

spring framework

[Document subtitle]



Ameena k a aysha C R aiswarya k haritha t h

245255 245254 245205 245182

1. INTRODUCTION

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. It was **developed by Rod Johnson in 2003**. Spring framework makes the easy development of JavaEE application.

The Spring Framework is divided into modules. Applications can choose which modules they need. At the heart are the modules of the core container, including a configuration model and a dependency injection mechanism. Beyond that, the Spring Framework provides foundational support for different application architectures, including messaging, transactional data and persistence, and web. It also includes the Servlet-based Spring MVC web framework and, in parallel, the Spring Web Flux reactive web framework.

#### **Features of the Spring Framework**

The features of the Spring framework such as IoC, AOP, and transaction management, make it unique among the list of frameworks. Some of the most important features of the Spring framework are as follows:

* **IoC container:**

Refers to the core container that uses the DI or IoC pattern to implicitly provide an object reference in a class during runtime. This pattern acts as an alternative to the service locator pattern. The IoC container contains assembler code that handles the configuration management of application objects.  
The Spring framework provides two packages, namely org.springframework.beans and org.springframework.context which helps in providing the functionality of the IoC container.

* **Data access framework:**

Allows the developers to use persistence APIs, such as JDBC and Hibernate, for storing persistence data in database. It helps in solving various problems of the developer, such as how to interact with a database connection, how to make sure that the connection is closed, how to deal with exceptions, and how to implement transaction management It also enables the developers to easily write code to access the persistence data throughout the application.

* **Spring MVC framework:**

Allows you to build Web applications based on MVC architecture. All the requests made by a user first go through the controller and are then dispatched to different views, that is, to different JSP pages or Servlets. The form handling and form validating features of the Spring MVC framework can be easily integrated with all popular view technologies such as ISP, Jasper Report, FreeMarker, and Velocity.

* **Transaction management:**

Helps in handling transaction management of an application without affecting its code. This framework provides Java Transaction API (JTA) for global transactions managed by an application server and local transactions managed by using the JDBC Hibernate, Java Data Objects (JDO), or other data access APIs. It enables the developer to model a wide range of transactions on the basis of Spring’s declarative and programmatic transaction management.

* **Spring Web Service:**

Generates Web service endpoints and definitions based on Java classes, but it is difficult to manage them in an application. To solve this problem, Spring Web Service provides layered-based approaches that are separately managed by Extensible Markup Language (XML) parsing (the technique of reading and manipulating XML). Spring provides effective mapping for transmitting incoming XML message request to an object and the developer to easily distribute XML message (object) between two machines.

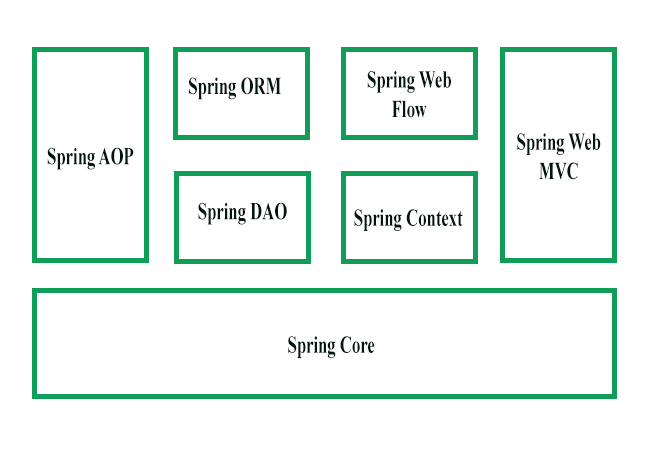
* **JDBC abstraction layer:**

Helps the users in handling errors in an easy and efficient manner. The JDBC programming code can be reduced when this abstraction layer is implemented in a Web application. This layer handles exceptions such as DriverNotFound. All SQLExceptions are translated into the DataAccessException class. Spring’s data access exception is not JDBC specific and hence Data Access Objects (DAO) are not bound to JDBC only.

* **Spring TestContext framework:**

Provides facilities of unit and integration testing for the Spring applications. Moreover, the Spring TestContext framework provides specific integration testing functionalities such as context management and caching DI of test fixtures, and transactional test management with default rollback semantics.

#### **Spring Framework Architecture**



The Spring framework consists of seven modules which are shown in the above Figure. These modules are Spring Core, Spring AOP, Spring Web MVC, Spring DAO, Spring ORM, Spring context, and Spring Web flow. These modules provide different platforms to develop different enterprise applications; for example, you can use Spring Web MVC module for developing MVC-based applications.

#### **Spring Framework Modules**

* **Spring Core Module:**

The Spring Core module, which is the core component of the Spring framework, provides the IoC container There are two types of implementations of the Spring container, namely, bean factory and application context. Bean factory is defined using the org.springframework.beans.factory.BeanFactory interface and acts as a container for beans. The Bean factory container allows you to decouple the configuration and specification of dependencies from program logic. In the Spring framework, the Bean factory acts as a central IoC container that is responsible for instantiating application objects. It also configures and assembles the dependencies between these objects. There are numerous implementations of the Bean Factory interface. The XmlBeanFactory class is the most common implementation of the Bean Factory interface. This allows you to express the object to compose your application and remove interdependencies between application objects.

* **Spring AOP Module:**

Similar to Object-Oriented Programming (OOP), which breaks down the applications into hierarchy of objects, AOP breaks down the programs into aspects or concerns. Spring AOP module allows you to implement concerns or aspects in a Spring application in Spring AOP, the aspects are the regular Spring beans or regular classes annotated with @Aspect annotation. These aspects help in transaction management and logging and failure monitoring of an application. For example, transaction management is required in bank operations such as transferring an amount from one account to another Spring AOP module provides a transaction management abstraction layer that can be applied to transaction APIs.

* **Spring ORM Module:**

The Spring ORM module is used for accessing data from databases in an application. It provides APIs for manipulating databases with JDO, Hibernate, and iBatis. Spring ORM supports DAO, which provides a convenient way to build the following DAOs-based ORM solutions:

* + Simple declarative transaction management
  + Transparent exception handling
  + Thread-safe, lightweight template classes
  + DAO support classes
  + Resource management
* **Spring Web MVC Module:**

The Web MVC module of Spring implements the MVC architecture for creating Web applications. It separates the code of model and view components of a Web application. In Spring MVC, when a request is generated from the browser, it first goes to the Dispatcher Servlet class (Front Controller), which dispatches the request to a controller (SimpleFormController class or AbstractWizardformController class) using a set of handler mappings. The controller extracts and processes the information embedded in a request and sends the result to the Dispatcher Servlet class in the form of the model object. Finally, the Dispatcher Servlet class uses View Resolver classes to send the results to a view, which displays these results to the users.

* **Spring Web Flow Module:**

The Spring Web Flow module is an extension of the Spring Web MVC module. Spring Web MVC framework provides form controllers, such as class SimpleFormController and AbstractWizardFormController class, to implement predefined workflow. The Spring Web Flow helps in defining XML file or Java Class that manages the workflow between different pages of a Web application.   
The following are the advantages of Spring Web Flow:

* + The flow between different UIs of the application is clearly provided by defining Web flow in XML file.
  + Web flow definitions help you to virtually split an application in different modules and reuse these modules in multiple situations.
* **Spring Web DAO Module:**

The DAO package in the Spring framework provides DAO support by using data access technologies such as JDBC, Hibernate, or JDO. This module introduces a JDBC abstraction layer by eliminating the need for providing tedious JDBC coding. It also provides programmatic as well as declarative transaction management classes. Spring DAO package supports heterogeneous Java Database Connectivity and O/R mapping, which helps Spring work with several data access technologies. For easy and quick access to database resources, the Spring framework provides abstract DAO base classes. Multiple implementations are available for each data access technology supported by the Spring framework. For example, in JDBC, the JdbcDaoSupport class and its methods are used to access the DataSource instance and a preconfigured JdbcTemplate instance. You need to simply extend the JdbcDaoSupport class and provide a mapping to the actual DataSource instance in an application context configuration to access a DAO-based application.

* **Spring Application Context Module:**

The Spring Application context module is based on the Core module. Application context org.springframework.context.ApplicationContext is an interface of Bean Factory. This module derives its feature from the org.springframework.beans package and also supports functionalities such as internationalization (I18N), validation, event propagation, and resource loading. The Application context implements Message Source interface and provides the messaging functionality to an application.

**IOC (Inversion of control) Container:**

Spring IoC Container is a core part of the Spring framework which is used to manage the application bean. It injects dependencies when a bean is created and **manages the bean life cycle** during execution.

The fundamental tasks of Spring IoC are:

* Instantiating
* Configuring, and
* Assembling Bean

The IOC container gets configuration related information from the Spring configuration file. That can be either **XML** or **Java** files.

The container uses [**dependency injection (DI)**](https://www.studytonight.com/spring-framework/spring-constructorbased-dependency-injection) to manage the components that make up an application.

Spring provides two types of IOC containers:

* Bean Factory
* Application Context

The Spring framework provides several implementations of the Application Context interface: ClassPathXmlApplicationContext and FileSystemXmlApplicationContext for standalone applications, and WebApplicationContext for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations.

Here's one way to manually instantiate a container:

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

**A BeanFactory:**

BeanFactory org.springframework.beans.factory.BeanFactory is the interface and XmlBeanFactory is an implementation class of it. It is a simple container which provides the basic support for dependency injection.

Syntax to use BeanFactory:

Resource resource = new ClassPathResource(“spring configuration file”);

BeanFactory beanFactory = new XmlBeanFactory(resource);

**ApplicationContext:**

ApplicationContext org.springframework.context.ApplicationContext is the interface and ClassPathXmlApplicationContext is an implementation class of it. ApplicationContext container includes all functionality of the BeanFactory container with some extra functionality like internationalization, event listeners etc.

Syntax to use ApplicationContext:

ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring configuration file");

**DEPENDENCY INJECTION IN SPRING FRAMEWORK**

Dependency Injection (DI) is a design pattern that allows the decoupling of components in an application by removing direct dependencies between them. The Spring Framework provides a powerful and flexible implementation of DI that makes it easy to manage the relationships between different components of an application.

A design pattern that is used to provide objects with their dependencies. In the context of Spring, it refers to the mechanism by which the Spring container provides objects with the dependencies they need to function correctly.

In Spring, DI is achieved through the use of a container that manages the creation and configuration of objects (beans) in an application. The container is responsible for instantiating beans and injecting any required dependencies, which allows components to be decoupled and easily tested in isolation.

Dependency injection can be done using constructor injection, setter injection, or field injection. Constructor injection is the preferred method in Spring, as it ensures that all required dependencies are provided at object creation time.

There are several ways to implement DI in Spring, including:

Constructor injection: The dependencies are injected via constructor parameters.

Setter injection: The dependencies are injected via setter methods.

Field injection: The dependencies are injected directly into fields of the class.

Method injection: The dependencies are injected via a method.

Spring also provides a feature called autowiring, which allows the container to automatically wire up dependencies based on their types or names. Autowiring can simplify configuration and make it easier to maintain large codebases. It can save developers time and reduce the amount of configuration code they need to write.

Here's an example of autowiring in Spring:

@Component

public class MyService {

@Autowired

private MyRepository repository;

}

In this example, MyService has a single dependency, MyRepository, which is annotated with @Autowired. This tells Spring to search for a bean of type MyRepository and inject it into the repository field of MyService.

Note that autowiring is not always appropriate or necessary, and it should be used judiciously. In some cases, it may be better to explicitly configure dependencies using constructor injection, setter injection, or other methods of dependency injection.

Benefits of Dependency Injection:

* DI provides a number of benefits, including making code more modular, testable, and maintainable.
* By separating concerns and reducing tight coupling, DI can also make it easier to change or add functionality to an application.

Inversion of Control (IoC): IoC is the principle that underlies DI. It means that control of object creation and management is shifted from the application to a container or framework, which then manages the dependencies between objects.

Spring IoC Container: The Spring IoC container is a lightweight container that provides DI functionality. It manages the lifecycle of beans, including instantiation, configuration, and dependency injection.

Dependency Injection Implementation using Constructor Injection:

public class MyService {

private final MyRepository repository;

public MyService(MyRepository repository) {

this.repository = repository;

}

}

In this example, MyService is constructed with a single dependency, MyRepository. The dependency is injected through the constructor.

Constructor Injection can be used with collections and maps, allowing multiple dependencies to be injected into a single component.

Constructor Injection with Collections and Maps:

In constructor injection, dependencies are passed into a class through its constructor. This can be useful when a component has several dependencies that need to be injected at once.

To use constructor injection with a collection, we can define a constructor that takes a collection of objects as a parameter. For example:

public class MyComponent

{

private List<MyDependency> dependencies;

public MyComponent(List<MyDependency> dependencies)

{

this.dependencies = dependencies;

}

}

We can then create the component and inject its dependencies using a DI container like this:

List<MyDependency> dependencies = new ArrayList<> ();

dependencies.add(new MyDependency1());

dependencies.add(new MyDependency2());

dependencies.add(new MyDependency3());

MyComponent component = new MyComponent(dependencies);

To use constructor injection with a map, we can define a constructor that takes a map of objects as a parameter. For example:

public class MyComponent

{

private Map<String, MyDependency> dependencies;

public MyComponent(Map<String, MyDependency> dependencies)

{

this.dependencies = dependencies;

}

}

We can then create the component and inject its dependencies using a DI container like this:

Map<String, MyDependency> dependencies = new HashMap<> ();

dependencies.put("dependency1", new MyDependency1());

dependencies.put("dependency2", new MyDependency2());

dependencies.put("dependency3", new MyDependency3());

MyComponent component = new MyComponent(dependencies);

Dependency Injection Implementation using Setter Injection:

public class MyService {

private MyRepository repository;

@Autowired

public void setRepository(MyRepository repository) {

this.repository = repository;

}

}

In this example, MyService has a single dependency, MyRepository, which is injected through a setter method annotated with @Autowired.

In setter injection, dependencies are passed into a class through its setter methods. This can be useful when a component has several optional dependencies, or when dependencies need to be changed at runtime.

To use setter injection with a collection, define a setter method that takes a collection of objects as a parameter. For example:

public class MyComponent {

private List<MyDependency> dependencies;

public void setDependencies(List<MyDependency> dependencies) {

this.dependencies = dependencies;

}

}

Then create the component and inject its dependencies using a DI container like this:

List<MyDependency> dependencies = new ArrayList<> ();

dependencies.add(new MyDependency1());

dependencies.add(new MyDependency2());

dependencies.add(new MyDependency3());

MyComponent component = new MyComponent();

component.setDependencies(dependencies);

To use setter injection with a map, define a setter method that takes a map of objects as a parameter. For example:

public class MyComponent {

private Map<String, MyDependency> dependencies;

public void setDependencies(Map<String, MyDependency> dependencies) {

this.dependencies = dependencies;

}

}

create the component and inject its dependencies using a DI container like this:

Map<String, MyDependency> dependencies = new HashMap<> ();

dependencies.put("dependency1", new MyDependency1());

dependencies.put("dependency2", new MyDependency2());

dependencies.put("dependency3", new MyDependency3());

MyComponent component = new MyComponent();

component.setDependencies(dependencies);

Overall, using collections and maps with CI and SI can help simplify dependency injection in an application, and make it easier to manage complex dependencies between components.

Dependency Injection Implementation using Field Injection:

public class MyService {

@Autowired

private MyRepository repository;

}

In this example, MyService has a single dependency, MyRepository, which is injected directly into a field annotated with @Autowired.

Dependency Injection Implementation using Method Injection:

public class MyService {

private MyRepository repository;

@Autowired

public void injectRepository(MyRepository repository) {

this.repository = repository;

}

}

In this example, MyService has a single dependency, MyRepository, which is injected through a method annotated with @Autowired.

The @Autowired annotation is used to tell the Spring Framework to inject the dependency. This annotation can be used with any of the types of dependency injection in Spring.

Constructor Injection vs Setter Injection

In the context of the Spring Framework, "CI" and "SI" typically refer to "Constructor Injection" and "Setter Injection", respectively. Here's a brief comparison of the two:

Constructor Injection:

Involves passing dependencies as arguments to a class constructor result in immutable objects, since the dependencies are provided at object creation time often preferred for mandatory dependencies or when a class has a large number of dependencies can result in more readable and maintainable code.

Setter Injection:

Involves setting dependencies using setter methods on a class result in mutable objects, since dependencies can be changed after object creation time often preferred for optional dependencies or when a class has a small number of dependencies can be easier to implement when working with third-party libraries or frameworks that require default constructors.

Overall, both constructor injection and setter injection are valid approaches to dependency injection in the Spring Framework.

The choice between the two depends on factors like the nature of the dependencies, the design of the class being injected, and personal preference. In many cases, a combination of both approaches may be used for different dependencies within a single class.

AOP Concepts in SPRING Framework

The Spring Framework is built on several key concepts from Aspect-Oriented Programming (AOP). AOP is a programming paradigm that complements Object-Oriented Programming (OOP) by allowing the separation of concerns that crosscut the traditional OOP modularization boundaries. Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by cross-cutting concerns. A cross-cutting concern is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

Here are some key AOP concepts used in Spring:

1. Aspect: An aspect is a module that encapsulates behaviour that can be applied to a set of classes. In Spring, an aspect is defined using a combination of a pointcut and advice.
2. Join point: A join point is a point in the execution of a program where an aspect can be applied. Examples include method calls, exception handling, and field access.
3. Pointcut: A pointcut is a set of join points where an aspect can be applied. In Spring, a pointcut is defined using expressions that match the names of methods or classes.
4. Advice: Advice is the behaviour that is applied at a join point. There are several types of advice in Spring:

a. Before advice: executed before a join point

b. After returning advice: executed after a join point completes normally

c. After throwing advice: executed after a join point throws an exception

d. After advice: executed after a join point, regardless of whether it completed normally or threw an exception

e. Around advice wraps around a join point, allowing the advice to control the execution of the join point.

1. AspectJ: AspectJ is a language that extends Java with additional constructs for AOP. Spring supports the use of AspectJ annotations and syntax for defining aspects.
2. Introduction: An introduction allows an aspect to add new methods or fields to an existing class at runtime. This is useful for adding functionality to an existing class without modifying its source code.
3. Weaving: Weaving is the process of applying aspects to a program. In Spring, weaving can be done at compile-time, load-time, or runtime.
4. Target object: The target object is the object that the aspect is applied to.
5. Proxy: A proxy is an object that is used as a substitute for the target object. The proxy intercepts method invocations and applies the advice before or after the target object's method is executed.

AspectJ annotations: AspectJ provides a set of annotations that can be used to define aspects and apply them to methods or classes. Spring supports the use of these annotations for defining aspects.

AspectJ pointcut expressions: AspectJ provides a rich set of expressions for defining pointcuts, such as method signature patterns, method call join points, and control flow join points. Spring supports the use of AspectJ pointcut expressions in its AOP framework.

By using AOP concepts, Spring provides a flexible and powerful framework for applying cross-cutting concerns to Java applications, allowing developers to create cleaner, more maintainable code.

It allows developers to modularize cross-cutting concerns, such as security, logging, and transactions, and apply them consistently across an application. This results in cleaner, more maintainable code.

Here's an example code snippet that demonstrates how to use some of these AOP concepts in Spring:

@Aspect

@Component

public class LoggingAspect {

@Before ("execution (\* com.example.myapp.service.\*. \*(..))")

public void logBefore(JoinPoint joinPoint) {

String methodName = joinPoint.getSignature().getName();

Object [] args = joinPoint.getArgs();

System.out.println("Before " + methodName + " method: " + Arrays.toString(args));

}

@AfterReturning (pointcut = "execution (\* com.example.myapp.service.\*. \*(..))", returning = "result")

public void logAfterReturning(JoinPoint joinPoint, Object result) {

String methodName = joinPoint.getSignature().getName();

System.out.println("After returning from " + methodName + " method: " + result);

}

@AfterThrowing (pointcut = "execution (\* com.example.myapp.service.\*. \*(..))", throwing = "exception")

public void logAfterThrowing(JoinPoint joinPoint, Throwable exception) {

String methodName = joinPoint.getSignature().getName();

System.out.println("After throwing from " + methodName + " method: " + exception.getMessage());

}

@Around ("execution (\* com.example.myapp.service.\*. \*(..))")

public Object logAround(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

String methodName = proceedingJoinPoint.getSignature().getName();

System.out.println("Before " + methodName + " method");

Object result = proceedingJoinPoint.proceed();

System.out.println("After " + methodName + " method");

return result;

}

}

In this code, we have defined a LoggingAspect class annotated with @Aspect and @Component annotations, indicating that it is an aspect and a Spring component. The aspect defines four advice methods:

logBefore(): This advice is executed before any method in the com.example.myapp.service package is invoked. It logs the method name and arguments.

logAfterReturning(): This advice is executed after a method in the com.example.myapp.service package returns normally. It logs the method name and return value.

logAfterThrowing(): This advice is executed after a method in the com.example.myapp.service package throws an exception. It logs the method name and exception message.

logAround(): This advice surrounds a method in the com.example.myapp.service package. It logs before and after the method invocation, and it can modify the method's behavior by returning a different value or throwing a different exception.

By using these advice methods with appropriate pointcuts, we can log the behavior of methods in the com.example.myapp.service package, without modifying the source code of those methods.

**SPRING FRAMEWORK WITH JDBC TEMPLATE**

The Spring JDBC Template is a part of the Spring Framework's JDBC (Java Database Connectivity) module.It is a powerful mechanism to connect to the database and execute SQL queries. It provides a simplified way to access relational databases from Java applications.

It internally uses JDBC api, but eliminates a lot of problems of JDBC API.It is an abstraction layer that provides a consistent programming interface to work with databases, irrespective of the underlying database vendor.Helps the users in handling errors in an easy and efficient manner. The JDBC programming code can be reduced when this abstraction layer is implemented in a Web application. This layer handles exceptions such as DriverNotFound. All SQLExceptions are translated into the DataAccessException class.Spring’s data access exception is not JDBC specific and hence Data Access Objects (DAO) are not bound to JDBC only.

**Advantages of Spring Jdbc Template**

The JDBC Template provides several advantages over the traditional JDBC approach, such as:

1. Reduced Boilerplate Code: The JDBC Template reduces the amount of boilerplate code required to execute database operations by handling common tasks such as connection management, statement creation, and result set handling.
2. Improved Exception Handling: The JDBC Template provides a uniform exception handling mechanism that translates low-level database exceptions into more meaningful and user-friendly Spring exceptions.
3. Improved Performance: The JDBC Template provides support for query caching and batch updates, which can significantly improve the performance of database operations.
4. Spring Jdbc Template eliminates all the above-mentioned problems of JDBC API. It provides you with methods to write the queries directly, so it saves a lot of work and time.

**Problems of JDBC API**

The problems of JDBC API are as follows:

* We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
* We need to perform exception handling code on the database logic.
* We need to handle transactions.
* Repetition of all these codes from one to another database logic is a time-consuming task.

To use the JDBC Template, you first configure a data source that provides access to your database. Then, you create an instance of the JdbcTemplate class and use it to execute SQL statements against the database.

**Here's an example of using the JDBC Template to query a database:**

@Autowired

private JdbcTemplate jdbcTemplate;

public List<Person> getAllPersons() {

String sql = "SELECT \* FROM person";

return jdbcTemplate.query(sql, (rs, rowNum) -> {

Person person = new Person ();

person.setId(rs.getInt("id"));

person.setName(rs.getString("name"));

person.setAge(rs.getInt("age"));

return person;

});

}

In this example, we first inject an instance of the JdbcTemplate class using Spring's dependency injection mechanism. Then, we use the JdbcTemplate to execute a SELECT query and map the result set to a list of Person objects using a lambda expression.

Overall, the Spring JDBC Template provides a simple, consistent, and powerful way to access relational databases from Java applications, and it is widely used in a variety of applications and industries.

**Spring framework provides following approaches for JDBC database access:**

* JdbcTemplate
* NamedParameterJdbcTemplate
* SimpleJdbcTemplate
* SimpleJdbcInsert and SimpleJdbcCall

**JdbcTemplate**

* It is the central class in the Spring JDBC support classes.
* Creation and release of resources such as creating and closing of connection object,so will not lead to any problem if forget to close the connection.
* It handles the exception and provides informative exception messages with the help of exception classes.
* We can perform all the database operations with the help of Jdbc Template class such as insertion, updation, deletion and retrieval of the data from the database.

Table

Description automatically generated

**Spring NamedParameterJdbcTemplate:**

* It provides another way to insert data by named parameter.
* we use names instead of? (Question mark). So, it is better to remember the data for the column.
* It is a class provided by the Spring Framework that extends the functionality of JdbcTemplate by allowing named parameters in SQL queries instead of just using placeholders (?). This makes the SQL queries more readable and easier to maintain, especially when dealing with complex queries with many parameters.
* It also supports other operations such as updates, inserts, deletes, and stored procedure calls, all with named parameters.
* To use this, you need to create an instance of it by passing in a DataSource object.
* Once you have an instance, you can use it to execute SQL queries that contain named parameters.

**SimpleJdbcTemplate**

* Class provided by the Spring Framework in earlier versions (before Spring 3.1) that acted as a simpler version of the JdbcTemplate.
* It provided a more concise and fluent API for executing database operations using JDBC.
* It also supports other operations such as updates, inserts, deletes, and stored procedure calls.

**SimpleJdbcInsert and SimpleJdbcCal**

* SimpleJdbcInsert and SimpleJdbcCall are classes provided by the Spring Framework that simplify the process of inserting data into a database and calling stored procedures respectively.
* SimpleJdbcInsert:
  + SimpleJdbcInsert provides a convenient way to insert data into a database without writing SQL queries explicitly. Instead of writing INSERT statements, you can use SimpleJdbcInsert to insert data into a table using a Map or a Bean object.
* SimpleJdbcCall:
  + SimpleJdbcCall provides a convenient way to call stored procedures without writing SQL queries explicitly. You can use SimpleJdbcCall to call a stored procedure with input parameters, output parameters, and a return value.

**SPRINGS WITH ORM FRAMEWORK**

* Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA (Java Persistence API), JDO (Java Data Objects), Oracle Toplink and iBATIS.
* The Spring ORM module is used for accessing data from databases in an application.
* It provides APIs for manipulating databases with JDO, Hibernate, and iBatis.
* Spring ORM supports DAO, which provides a convenient way to build the following DAOs-based ORM solutions:
  + - Simple declarative transaction management
    - Transparent exception handling
    - Thread-safe, lightweight template classes
    - DAO support classes
    - Resource management
* Spring Framework provides excellent support for Object-Relational Mapping (ORM) frameworks such as Hibernate, JPA (Java Persistence API), and MyBatis. ORM frameworks help developers to work with relational databases using object-oriented programming paradigms.

**Here's an example of using Hibernate with Spring to perform database operations:**

@Repository

public class PersonRepositoryImpl implements PersonRepository {

@Autowired

private SessionFactory sessionFactory;

@Override

public void saves (Person person) {

Session session = sessionFactory.getCurrentSession();

session.saveOrUpdate(person);

}

@Override

public List<Person> findAll() {

Session session = sessionFactory.getCurrentSession();

CriteriaQuery<Person> query = session.getCriteriaBuilder().createQuery(Person.class);

query.from(Person.class);

return session.createQuery(query).getResultList();

}

@Override

public Person findById(Long id) {

Session session = sessionFactory.getCurrentSession();

return session.find(Person.class, id);

}

@Override

public void deletes (Person person) {

Session session = sessionFactory.getCurrentSession();

session.delete(person);

}}

* In this example, we use Hibernate as the ORM framework and Spring for dependency injection. We use the sessionFactory bean to obtain a Session object that can be used to perform database operations. We also use Hibernate's Criteria API to perform a SELECT query.
* Overall, Spring provides a rich set of features and tools that make it easy to use ORM frameworks in your applications.

**Advantage of ORM Frameworks with Spring**

1. Simplified Database Operations: ORM frameworks provide an abstraction layer over the database that reduces the amount of code required to perform CRUD (Create, Read, Update, Delete) operations on the database. Spring makes it even easier by providing a consistent programming model to work with different ORM frameworks.
2. Improved Performance: ORM frameworks can help improve performance by reducing the number of database queries and by using caching. Spring supports various caching mechanisms such as Ehcache, Redis, and Hazelcast.
3. Easy Integration with Spring Ecosystem: Spring provides a range of modules such as Spring Security, Spring Data, and Spring MVC, which can be easily integrated with ORM frameworks to build robust and scalable applications.

4.Less coding is required: you don't need to write extra codes before and after the actual database logic such as getting the connection, starting transaction, commiting transaction, closing connection etc.

5.Easy to test: Spring's IoC approach makes it easy to test the application.

6.Better exception handling: Spring framework provides its own API for exception handling with ORM framework.

7.Integrated transaction management: By the help of Spring framework, we can wrap our mapping code with an explicit template wrapper class or AOP style method interceptor.

Spring Mvc

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**. Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

Spring Web Model-View-Controller



* **Model** - A model contains the data of the application. A data can be a single object or a collection of objects.
* **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
* **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
* **Front Controller** - In Spring Web MVC, the Dispatcher Servlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

## Understanding the flow of Spring Web MVC



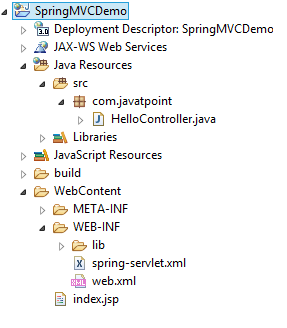
* As displayed in the figure, all the incoming request is intercepted by the Dispatcher Servlet that works as the front controller.
* The Dispatcher Servlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
* The controller returns an object of ModelAndView.
* The Dispatcher Servlet checks the entry of view resolver in the XML file and invokes the specified view component.

## Spring Web MVC Framework Example

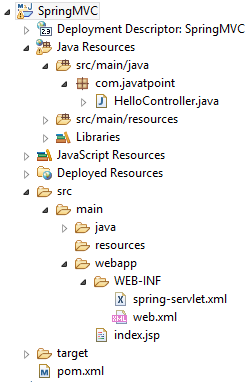
Let's see the simple example of a Spring Web MVC framework. The steps are as follows:

* Load the spring jar files or add dependencies in the case of Maven
* Create the controller class
* Provide the entry of controller in the web.xml file
* Define the bean in the separate XML file
* Display the message in the JSP page
* Start the server and deploy the project

## Directory Structure of Spring MVC



## Directory Structure of Spring MVC using Maven



## Required Jar files or Maven Dependency

To run this example, you need to load:

* Spring Core jar files
* Spring Web jar files
* JSP + JSTL jar files (If you are using any another view technology then load the corresponding jar files).

**Download Link:** [Download all the jar files for spring including JSP and JSTL](https://static.javatpoint.com/src/sp/springjars.zip).

If you are using Maven, you don't need to add jar files. Now, you need to add maven dependency to the pom.xml file.

### **1. Provide project information and configuration in the pom.xml file.**

**pom.xml**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.javatpoint**</groupId>**
5. **<artifactId>**SpringMVC**</artifactId>**
6. **<packaging>**war**</packaging>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **<name>**SpringMVC Maven Webapp**</name>**
9. **<url>**http://maven.apache.org**</url>**
10. **<dependencies>**
11. **<dependency>**
12. **<groupId>**junit**</groupId>**
13. **<artifactId>**junit**</artifactId>**
14. **<version>**3.8.1**</version>**
15. **<scope>**test**</scope>**
16. **</dependency>**
18. <! -- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
19. **<dependency>**
20. **<groupId>**org.springframework**</groupId>**
21. **<artifactId>**spring-webmvc**</artifactId>**
22. **<version>**5.1.1. RELEASE**</version>**
23. **</dependency>**
25. <! -- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
26. **<dependency>**
27. **<groupId>**javax.servlet**</groupId>**
28. **<artifactId>**servlet-api**</artifactId>**
29. **<version>**3.0-alpha-1**</version>**
30. **</dependency>**
32. **</dependencies>**
33. **<build>**
34. **<finalName>**SpringMVC**</finalName>**
35. **</build>**
36. **</project>**

### **2. Create the controller class**

To create the controller class, we are using two annotations @Controller and @RequestMapping.

The @Controller annotation marks this class as Controller.

The @Requestmapping annotation is used to map the class with the specified URL name.

**HelloController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. @Controller
5. **public** **class** HelloController {
6. @RequestMapping ("/")
7. **public** String display ()
8. {
9. **return** "index";
10. }
11. }

### **3. Provide the entry of controller in the web.xml file**

In this xml file, we are specifying the servlet class DispatcherServlet that acts as the front controller in Spring Web MVC. All the incoming request for the html file will be forwarded to the DispatcherServlet.

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### **4. Define the bean in the xml file**

This is the important configuration file where we need to specify the View components.

The context:component-scan element defines the base-package where DispatcherServlet will search the controller class.

This xml file should be located inside the WEB-INF directory.

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
14. <! -- Provide support for component scanning -->
15. **<context:component-scan** base-package="com.javatpoint" **/>**
17. <! --Provide support for conversion, formatting and validation -->
18. **<mvc:annotation-driven/>**
20. **</beans>**

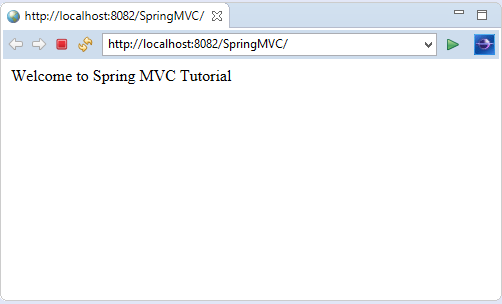
### **5. Display the message in the JSP page**

This is the simple JSP page, displaying the message returned by the Controller.

**index.jsp**

1. **<html>**
2. **<body>**
3. **<p>**Welcome to Spring MVC Tutorial**</p>**
4. **</body>**
5. **</html>**

**Output:**



Features

* [Core technologies](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html): dependency injection, events, resources, i18n, validation, data binding, type conversion, SpEL, AOP.

* [Testing](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html): mock objects, TestContext framework, Spring MVC Test, WebTestClient.
* [Data Access](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/data-access.html): transactions, DAO support, JDBC, ORM, Marshalling XML.

* [Spring MVC](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html) and [Spring WebFlux](https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html) web frameworks.

* [Integration](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/integration.html): remoting, JMS, JCA, JMX, email, tasks, scheduling, cache.

* [Languages](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/languages.html): Kotlin, Groovy, dynamic languages.

Spring Pros

There are the following advantages of the Spring framework:

1. **Light Weight:** Spring is a lightweight framework because of its POJO implementation. It does not force the programmer to inherit any class and implement any interface. With the help of Spring, we can enable powerful, scalable applications using POJOs (Plain Old Java Object).
2. **Flexible:** It provides flexible libraries trusted by developers all over the world. The developer can choose either XML or Java-based annotations for configuration options. The IoC and DI features provide the foundation for a wide-ranging set of features and functionality. It makes the job simpler.
3. **Loose Coupling:** Spring applications are loosely coupled because of dependency injection. It handles injecting dependent components without a component knowing where they came from.
4. **Powerful Abstraction:** It provides a powerful abstraction to JEE specifications such as JMS, JDBC, JPA, and JTA.
5. **Declarative Support:** It provides declarative support for caching, validation, transaction, and formatting.
6. **Portable:** We can use server-side in web/EJB app, client-side in swing app business logic is completely portable.
7. **Cross-cutting behavior:** Resource management is a cross-cutting concern, easy to copy and paste everywhere.
8. **Configuration:** It provides a consistent way of configuring everything, separate configuration from application logic, varying configuration.
9. **Lifecycle:** Responsible for managing all your application components, particularly those in middle-tier container sees components through well-defined lifecycle: init(), destroy ().
10. **Dependency Injection:** The use of dependency injection makes the easy development of JavaEE.
11. **Easier Testing:** The use of dependency injection makes the testing easy. The spring framework does not require a server while the EJB and Struts application requires a server.
12. **Fast:** The team of Spring engineers deeply cares about the performance. Its fast startup, fast shutdown, and optimized execution maintain performance make it fast. Even, we can start a new Spring project in seconds by using Spring Initializr.
13. **Secure:** It monitors third-party dependencies closely. The regular update is issued that make our data and application secure. We can make our application secure by using the Spring Security framework. It provides industry-standard security schemes and delivers a trustworthy solution that is secure by default.
14. **Supportive:** The Spring community provides support and resources to get you to the next level QuickStart guides, tutorials, videos, and meetup helps a lot.
15. **Productive:** It is more productive because the spring application can integrate with other Spring-based applications. For example, we can combine the Spring Boot application with Spring Cloud.

Spring Cons

1. **Complexity:** Working with Spring is more complex. It requires a lot of expertise. If you have not used Spring before, first you will have to learn. The learning curve is also difficult, so if you have not a lot of development experience, it is difficult to learn.
2. **Parallel Mechanism:** It provides multiple options to developers. These options create confusion to developers that which feature to use and which to not and wrong decisions may lead to significant delays.
3. **No Specific Guidelines:** It does not care about XSS or cross-site scripting. With this in mind, we need to figure out ways on how to stop hackers from infiltrating your application yourself.
4. **High Learning Curve:** If you have not development experience in the field, it would be quite difficult to learn It is difficult due to new programming methods.
5. **Lots of XML:** Developing a Spring application requires lots of XML.