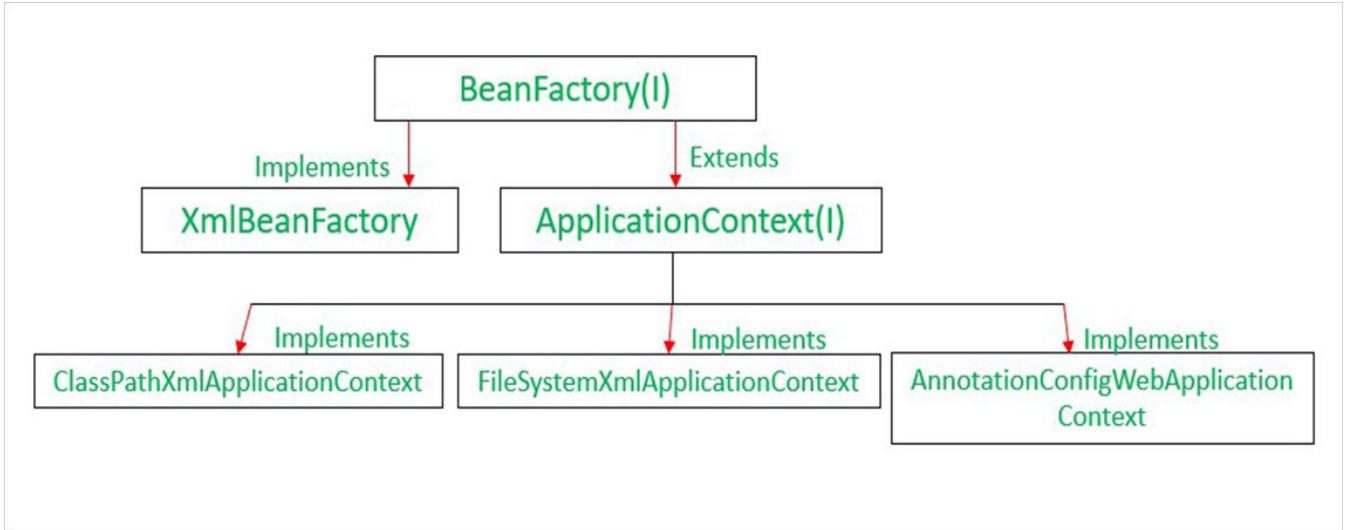

Form Controllers
Multipart Resolver
Dymaic Binding to Domain Model
Integration with JSP, Velocity, SLT. PDF, Excel
Spring Framework Runtime
WebApplication Context
Web
Servlet
Core Container
Beans
Core
Context
Expression Language
Sending Email
Remote Access
Custom domain logic
AOP
Instrumentation
Transactions
ORM
Declarative Transactions for POJOs
ORM Mappings Custom DAO/Repositories
Tomcat Servlet Container

Spring containers

•Spring containers are responsible for creating bean objects and injecting them into the classes. The two containers are namely,

•**BeanFactory(I)** – Available in org.springframework.beans.factory package.

•**ApplicationContext(I)** – Available in org.springframework.context package.



BeanFactory

- Spring BeanFactory Container is the simplest container which provides basic support for DI.
- It is defined by org.springframework.beans.factory.BeanFactory interface.
- There are many implementations of BeanFactory interface . The most commonly used BeanFactory implementation is –

  org.springframework.beans.factory.xml.XmlBeanFactory

- **Example-**

```
Resource resource=new ClassPathResource("Beans.xml");
BeanFactory factory=new XmlBeanFactory(resource);
```

- The Resource interface has many implementaions. Two mainly used are:

1)**org.springframework.core.io.FileSystemResource** :Loads the resource from underlying file system.

Example-

```
BeanFactory bfObj = new XmlBeanFactory(new FileSystemResource ("c:/beansconfig.xml"));
```

2)**org.springframework.core.io.ClassPathResource**:Loads the resource from classpath.

ApplicationContext

- The ApplicationContext container is Spring's advanced container.
- It is defined by org.springframework.context.ApplicationContext interface.
- The ApplicationContext interface is built on top of the BeanFactory interface.
- It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation etc.
- There are many implementations of ApplicationContext interface . The most commonly used ApplicationContext implementation is –

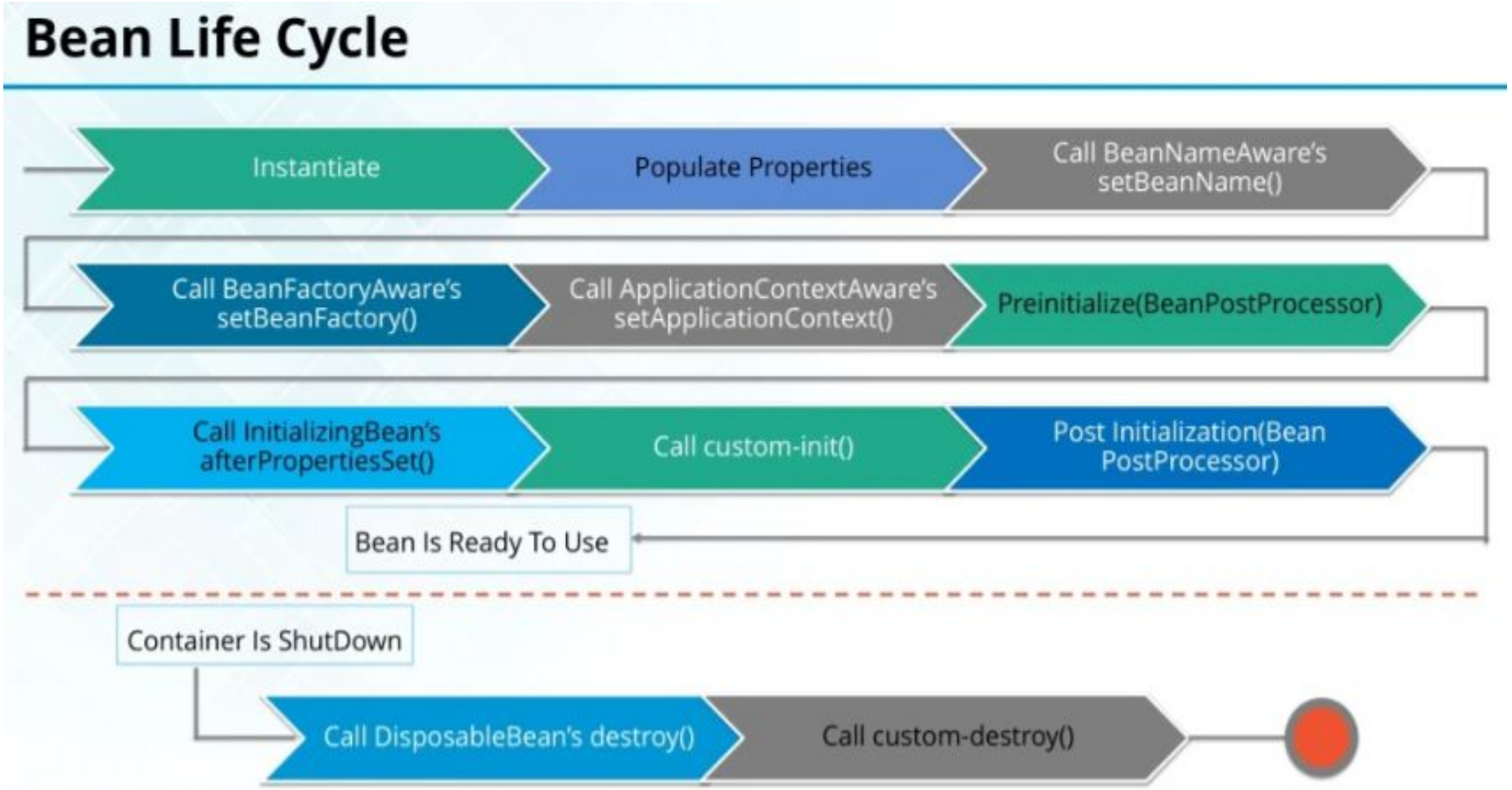
  org.springframework.context.support.ClassPathXmlApplicationContext

- **Example-**

```
ApplicationContext context=new ClassPathXmlApplicationContext("Beans.xml");
```

BeanFactory	ApplicationContext
It is a fundamental container that provides the basic functionality for managing beans.	It is an advanced container that extends the BeanFactory that provides all basic functionality and adds some advanced features.
It is suitable to build standalone applications.	It is suitable to build Web applications, integration with AOP modules, ORM and distributed applications.
It supports only Singleton and Prototype bean scopes.	It supports all types of bean scopes such as Singleton, Prototype, Request, Session etc.
It does not support Annotations. In Bean Autowiring, we need to configure the properties in XML file only.	It supports Annotation based configuration in Bean Autowiring.
This interface does not provides messaging (i18n or internationalization) functionality.	ApplicationContext interface extends MessageSource interface, thus it provides messaging functionality.

Bean Life Cycle



AOP Aspect-Oriented Programming

•An aspect is a class that **encapsulates behaviors affecting multiple classes or methods**, like logging, security, or transaction management.

Concerns: These refer to **specific functionalities or behaviors** in your application, like logging, data validation, or security.

A **crosscutting concern refers to a behavior or functionality in a program that is scattered and tangled over various system modules,** resulting in non-modularity. It can be homogeneous when the same behavior is replicated at multiple points or heterogeneous when the behavior at multiple points is different.

AOP entails breaking down program logic into distinct parts called so-called **concerns**.

•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.

AOP Aspect-Oriented Programming

•In AOP the unit of modularity is the aspect. **AOP is like triggers in programming** languages such as Perl, .NET, Java, and others.

•Spring AOP module provides interceptors to intercept an application. For example, when a method is executed, you can add extra functionality before or after the method execution.

•AOP is used in the Spring Framework to

–provide **declarative enterprise services**, especially as a replacement for EJB declarative services. The most important such service is ***declarative transaction management*.**

–allow users to implement **custom aspects**, complementing their use of OOP with AOP.

- **Problem without AOP**

We can call methods (that maintains log and sends notification) from the methods.

- But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

- **Solution with AOP**

We don't have to call methods from the method. Now we can define the additional **concern** like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

- In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.
- **AOP is mostly used in following cases:**
- to provide declarative enterprise services such as declarative transaction management.
- It allows users to implement custom aspects.

AOP Terminologies

- **Aspect :** This is a module which has a set of APIs providing **cross-cutting requirements**. An application can have any number of aspects depending on the requirement.
- **Join point:** This represents a point in your application where you can **plug-in the AOP aspect**. It is a point during the execution of a program, such as the **execution of a method or the handling of an exception**. In Spring AOP, a join point always represents a method execution.
- **Advice:** This is the **actual action to be taken either before or after the method execution**. This is an actual piece of code that is invoked during the program execution by Spring AOP framework.
- **Pointcut:** This is a **set of one or more join points where an advice should be executed**. You can specify pointcuts **using expressions or patterns** as we will see in our AOP examples.
- **Introduction:** declaring additional methods or fields on behalf of a type. Spring AOP allows you to **introduce new interfaces** (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a **bean implement an IsModified interface**, **to simplify caching**. (An introduction is known as an inter-type declaration in the AspectJ community.)
- **Target object:** The **object being advised by one or more aspects**. This object will always be a proxied object, also referred to as the advised object.
- **Weaving:** Weaving is the **process of linking aspects with other application types or objects** to create an advised object. . This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

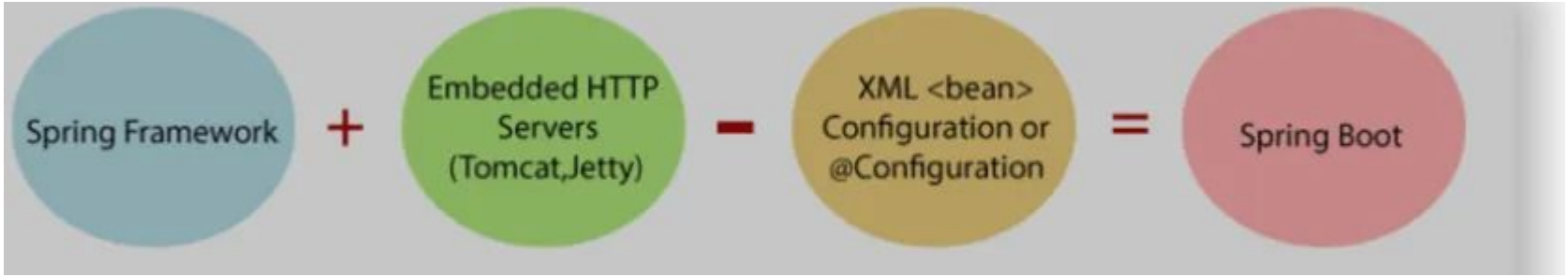
Types of Advice

- **Before Advice**: These advices runs before the execution of join point methods. We can use @Before annotation to mark an advice type as Before advice.
- **After (finally) Advice**: An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using @After annotation.
- **After Returning Advice**: Sometimes we want advice methods to execute only if the join point method executes normally. We can use @AfterReturning annotation to mark a method as after returning advice.
- **After Throwing Advice**: This advice gets executed only when join point method throws exception, we can use it to rollback the transaction declaratively. We use @AfterThrowing annotation for this type of advice.
- **Around Advice**: This is the most important and powerful advice. This advice surrounds the join point method and we can also choose whether to execute the join point method or not. We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning

Spring Boot

Spring Boot is a project that is **built on top of the Spring Framework**. It provides an **easier and faster way to set up, configure, and run both simple and web-based applications**.

It is a Spring module that provides the **RAD (Rapid Application Development)** feature to the Spring Framework used to create a **stand-alone Spring-based application** that you can just run because it needs minimal Spring configuration.



How did Spring Boot originate?

The Spring framework has gained the interest of the Java developer's community as a **lightweight alternative to EJB** and became one of the most popular application development frameworks.

It consists of a **large number of modules providing a range of services** including a component container, ASP support, security framework, data access framework, web application framework, support classes for testing components and etc.

Collecting all the required spring components together, setting up library dependencies in gradle/maven, and then configuring the required spring beans using xml, annotations, or java code requires quite an effort within the Spring framework.

Spring boot is already **a launchable app**.

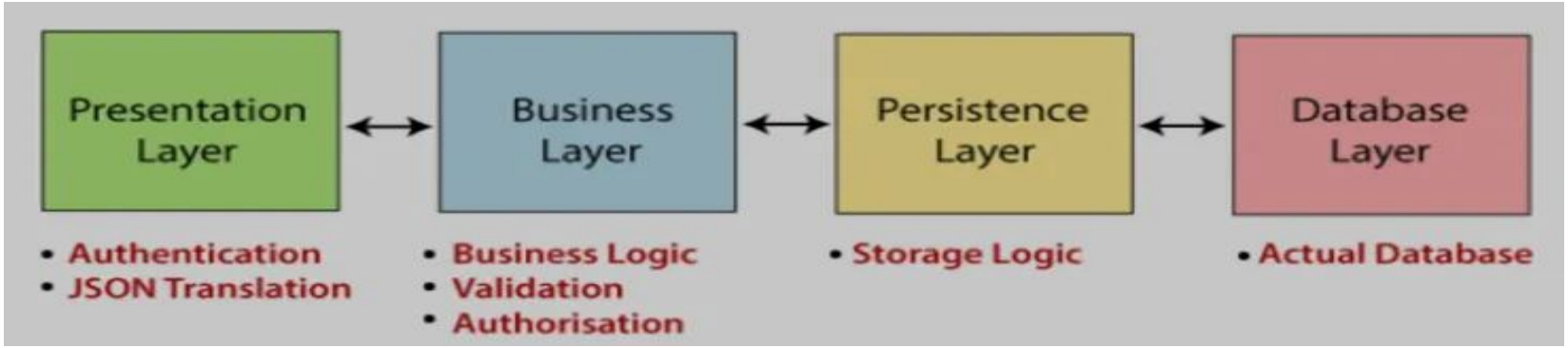
Spring Boot Architecture

In Spring Boot, there is **no requirement for XML configuration (deployment descriptor)**. It **uses convention over configuration software design paradigm** which means that it decreases the effort of the developer.

The main goal of Spring Boot is **to reduce development, unit test, and integration test time** and leveraging the following features:

- Create stand-alone Spring applications
- Embed Tomcat, Jetty, or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' POMs to simplify your Maven configuration
- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks, and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

Spring Boot follows a layered architecture in which **each layer communicates with the layer directly below or above (hierarchical structure)** it.



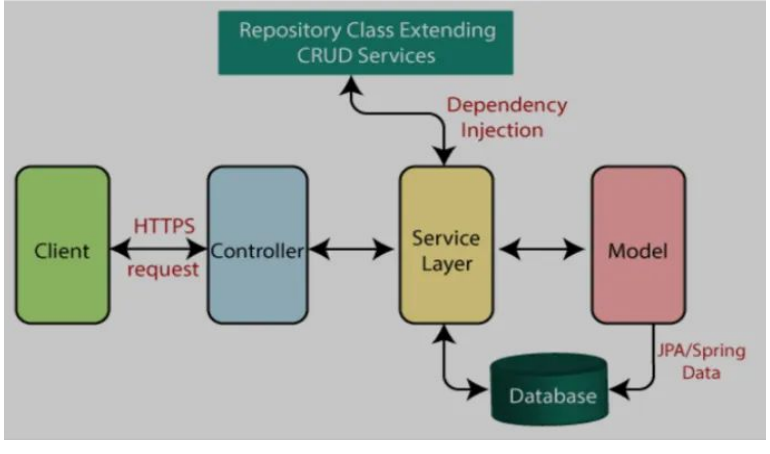
Spring Boot Architecture

- **Presentation Layer:** The presentation layer **handles the HTTP requests, translates the JSON parameter to object, and authenticates the request, and transfer it to the business** layer. In short, it consists of views.
- **Business Layer:** The business layer **handles all the business logic. It consists of service classes and uses services provided by data access layers**. It also performs authorization and validation.
- **Persistence Layer:** The persistence layer **contains all the storage logic and translates business objects** from and to database rows.
- **Database Layer:** In the database layer, **CRUD (create, retrieve, update, delete) operations are performed.**

Spring Boot Architecture

Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc. The architecture of Spring Boot is the same as the architecture of Spring MVC, except for one thing: **there is no need for DAO and DAOImpl classes in Spring boot.**

- Data access layer gets created and CRUD operations are performed.
- The client makes the HTTP requests.
- The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.
- In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.
- A response page is returned to the user if no error occurs



Spring Boot Components

- Spring Boot Starters
- Automatic configuration
- Spring Boot CLI
- Spring Boot Actuator

Spring Boot Starters

Spring Boot Starters are **dependency descriptors** that can be added under the <dependencies> section in pom.xml. There are around 50+ Spring Boot Starters for different Spring and related technologies. These starters give all the dependencies under a single name.

Advantages of Spring Boot Starters

- Increase productivity by decreasing the Configuration time for developers.
- Managing the POM is easier since the number of dependencies to be added is decreased.
- Tested, Production-ready, and supported dependency configurations.
- No need to remember the name and version of the dependencies.

Spring Boot Starter Data JPA is illustrated below:

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
</dependencies>
```

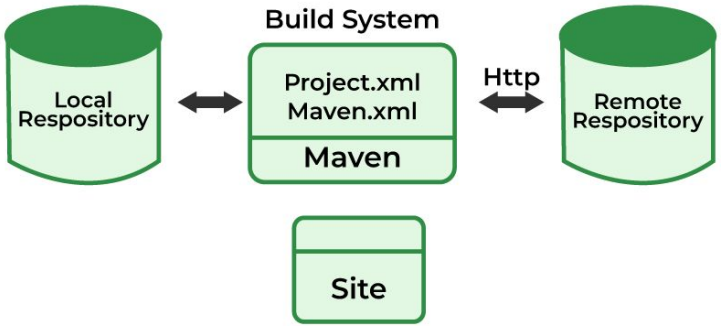
pom.xml

The pom.xml file, short for **Project Object Model**, is fundamental to Maven-based Spring Boot projects. It's an XML file containing **project metadata and configurations needed by Maven to build, manage, and deploy the application**.

<modelVersion>: Specifies the POM model version, typically 4.0.0.
<groupId>: Identifies the project's group or organization.
<artifactId>: Defines the project's unique name or identifier.
<version>: Sets the project's version number.
<packaging>: Specifies the project's output type (e.g., jar, war).
<name>: A descriptive name for the project.
<description>: Provides a brief description of the project.
<parent>: Establishes inheritance from a parent POM which manages dependencies and configurations.
<properties>: Defines project-specific properties, such as Java version.
<dependencies>: Declares project dependencies on external libraries/modules including Spring Boot starters.
<build>: Configures the build process, including plugins for creating executable JARs.
<plugins>: Specifies Maven plugins to extend build functionality.

Maven-based Spring Boot projects

Maven is a powerful build automation and project management tool primarily used for Java projects, but also applicable to other languages like C#, Ruby, and Scala.
It simplifies the development process by **automating tasks like compiling, testing, packaging, and dependency management**.
Essentially, Maven helps manage projects, their dependencies, and build processes, making it a vital tool for Java developers.



Spring Boot – Auto-configuration

- For example, If we want to use Spring MVC, we need to use @ComponentScan annotation, Dispatcher servlet, view resolver, web jars, etc. This kind of configuration makes it slow to develop an application. So, in this place, Spring Boot Autoconfiguration comes in. It looks at what types of frameworks are available at the classpath and it looks at what configurations are provided by the programmers or what configurations are provided already for the application. It will look at both of them. Data is not configured but there is hibernation on the classpath, so it will configure the data source automatically. It will configure the in-memory database, it will configure the dispatcher servlet automatically. This is called autoconfiguration.
- @Conditional annotation acts as a base for the Spring Boot auto-configuration annotation extensions.
- It automatically registers the beans with @Component, @Configuration, @Bean, and meta-annotations for building custom stereotype annotations, etc.
- The annotation @EnableAutoConfiguration is used to enable the auto-configuration feature.
- The @EnableAutoConfiguration annotation enables the auto-configuration of Spring ApplicationContext by scanning the classpath components and registering the beans.
- This annotation is wrapped inside the @SpringBootApplication annotation along with @ComponentScan and @SpringBootConfiguration annotations.
- When running main() method, this annotation initiates auto-configuration.

How Does Spring Boot Apply Auto-Configuration?

Step 1: Enabling Auto-Configuration

The `@SpringBootApplication` annotation implicitly includes `@EnableAutoConfiguration`, which is the entry point for triggering the auto-configuration process. This annotation triggers the loading of auto-configuration classes defined in `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`.

Step 2: Loading Configuration Classes

During application startup, Spring Boot retrieves the configuration classes listed in the `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file. Each class is evaluated to determine if it meets the conditions specified by its annotations. **Configuration classes are loaded in a specific order determined by annotations such as `@AutoConfigureOrder`, `@AutoConfigureBefore`, and `@AutoConfigureAfter`.** These annotations prioritize the initialization of foundational components like data sources and web servers before higher-level components, such as controllers, to prevent dependency-related issues.

Step 3: Applying Conditional Logic

Each configuration class is processed, and Spring Boot evaluates the conditions defined by its annotations. If all conditions are met, the corresponding beans are registered in the application context. For instance:

- If `DataSourceAutoConfiguration` is active, Spring Boot creates a `DataSource` bean using properties defined in `application.properties` or `application.yml`.
- If no database driver is present on the classpath, `DataSourceAutoConfiguration` is skipped entirely.

Step 4: Resolving Bean Definitions

Auto-configuration applies conditions to register beans dynamically. For example:

- If a user explicitly defines a DataSource bean in their application, Spring Boot detects its presence and skips the default auto-configuration for DataSource. However, overriding default beans may lead to dependency injection issues if not properly aligned with the application's requirements.

Spring Boot CLI

- Spring Boot also provides a **command-line tool** that can be used to quickly write Spring applications. CLI is smart; **it detects classes being used in your application** and it also **knows which Starter dependencies should be used for these classes**; accordingly, Spring Boot CLI adds dependencies to the classpath to make it work.
  As **Spring Boot CLI adds dependencies**, a series of auto-configuration kicks in and adds the required bean method configuration so that your application is able to respond to HTTP requests.

Common Spring Boot CLI commands include:

- spring run: Runs a Groovy script or a Java-based Spring Boot application.
- spring init: Initializes a new project using Spring Initializr.
- spring shell: Launches an integrated shell for interacting with the application.
- spring help: Displays help information for available commands.
- spring --version: Shows the installed Spring Boot CLI version.

Spring Boot Actuator

It also provides **post-production features**. This allows you to **monitor your Spring application during production using HTTP endpoints or with JMX**.
The Actuator instead offers the ability to **inspect the internals of your application at runtime**.
The Actuator provides data on auditing, metrics, and the health of your Spring Boot application using HTTP endpoints or with JMX. It helps you to manage your application when it's in production.

- It provides **details of all beans configured** in the Spring application context
- It also **ensures all environment variables, system properties, configuration properties, and command-line arguments** are available to your application
- The Actuator gives **various metrics pertaining to memory usage, garbage collection, web requests, and data source usage**
- It provides **a trace of recent HTTP requests** handled by your application
- It also gives **information about the current state of the threads** in the Spring Boot application

Spring Boot Actuator

Advantages

- **Simplified & version conflict-free dependency management** through the starter POMs.
- We can quickly **set up and run standalone, web applications, and microservices**.
- **Reduces the time spent on development and increases the overall efficiency** of the development team.
- Helps to **avoid all the manual work of writing boilerplate code, annotations, and complex** XML configurations. The beans are initialized, configured, and wired automatically.
- Provides **many plugins that developers can use to work with embedded and in-memory databases smoothly and readily**.
- **Allows for easy connection with database and queue services** like Oracle, PostgreSQL, MySQL, MongoDB, Redis, Solr, ElasticSearch, Rabbit MQ, ActiveMQ, and many more.

Disadvantages

- Spring boot **may unnecessarily increase the deployment binary size with unused dependencies.**
- Spring Boot sticks well with microservices. The Spring Boot artifacts can be deployed directly into Docker containers. In **large and monolithic applications it is not encouraged much.**

Basis	Spring	Spring Boot
Where it's used?	Spring framework is a java EE framework that is used to build applications.	Spring Boot framework is mainly used to develop REST API's
Key feature	The primary or most important feature of the Spring framework is dependency injection(Dependency Injection (DI) is a design technique that removes dependencies from computer code, making the application easier to maintain and test).	The main or primary feature of the Spring Boot is Autoconfiguration(Simply described, Spring Boot autoconfiguration is a method of automatically configuring a Spring application based on the dependencies found on the classpath.) Autoconfiguration can speed up and simplify development by removing the need to define some beans that are part of the auto-configuration classes.
Why it's used	Its goal is to make Java EE (Enterprise Edition) development easier, allowing developers to be more productive.	Spring Boot provides the RAD(Rapid Application Development) feature to the Spring framework for faster application development.
Type of Application Development	Spring framework helps to create a loosely coupled application.	Spring Boot helps to create a stand-alone application.

Basis	Spring	Spring Boot
Servers dependency	In the Spring framework to test the Spring Project, we need to set up the servers explicitly.	Spring Boot offers built-in or embedded servers such as Tomcat and jetty.
Deployment descriptor	To run a Spring application a deployment descriptor is required.	In Spring Boot there is no need for the Deployment descriptor.
In-memory database support	Spring framework does not provide support for the in-memory database.	Spring Boot provides support for the in-memory database such as H2.
Boilerplate code	Spring framework requires too many lines of code (boilerplate code) even for minimal tasks.	You avoid boilerplate code which reduces time and increases productivity.
Configurations	In the Spring framework, you have to build configurations manually.	In Spring Boot there are default configurations that allow faster bootstrapping.
Dependencies	Spring Framework requires a number of dependencies to create a web app.	Spring Boot, on the other hand, can get an application working with just one dependency. There are several more dependencies required during build time that is added to the final archive by default.

Basis	Spring	Spring Boot
HTTP Authentication	HTTP Basic Authentication is for enabling security confirmations, it indicates that several dependencies and configurations need to be enabled to enable security. Spring requires both the standard spring-security-web and spring-security-config dependencies to set up security in an application. Next, we need to add a class that extends the WebSecurityConfigurerAdapter and makes use of the @EnableWebSecurity annotation.	Spring Boot also requires these dependencies to make it work, but we only need to define the dependency of spring-boot-starter-security as this will automatically add all the relevant dependencies to the classpath.
Testing	Testing in Spring Boot is difficult in comparison to Spring Boot due to a large amount of source code.	Testing in Spring Boot is easier due to the reduced amount of source code.
XML Configuration	In the Spring framework, XML Configuration is required.	No need for XML configuration in Spring Boot.
CLI Tools	Spring framework does not provide any CLI tool for developing and testing applications.	Spring Boot provides a CLI tool for developing and testing Spring Boot applications.
Plugins	Spring framework does not provide any plugin for maven, Gradle, etc. like Spring Boot.	Spring Boot provides build tool plugins for Maven and Gradle. The Plugins offer a variety of features, including the packaging of executable jars.

CRUD App in 30 mins | Simplest Explanation | Spring Boot | REST | JPA | H2 | Tutorials for Beginners

https://www.youtube.com/watch?v=ZZTYQIUd_uY

What are the main features of Spring?

The most fundamental aspect of Spring and Spring Boot is Dependency Injection (DI) or Inversion of Control (IoC), the cornerstone of every Spring Module. We can create loosely coupled applications that can be easily tested and maintained using these design patterns. The Spring framework also includes several out-of-the-box modules, namely:

- Spring MVC
- Spring Security
- Spring ORM
- Spring Test
- Spring AOP
- Spring Web Flow
- Spring JDBC.

Spring Use Cases:

- Developing serverless applications
- Building scalable microservices
- Securing the server-side of your application
- Asynchronous application development
- Automating tasks by creating batches

What are the main features of Spring Boot?

- Embedded server eliminates the need for complex application development
- Starter dependencies that facilitate building and configuring apps
- Automated Spring configuration
- Metrics, health check, and other reports
- Everything in Spring Boot is pre-configured. We simply need to use the proper configuration to use a specific functionality. If we want to create a REST API, we can use Spring Boot.

Advantages of Spring Boot

- You can use it to create standalone applications
- There is no need to deploy WAR files while using SpringBoot
- It doesn't require XML configuration
- Embeds Tomcat, Jetty and Undertow directly
- Offers production-ready features
- SpringBoot is easier to launch
- Easier customization and management

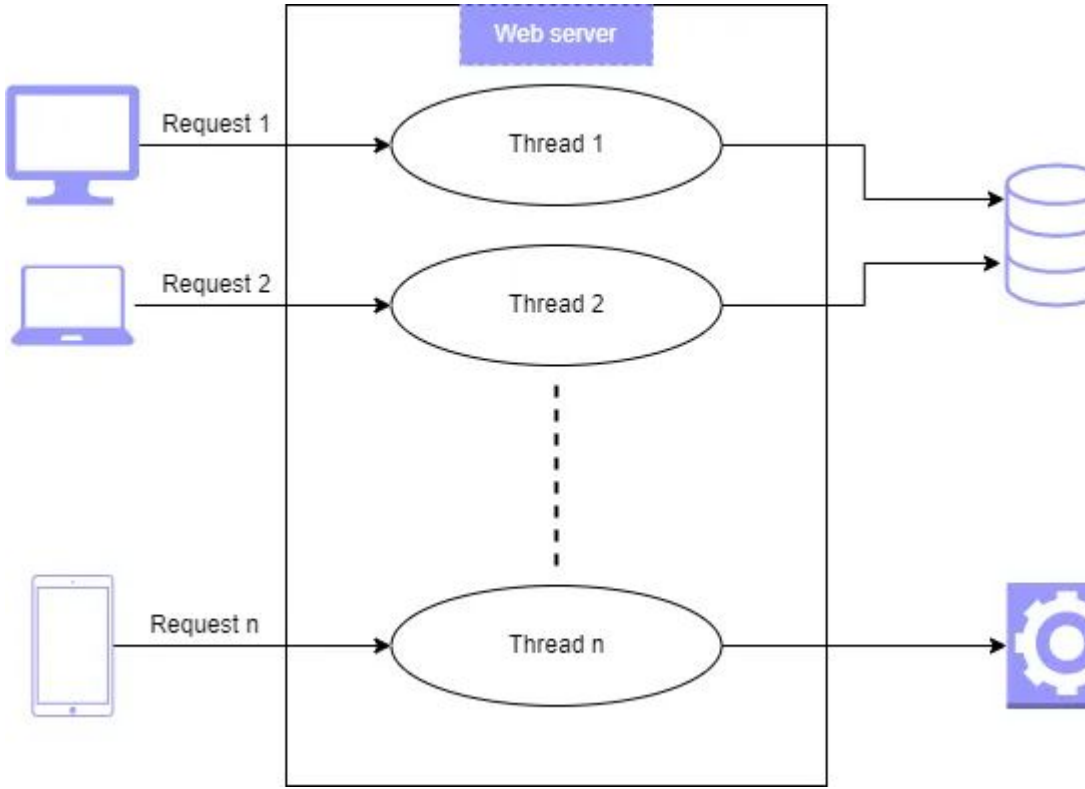
Drawbacks of Spring Boot:

- Spring Boot generates a large number of unused dependencies, which results in a large deployment file.
- Complex process
- The difficult and time-consuming process of converting a legacy or existing Spring project to a Spring Boot application.
- Limited suitability
- Although Spring Boot is ideal for working with microservices, many developers believe it is not appropriate for building large-scale applications.

How does Spring Boot solve the main issues of Spring?

- When compared with Spring Vs Spring Boot, Spring Boot is an update to the Spring framework that enables developers to build stand-alone and integrated web applications in the Java language.
- Spring Boot improves Spring functionality by streamlining the development process and providing external configurations for services such as MongoDB, MySQL, and Oracle databases. Additional features include integrating external libraries such as Amazon Web Services (AWS), messengers, etc.
- Assembled on top of Spring, The magic of Spring Boot Auto-Configuration is that it automatically creates configurations for your application that would otherwise be very laborious to set up. This is done by automatically assigning a value to different components of the application when these components are detected on the classpath.
- Spring Boot also has its own set of annotations that were created from a combination of Spring's annotations, making it simpler and easier to use popular web frameworks in our applications.

Traditional way



there is a maximum thread count for a particular web server and that count may be based on the web server. If we use spring-boot, the webserver container will be tomcat and that container will have 200 maximum threads.

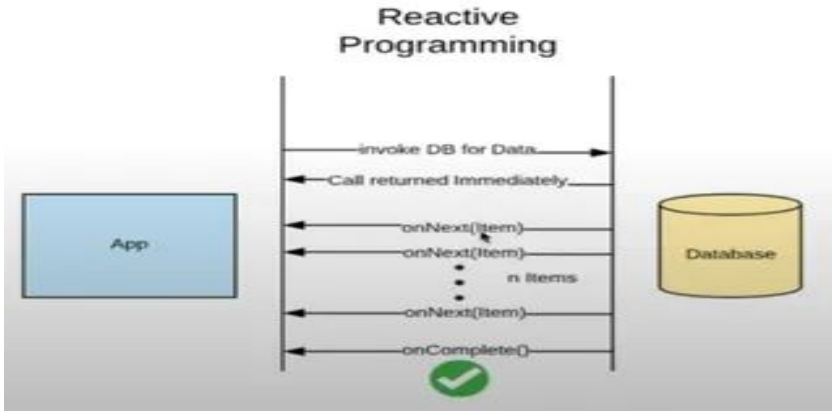
Drawbacks in the traditional way

- Once a **thread is assigned to a request that thread won't be available** until that request finishes its process.
- If all the threads are occupied, **the next requests that come into the server** will have to wait until at least one thread frees up.
- **When all the threads are busy, performance will be degraded** because memory is being used by all the threads.

Reactive programming

Reactive programming is a programming paradigm where the focus is on developing asynchronous and non-blocking applications in an event-driven form

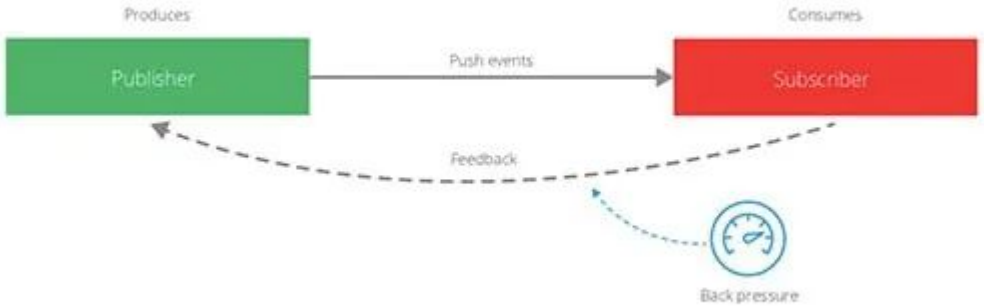
- **Asynchronous and non-blocking :** Asynchronous execution is a way of executing code w**ithout the top-down flow of the code**. In a synchronous way, if there is a blocking call, like calling a database or calling a 3rd party API, the execution flow will be blocked. But in a non-blocking and asynchronous way, the execution flow will not be blocked. Rather than that, futures and callbacks will be used in asynchronous code execution.
- **Event/Message Driven stream data flow:** In reactive programming, data will flow like a stream and because it is reactive, **there will be an event and a response message to that event**. In Java, it is similar to java streams which was introduced in Java 1.8. In the traditional way, when we get data from a data source (Eg: database, API), all the data will be fetched at once. In an event-driven stream, **the data will be fetched one by one and it will be fetched as an event to the consumer.**



Reactive programming

Reactive programming is a programming paradigm where the focus is on developing asynchronous and non-blocking applications in an event-driven form

- **Functional style code**
  In Java, we write lambda expressions for Functional Programming. **Lambda expressions are functional style codes**. In reactive programming, we mostly use this lambda expression style.
- **Back Pressure**
  In reactive streams **when a reactive application (Consumer) is consuming data from the Producer, the producer will publish data to the application continuously as a stream**. Sometimes the application cannot process the data at the speed of the producer. In this case, the consumer can notify the producer to slow down the data publishing.



Need for Java Reactive Programming

Reactive programming in Java can be **beneficial for building modern, responsive, and scalable applications**. Its **ability to handle asynchronous and non-blocking code**. This **allows developers to write code that can efficiently handle concurret tasks without blocking the main thread of execution, leading to improved performance and responsiveness in applications**.
Another benefit of reactive programming is **its focus on composability and the transformation of data**. Reactive programming provides a **stream-based processing model, where data flows in a continuous stream of events**.
This enables developers to easily compose and transform data in a declarative and concise manner, allowing for efficient data processing and manipulation. This can be particularly useful in scenarios where data **needs to be processed in real-time, such as in streaming data analytics or event-driven applications**.

Usage of Java Reactive Programming

- **Asynchronous and non-blocking:** It allows you to write asynchronous code that can handle multiple tasks concurrently without blocking the main thread of execution. This can result in more responsive and scalable applications that can handle a large number of concurrent requests and perform better under high loads.
- **Scalability:** It leverages the power of multi-core processors and other hardware advancements to enable scalable processing of large amounts of data and complex operations in parallel. This can help your application scale horizontally and efficiently utilize system resources, making it suitable for handling high-traffic and high-concurrency scenarios.
- **Responsiveness:** It promotes an event-driven approach, where your application can respond to events and data streams in real-time. This can enable real-time data processing, event-driven architectures, and interactive user interfaces that provide a more engaging user experience.
- **Flexibility:** It provides a flexible and composable programming model, where you can easily compose and transform streams of data using operators, filters, and transformations. This makes it easier to handle complex data processing scenarios and adapt to changing requirements.
- **Error handling:** It provides built-in error handling mechanisms, such as error channels and error recovery operators, which make it easier to handle errors and failures in a more controlled and graceful manner. This can help improve the fault tolerance and resilience of application

Disadvantages of Java Reactive Programming

- **Steeper learning curve:** It requires developers to adopt a new mindset and learn new concepts, such as reactive streams, operators, and backpressure. This can involve a learning curve, especially for developers who are not familiar with reactive programming concepts or functional programming paradigms.
- **Increased complexity: It** can introduce additional complexity to the codebase due to the need to manage and compose streams of data, handle backpressure, and understand the behavior of reactive operators. This can make the codebase harder to understand, debug, and maintain, especially for complex applications.
- **Debugging challenges:** Debugging reactive code can be more challenging than traditional imperative code, as it involves streams of data flowing asynchronously and concurrently. This can make it harder to identify and fix issues related to data flow, sequencing, and error handling.
- **Overuse of reactive patterns:** In some cases, developers may be tempted to use reactive patterns excessively, even when they may not be the best fit for a particular use case. This can result in overly complex code and unnecessary overhead, leading to decreased performance and maintainability.
- **Tooling and library support:** Although there are many popular reactive programming libraries and frameworks available for Java, the tooling and ecosystem may not be as mature or comprehensive as traditional imperative programming. This can result in potential limitations or gaps in terms of available libraries, documentation, and community support.

```
import reactor.core.publisher.Flux;
import reactor.core.scheduler.Schedulers;
public class ReactiveProgrammingExample {
public static void main(String[] args)
{
// Create a Flux of integers from 1 to 10
Flux<Integer> flux = Flux.range(1, 10);
// Use reactive operators to  transform and process the data Filter out odd numbers
flux.filter(i -> i % 2 == 0)
// Double the remaining numbers
.map(i -> i * 2)
// Publish on parallel scheduler for concurrent execution
.publishOn(Schedulers.parallel())
// Subscribe to the final data stream and print the results
.subscribe(System.out::println);
// Wait for a moment to allow the async processing to complete
try {
        Thread.sleep(1000);
}
catch (InterruptedException e) {
        e.printStackTrace();
} } }
```

In this example, we create a Flux (a reactive stream) of integers from 1 to 10 using the Flux.range method. We then use reactive operators such as filter and map to transform the data stream. Finally, we subscribe to the resulting data stream and print the results using System.out::println. The publishOn operator is used to specify a parallel scheduler for concurrent execution of the processing steps.

Reactive Streams Specification

Reactive Streams Specification **is a set of rules or set of specifications that you need to follow when designing a reactive stream**. These specifications introduces four interfaces that should be used and overridden when creating a reactive stream.

- **Publisher**
  This is a **single method interface that will be used to register the subscriber to the publisher.** The subscribe method of this interface **accepts the subscriber object and registers** it.

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

Reactive Streams Specification

- **Subscriber**
  This is an interface that has four methods
  onSubscribe method will be called by the publisher when subscribing to the Subscribe object.
  onNext method will be called when the next data will be published to the subscriber
  onError method will be called when exceptions arise while publishing data to the subscriber
  onComplete method will be called after the successful completion of data publishing to the subscriber

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

Reactive Streams Specification

- **Subscription**
  This is an interface with two methods. The subscription object will be created when the user subscribes to the publisher in the publisher object as discussed earlier. The subscription object will be passed to the subscriber object via ***onSubscribe*** method
  request method will be called when the subscriber needs to request data from the publisher
  cancel method will be called when the subscriber needs to cancel and close the subscription

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

Reactive Streams Specification

- **Processor**
  This is an interface that is **extended by both publisher and subscriber interface**s. This interface is not very common but will be **used to process the logic of the subscribing and publishing workflow**

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

When should you use Reactive Programming

- When you build a **web app that will be used frequently, by many users at the same tim**e. If we use the traditional way, all the threads of the webserver will be busy and your app will become too unresponsive.
- When you are building a data streaming application. With this type of application, the data will flow in at all times. For these kinds of applications, the reactive approach is better.
- When you are building a big data and microservices applications. In Big data applications, lots of data will flow through the app, and with microservices the communications between services usually happen through streams.