**Introduction to the Spring Framework**

The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

Spring is the most popular application development framework for enterprise Java. Spring framework is an open source Java platform. Spring is lightweight when it comes to size and transparency.

Spring modules:

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, Messaging, and Test etc,

**Spring container:**

The **Spring container** is the core of **Spring** Framework. The **container**, use for creating the objects and configuring them. Also, **Spring** IoC **Containers** use for managing the complete lifecycle from creation to its destruction. It uses Dependency Injection (DI) to manage components and these objects are called **Spring** Beans.

**Inversion of Control**

IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes.

IoC Container

The IoC container is a framework used to manage automatic dependency injection throughout the application, so that we as programmers do not need to put more time and effort into it.

**Dependency Injection**

Dependency Injection (DI) is a design pattern which implements the IoC principle to invert the creation of dependent objects.

The task of instantiating objects is done by the container according to the configurations specified by the developer.

**Types of Spring Dependency Injection:**

There are two types of Spring Dependency Injection. They are:

**Setter Dependency Injection (SDI)**: This is the simpler of the two DI methods. In this, the DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the **bean-configuration file** For this, the property to be set with the SDI is declared under the **<property>** tag in the bean-config file.

**Constructor Dependency Injection (CDI)**: In this, the DI will be injected with the help of [contructors](https://www.geeksforgeeks.org/constructors-in-java/). Now to set the DI as CDI in bean, it is done through the **bean-configuration file** For this, the property to be set with the CDI is declared under the **<constructor-arg>** tag in the bean-config file.

The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods.

Sping Beans:

**Spring beans are Java objects that are managed by the Spring container.** The Spring container is responsible for instantiating, configuring, and assembling the Spring beans.

[BeanFactory](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface provides a simple, yet flexible configuration mechanism to manage objects of any nature via the Spring IoC container.

This is the root interface for accessing the Spring container.

The **ApplicationContext**is the central interface within a Spring application that is used for providing configuration information to the application.

It implements the BeanFactory interface. Hence, the ApplicationContext includes all functionality of the BeanFactory and much more! Its main function is to support the creation of big business applications.

## **Spring Configuration Metadata**

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. Following are the three important methods to provide configuration metadata to the Spring Container −

* **XML based configuration file.**

**Construtctor-based dependency injection:**

<bean id="emp" class="com.lti.model.Employee">

<constructor-arg name="employeeId" value="1001"></constructor-arg>

<constructor-arg name="employeeName" value="John"></constructor-arg>

<constructor-arg ref="dept"></constructor-arg>

<constructor-arg ref="prj"></constructor-arg>

</bean>

<bean id="dept" class="com.lti.model.Department">

<constructor-arg name="departmentId" value="1"></constructor-arg>

<constructor-arg name="departmentName" value="IT"></constructor-arg>

</bean>

<bean id="prj" class="com.lti.model.Project">

<constructor-arg name="projectId" value="1212"></constructor-arg>

<constructor-arg name="projectName" value="Bus Reservation"></constructor-arg>

<constructor-arg name="duration" value="2"></constructor-arg>

</bean>

**Setter Injection:**

<bean id="cust" class="com.lti.model.Customer">

<property name="customerId" value="1234"></property>

<property name="customerName" value="George"></property>

<property name="address" ref="add"></property>

</bean>

<bean id="add" class="com.lti.model.Address">

<property name="addressId" value="1010"></property>

<property name="houseNo" value="204"></property>

<property name="city" value="Pune"></property>

</bean>

* **Annotation-based configuration**

**@Component:**

@Component is the most generic Spring annotation. A Java class decorated with @Component is found during classpath scanning and registered in the context as a Spring bean. @Service, @Repository, and @Controller are specializations of @Component, which are used for more specific cases.

**@Autowired:** **Autowiring** feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection. **Autowiring** can't be used to inject primitive and string values. It works with reference only.

<context:component-scan base-package=*"com.lti.model"*/>

* Java-based configuration

@Configuration

@ComponentScan(basePackages="com.lti")

**public** **class** AppConfig {

}

**@ComponentScan** ensures that the classes decorated with @Component are found and registered as Spring beans.

Spring support for DataSource:

### DataSource

Spring obtains a connection to the database through a DataSource. A DataSource is part of the JDBC specification and is a generalized connection factory. It allows a container or a framework to hide connection pooling and transaction management issues from the application code. As a developer, you need not know details about how to connect to the database; that is the responsibility of the administrator that sets up the datasource.

@Configuration

@ComponentScan("com.baeldung.jdbc")

public class SpringJdbcConfig {

@Bean

public DataSource mysqlDataSource() {

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc");

dataSource.setUsername("guest\_user");

dataSource.setPassword("guest\_password");

return dataSource;

}

}

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/springjdbc"/>

<property name="username" value="guest\_user"/>

<property name="password" value="guest\_password"/>

</bean>

Connection pooling:

Connection pooling is a well-known data access pattern, whose main purpose is to reduce the overhead involved in performing database connections and read/write database operations.

public class DBCPDataSource {

private static BasicDataSource ds = new BasicDataSource();

static {

ds.setUrl("jdbc:h2:mem:test");

ds.setUsername("user");

ds.setPassword("password");

ds.setMinIdle(5);

ds.setMaxIdle(10);

ds.setMaxOpenPreparedStatements(100);

}

public static Connection getConnection() throws SQLException {

return ds.getConnection();

}

private DBCPDataSource(){ }

}

once the pool is created, connections are fetched from the pool, so there's no need to create new ones.

Furthermore, when a connection is released, it's actually returned back to the pool, so other clients can reuse it.

## DriverManager

* hampers the application performance as the connections are created/closed in java classes.
* does not support connection pooling.

For DriverManager you need to know all the details (host, port, username, password, driver class) to connect to DB and to get connections. Externalizing those in a properties file doesn't change anything about the fact that you need to know them.

## DataSource

* improves application performance as connections are not created/closed within a class, they are managed by the application server and can be fetched while at runtime.
* it provides a facility creating a pool of connections
* helpful for enterprise applications

Using a DataSource you only need to know the JNDI name. The AppServer cares about the details and is not configured by the client application's vendor, but by an admin where the application is hosted.

Spring JDBC support:

Spring provides a simplification in handling database access with the Spring JDBC Template. The Spring JDBC Template has the following advantages compared with standard JDBC.

* The Spring JDBC template allows to clean-up the resources automatically, e.g. release the database connections.
* The Spring JDBC template converts the standard JDBC SQLExceptions into RuntimeExceptions. This allows the programmer to react more flexible to the errors. The Spring JDBC template converts also the vendor specific error messages into better understandable error messages.

## JdbcTemplate class

It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.

It handles the exception and provides the informative exception messages by the help of excepion classes defined in the org.springframework.dao package.

We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.

**public int update(String query):** is used to insert, update and delete records.

**public int update(String query,Object... args):** is used to insert, update and delete records using PreparedStatement using given arguments.

**public List query(String sql, RowMapper rse) :** is used to fetch records using RowMapper.

**Spring REST**

REST has now become a standard way to develop web services and when it comes to Java, there are many frameworks and libraries available e.g. JAX-RS, Restlet, Jersey, RESTEasy, Apache CFX etc  
  
Spring RestController annotation is used to create RESTful web services using Spring MVC. Spring RestController takes care of mapping request data to the defined request handler method. Once response body is generated from the handler method, it converts it to JSON or XML response.

### The most important reason to use Spring for developing RESTful web service is that you can use your Spring MVC experience to develop RESTful web services. In Spring MVC, a controller can handle requests for all HTTP methods, which is a backbone of RESTful web services. @RequestMapping

The Controller class contains several handler methods to handle different HTTP. It provides the mapping between the request path and the handler method. It also supports some advanced option which can be used to specify separate handler methods for different types of request on the same URI like you can specify a method to handle GET request and another to handle POST request on same URI.

**@RequestParam**

### It is used to bind HTTP parameters into method arguments of handler methods. @PathVariable

### This is another annotation that is used to retrieve data from the URL. Unlike @RequestParam annotation which is used to extract query parameters, this annotation enables the controller to handle a request for parameterized URLs like URLs that have variable input as part of their path @RequestBody

This annotation can convert inbound HTTP data into Java objects passed into the controller's handler method. The @RequestBody annotations tell the Spring to find a suitable message converter to convert a resource representation coming from a client into an object.

### @RestController

This is a convenience annotation for developing a RESTful web service with the Spring MVC framework. The @RestController is a combination of @Controller and @ResponseBody,  
  
When you annotate a controller class with @RestController it does two purposes, first, it says that the controller class is handling a request for REST APIs and second you don't need to annotate each method with the @ResposneBody annotation to signal that the response will be converted into a Resource using various HttpMessageConverers.