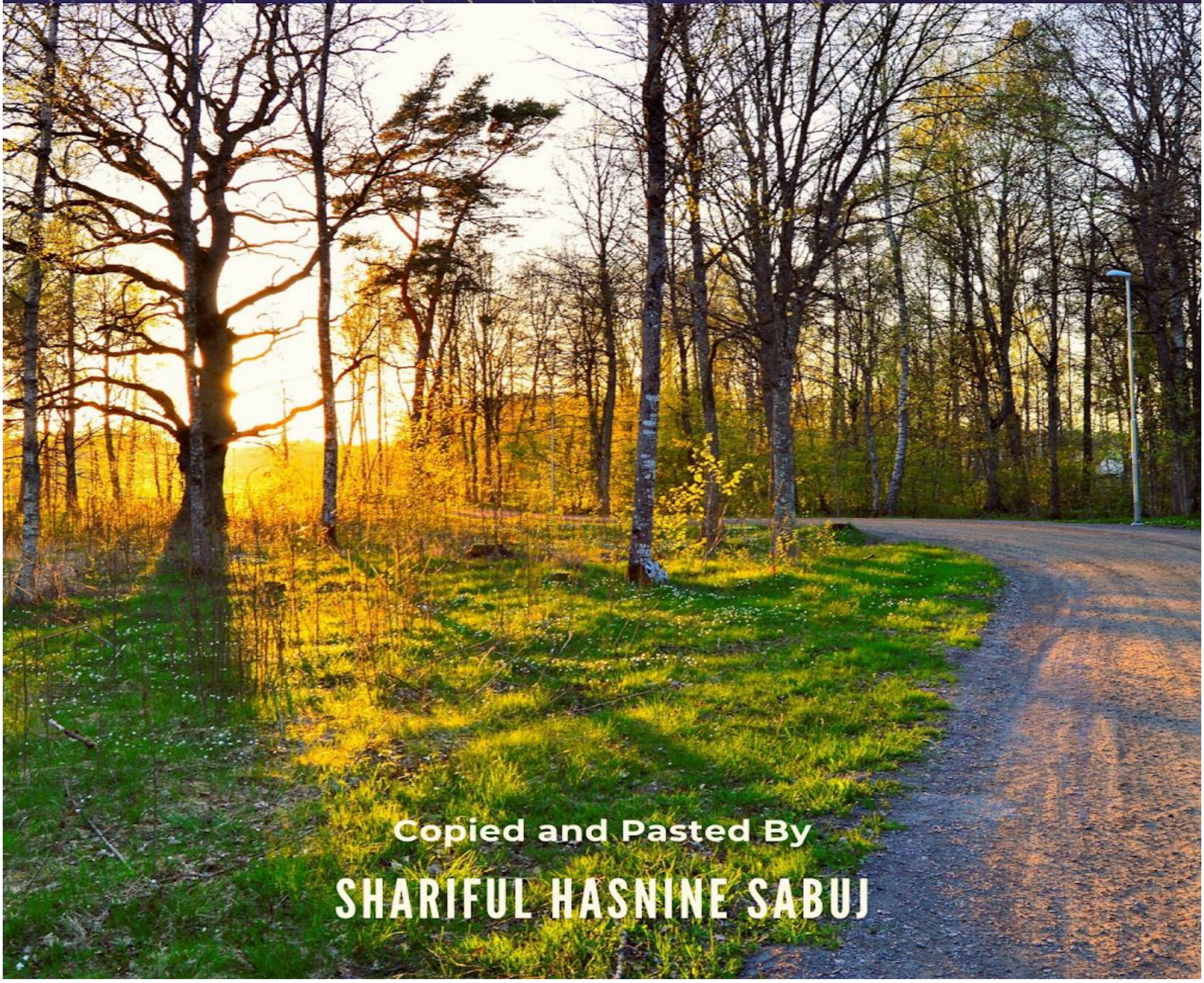
SPRING THEORY

THE SPRING DEVELOPER'S HANDBOOK



Copied and Pasted By

SHARIFUL HASNINE SABUJ

What is REST

REST is an acronym for REpresentational State Transfer. It is an architectural style for distributed hypermedia systems. Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as RESTful.

Guiding Principles of REST

1. **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
2. **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
3. **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
4. **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
5. **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.
6. **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

Resource

The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service, a collection of other resources, a non-virtual object (e.g. a person), and so on. REST uses a resource identifier to identify the particular resource involved in an interaction between components.

The state of the resource at any particular timestamp is known as resource representation. A representation consists of data, metadata describing the data and hypermedia links which can help the clients in transition to the next desired state.

The data format of a representation is known as a media type. The media type identifies a specification that defines how a representation is to be processed. A truly RESTful API looks like *hypertext*

REST and HTTP are not the same !!

A lot of people prefer to compare HTTP with REST. REST and HTTP are not the same.

REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs). The resources are acted upon by using a set of simple, well-defined operations. The clients and servers exchange representations of resources by using a standardized interface and protocol – typically HTTP.

Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. And most importantly, every interaction with a resource is stateless.

All these principles help RESTful applications to be simple, lightweight, and fast.

Microservice

A microservice is an engineering approach focused on decomposing applications into single-function modules with well-defined interfaces which are independently deployed and operated by small teams who own the entire lifecycle of the service. Microservices accelerate delivery by minimizing communication and coordination between people while reducing the scope and risk of change.

- **Decomposing**

Each service handles a specific business domain (logging, auth, orders, customers) and provides the implementation for the user interface, business logic, and connection to the database.

```
class UserApp {
```

```
User getUser() {
   // 1. auth user
   // 2. get user data
   // 3. log user actions
  }
}
class UserApp {
 void authUser(User user) { ... }
 User getUserData() { ... }
 void logUserActions() { ... }
}
```

- **Single-function**

Each and every service has a specific function or responsibility. And yes, a service can do many tasks, but all of them are nevertheless relevant to this single function.

- **Well-defined interfaces**

Services must provide an interface that defines how we can communicate with it. This basically defines a list of methods, and their inputs and outputs.

- **Independent**

Independent means services don't know about each other's implementation. They can get tested, deployed, and maintained independently.
It might be the case where services are implemented using different language stacks and communicate with different databases.
But that doesn't mean they don't work together. They do, in order to complete their required operation.

```
class UserApp {
    void authUser(User user) {
   // log user login action (success or failure)
   // using logUserActions
     }
   User getUserData() { ... }
   void logUserActions() { ... }
}
```

- **Small Teams**

We split the work up and team across the services. Each team focuses on a specific service, they don't need to know about the internal workings of other teams.
Those teams can work efficiently, communicate easily, and each service can be deployed rapidly as soon as it's ready.

- **Entire Lifecycle**

The team is responsible for the entire lifecycle of the service; from coding, testing, staging, deploying, debugging, maintaining.
In a traditional application, we may have a team for coding, and another one for deployment. In microservices, that's not the case.

- **Minimizing Communication**

Minimizing communication doesn't mean that the team members should ignore each other. It means the only essential cross-team communication should be through the interface that each service provides.
They all need to agree on the external interface so that communication between services is clearly defined.

- **The scope and risk of change**

Services should be changed without breaking other services. And so long as we don't change the external interface there will be no problem for other services.
As a result of changes, the versions of services are updating individually, and there is no relationship between them.

Most Important Features of the Spring Framework

- **Lightweight**

The Spring Framework is very lightweight with respect to its size and functionality. This is due to its POJO implementation, which doesn't force it to inherit any class or implement any interfaces.

- **Aspect-Oriented Programming (AOP)**

This is an important part of the Spring Framework. Aspect-Oriented Programming is used for separating cross-cutting concerns (for example, logging, security, etc.) from the business logic of the application. In the coming articles, you will be learning about this in greater detail.

- **Transaction Management**

This is used to unify several transaction management APIs and is used to coordinate transactions for Java objects. Also, it is not tied to the J2EE environment and is used with containerless environments.

- **Container**

The Spring framework designs and manages the lifecycle and configurations of application objects.

- **Dependency Injection**

This feature of the Spring Framework allows you to develop loosely coupled applications. Therefore, the unit testing of these loosely coupled applications becomes easier. This also allows the developer to swap out some of the modules according to its need.

- **Integration With Other Frameworks**

A great thing about this framework is that it doesn't try to solve the problems that have already been solved. It just tries to integrate them with its framework, which provides a solution to greater problems. For example, this could include IBATIS, Hibernate, Toplink, etc.

Spring boot

Spring Boot brings an opinionated approach to the Spring ecosystem. There are different areas where this popular library tries to help us out:

- Dependency management. Through starters and various package manager integrations
- Autoconfiguration. Trying to minimize the amount of config a Spring app requires to get ready to go and favoring convention over configuration
- Production-ready features. Such as *Actuator*, better logging, monitoring, metrics or various PAAS integration
- Enhanced development experience. With multiple testing utilities or a better feedback loop using *spring-boot-devtools*

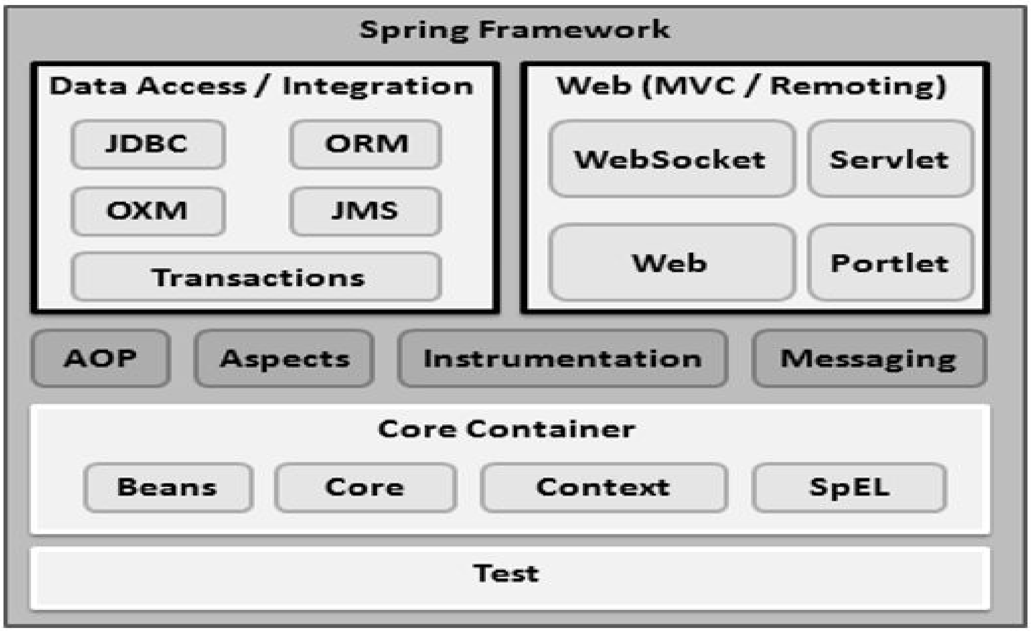
Spring Boot 2.x will no longer support Java 7 and below, being Java 8 the minimum requirement.A few highlights regarding minimum required versions:

- Tomcat minimum supported version is 8.5
- Hibernate minimum supported version is 5.2
- Gradle minimum supported version is 3.4

Spring Framework - Architecture

Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework.

The Spring Framework provides about 20 modules which can be used based on an application requirement.



Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

- **The Core module** provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- **The Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
- **The Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- **The SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

- **The JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- **The ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- **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.

Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows –

- **The 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-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- **The Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of the Web-Servlet module.

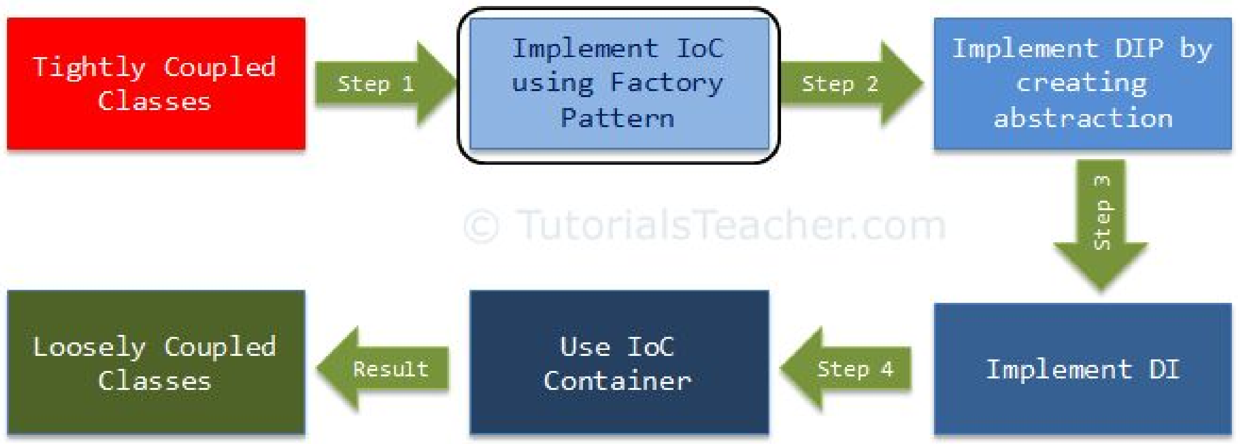
Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows –

- **The AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- **The Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
- **The Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- **The Messaging module** provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- **The Test module** supports the testing of Spring components with JUnit or TestNG frameworks.

Inversion of control

Inversion of Control (IoC) is a design principle (although, some people refer to it as a pattern). As the name suggests, it is used to invert different kinds of controls in object-oriented design to achieve loose coupling. Here, controls refer to any additional responsibilities a class has, other than its main responsibility. This includes control over the flow of an application, and control over the flow of an object creation or dependent object creation and binding.



IoC is all about inverting the control. To explain this in layman's terms, suppose you drive a car to your workplace. This means you control the car. The IoC principle suggests inverting the control, meaning that instead of driving the car yourself, you hire a cab, where another person will drive the car. Thus, this is called inversion of the control - from you to the cab driver. You don't have to drive a car yourself and you can let the driver do the driving so that you can focus on your main work.

The IoC principle helps in designing loosely coupled classes which make them testable, maintainable and extensible.

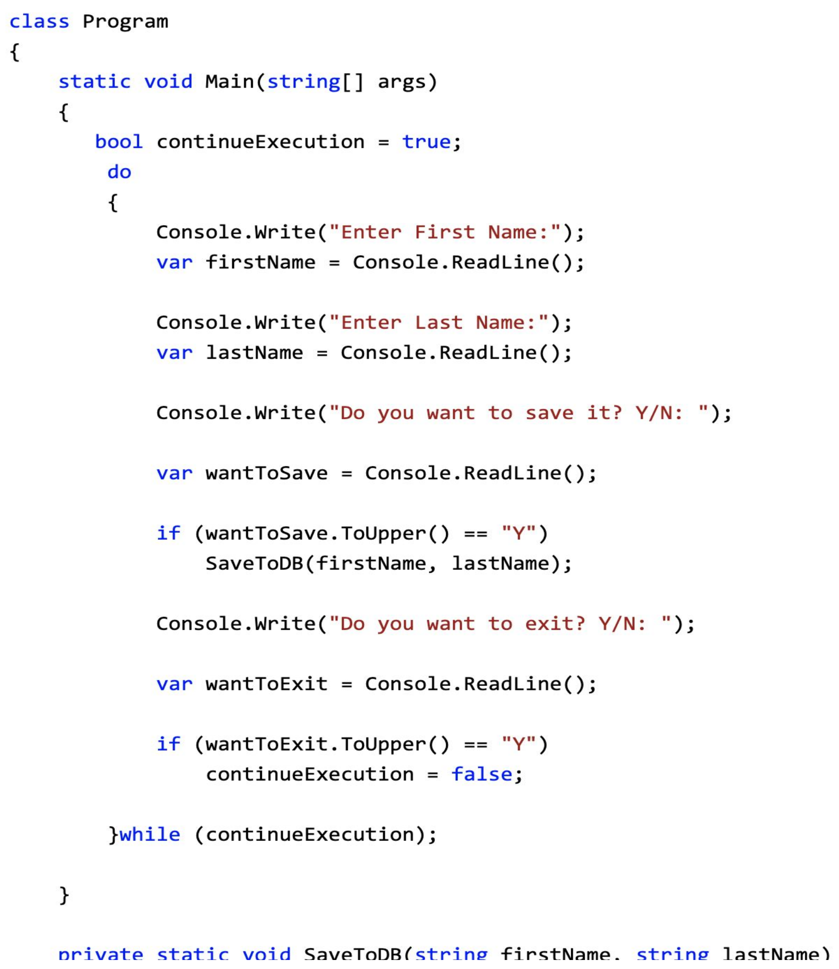
Let's understand how IoC inverts the different kinds of control.

Control Over the Flow of a Program

In a typical console application in C#, execution starts from the Main() function. The Main() function controls the flow of a program or, in other words, the sequence of user interaction. Consider the following simple console program.

In the next example, the Main() function of the program class controls the flow of a program. It takes the user's input for the first name and last name. It saves the data, and continues or exits the console, depending upon the user's input. So here, the flow is controlled through the Main() function.

IoC can be applied to the above program by creating a GUI-based application such as the following windows-based application, wherein the framework will handle the flow of a program by using events.



```
class Program
{
    static void Main(string[] args)
    {
       bool continueExecution = true;
        do
        {
            Console.Write("Enter First Name:");
            var firstName = Console.ReadLine();

            Console.Write("Enter Last Name:");
            var lastName = Console.ReadLine();

            Console.Write("Do you want to save it? Y/N: ");

            var wantToSave = Console.ReadLine();

            if (wantToSave.ToUpper() == "Y")
                SaveToDB(firstName, lastName);

            Console.Write("Do you want to exit? Y/N: ");

            var wantToExit = Console.ReadLine();

            if (wantToExit.ToUpper() == "Y")
                continueExecution = false;

        }while (continueExecution);

    }

    private static void SaveToDB(string firstName, string lastName)
```

Control Over the Dependent Object Creation

IoC can also be applied when we create objects of a dependent class. First of all, let's understand what we mean by dependency here.

Consider the following example.

In the next example, class A calls b.SomeMethod() to complete its task1. Class A cannot complete its task without class B and so you can say that "Class A is dependent on class B" or "class B is a dependency of class A".

In the object-oriented design approach, classes need to interact with each other in order to complete one or more functionalities of an application, such as in the next example - classes A and B. Class A creates and manages the lifetime of an object of class B. Essentially, it controls the creation and life time of objects of the dependency class.

```
public class A
{
    B b;

    public A()
    {
        b = new B();
    }

    public void Task1() {
        // do something here..
        b.SomeMethod();
        // do something here..
    }

}

public class B {

    public void SomeMethod() {
        //doing something..
    }
}
```

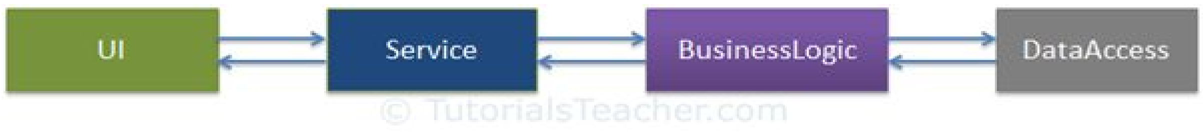
The IoC principle suggests to invert the control. This means to delegate the control to another class. In other words, invert the dependency creation control from class A to another class, as shown below.

```
public class A
{
    B b;

    public A()
    {
        b = Factory.GetObjectOfB ();
    }

    public void Task1() {
        // do something here..
        b.SomeMethod();
        // do something here..
    }
}

public class Factory
{
    public static B GetObjectOfB()
    {
        return new B();
    }
}
```

As you can see above, class A uses Factory class to get an object of class B. Thus, we have inverted the dependent object creation from class A to Factory. Class A no longer creates an object of class B, instead it uses the factory class to get the object of class B.

Let's understand this using a more practical example.

In an object-oriented design, classes should be designed in a loosely coupled way. Loosely coupled means changes in one class should not force other classes to change, so the whole application can become maintainable and extensible. Let's understand this by using typical n-tier architecture as depicted by the following figure:



In the typical n-tier architecture, the User Interface (UI) uses a Service layer to retrieve or save data. The Service layer uses the **BusinessLogic** class to apply business rules on the data. The **BusinessLogic** class depends on the **DataAccess** class which retrieves or saves the data to the underlying database. This is simple n-tier architecture design. Let's focus on the **BusinessLogic** and **DataAccess** classes to understand IoC.The following is an example of **BusinessLogic** and **DataAccess** classes for a customer.

```
public class CustomerBusinessLogic
{
    DataAccess _dataAccess;

    public CustomerBusinessLogic()
    {
        _dataAccess = new DataAccess();
    }

    public string GetCustomerName(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public class DataAccess
{
    public DataAccess()
    {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name"; // get it from DB in real app
    }
}
```

As you can see in the above example, the **CustomerBusinessLogic** class depends on the **DataAccess** class. It creates an object of the **DataAccess** class to get the customer data.

Now, let's understand what's wrong with the above classes.

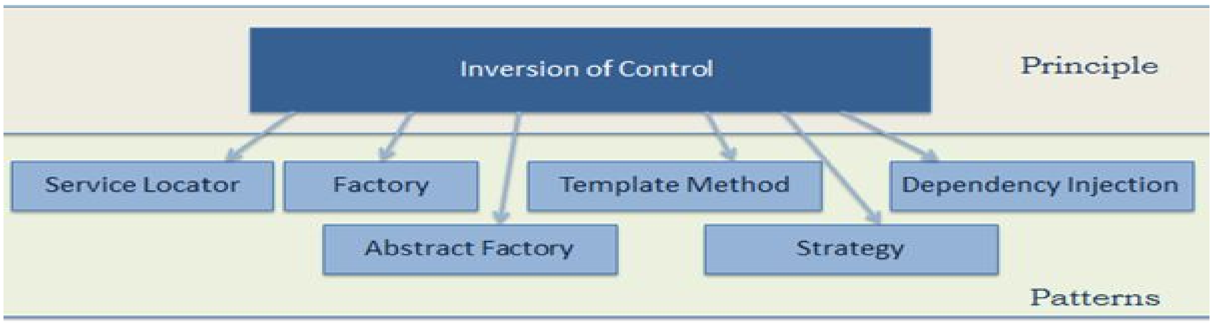
In the above example, CustomerBusinessLogic and DataAccess are tightly coupled classes because the CustomerBusinessLogic class includes the reference of the concrete

DataAccess class. It also creates an object of DataAccess class and manages the lifetime of the object.

Problems in the above example classes:

1. CustomerBusinessLogic and DataAccess classes are tightly coupled classes. So, changes in the DataAccess class will lead to changes in the CustomerBusinessLogic class. For example, if we add, remove or rename any method in the DataAccess class then we need to change the CustomerBusinessLogic class accordingly.
2. Suppose the customer data comes from different databases or web services and, in the future, we may need to create different classes, so this will lead to changes in the CustomerBusinessLogic class.
3. The CustomerBusinessLogic class creates an object of the DataAccess class using the new keyword. There may be multiple classes which use the DataAccess class and create its objects. So, if you change the name of the class, then you need to find all the places in your source code where you created objects of DataAccess and make the changes throughout the code. This is repetitive code for creating objects of the same class and maintaining their dependencies.
4. Because the CustomerBusinessLogic class creates an object of the concrete DataAccess class, it cannot be tested independently (TDD). The DataAccess class cannot be replaced with a mock class.

To solve all of the above problems and get a loosely coupled design, we can use the IoC and DIP principles together. Remember, IoC is a principle, not a pattern. It just gives high-level design guidelines but does not give implementation details. You are free to implement the IoC principle the way you want.The following pattern (but not limited) implements the IoC principle.



Let's use the *Factory* pattern to implement IoC in the above example, as the first step towards attaining loosely coupled classes.

First, create a simple Factory class which returns an object of the DataAccess class as shown below.

```
public class DataAccessFactory
{
    public static DataAccess GetDataAccessObj()
    {
        return new DataAccess();
    }
}
```

Now, use this DataAccessFactory class in the CustomerBusinessLogic class to get an object of DataAccess class.

```
public class CustomerBusinessLogic
{

    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        DataAccess _dataAccess =  DataAccessFactory.GetDataAccessObj();

        return _dataAccess.GetCustomerName(id);
    }
}
```

As you can see, the CustomerBusinessLogic class uses the DataAccessFactory.GetCustomerDataAccessObj() method to get an object of the DataAccess class instead of creating it using the *new* keyword. Thus, we have inverted the control of creating an object of a dependent class from the CustomerBusinessLogic class to the DataAccessFactory class.

This is a simple implementation of IoC and the first step towards achieving fully loose coupled design. As mentioned in the previous chapter, we will not achieve complete loosely coupled classes by only using IoC. Along with IoC, we also need to use DIP, Strategy pattern, and DI (Dependency Injection).

Let's move to the second step to understand DIP and how it helps in achieving loose coupled design in the next chapter.

Dependency Injection

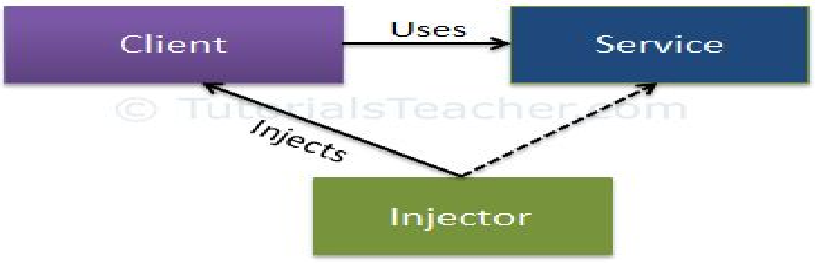
Here, we are going to implement Dependency Injection and strategy pattern together to move the dependency object creation completely out of the class. This is our third step in making the classes completely loose coupled.

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

1. **Client Class**: The client class (dependent class) is a class which depends on the service class
2. **Service Class**: The service class (dependency) is a class that provides service to the client class.
3. **Injector Class**: The injector class injects the service class object into the client class.

The following figure illustrates the relationship between these classes:



As you can see, the injector class creates an object of the service class, and injects that object to a client object. In this way, the DI pattern separates the responsibility of creating an object of the service class out of the client class.

Types of Dependency Injection

As you have seen above, the injector class injects the service (dependency) to the client (dependent). The injector class injects dependencies broadly in three ways: through a constructor, through a property, or through a method.

Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.

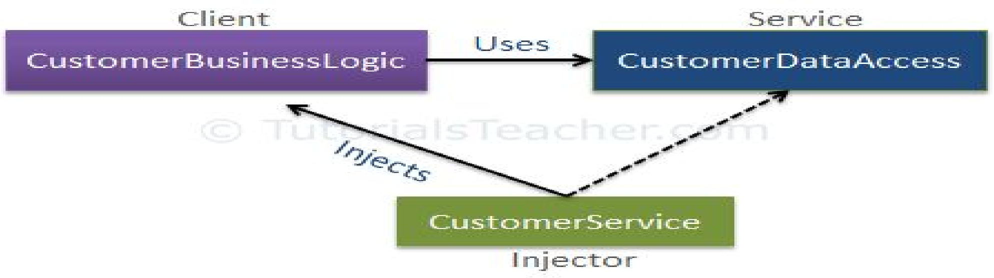
Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

Let's take an example from the previous chapter to maintain the continuity. In the previous section of DIP, we used Factory class inside the CustomerBusinessLogic class to get an object of the CustomerDataAccess object, as shown below.

```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess() {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name";
    }
}

public class DataAccessFactory
{
    public static ICustomerDataAccess GetCustomerDataAccessObj()
    {
        return new CustomerDataAccess();
    }
}

public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

    public CustomerBusinessLogic()
    {
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

    public string GetCustomerName(int id)
    {
        return _custDataAccess.GetCustomerName(id);
    }
}
```

The problem with the above example is that we used DataAccessFactory inside the CustomerBusinessLogic class. So, suppose there is another implementation of ICustomerDataAccess and we want to use that new class inside CustomerBusinessLogic. Then, we need to change the source code of the CustomerBusinessLogic class as well. The Dependency injection pattern solves this problem by injecting dependent objects via a constructor, a property, or an interface.

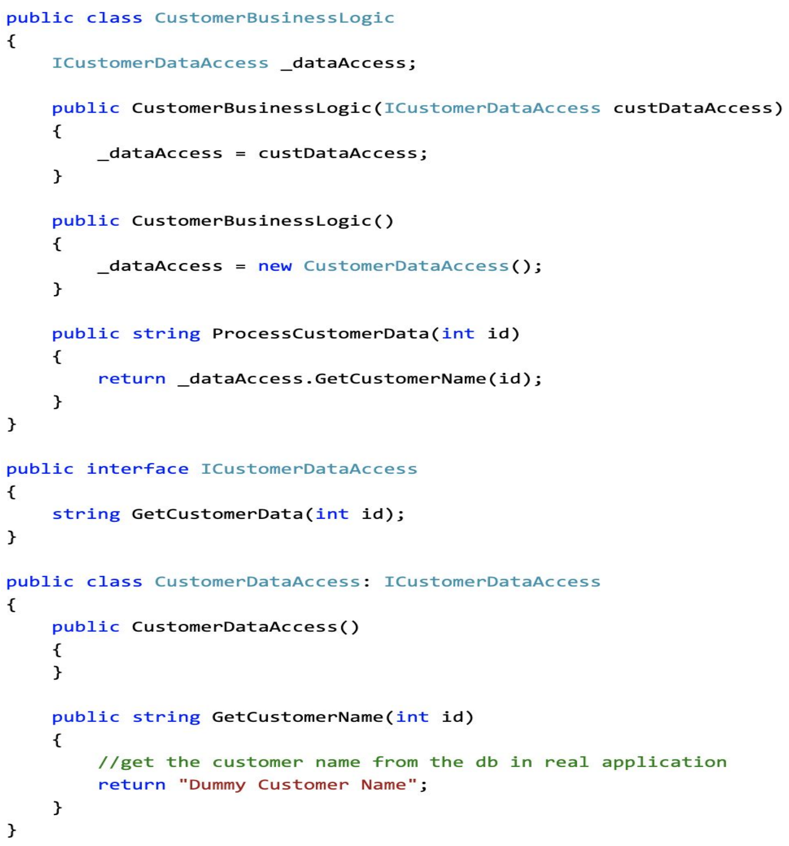
The following figure illustrates the DI pattern implementation for the above example.



As you see, the CustomerService class becomes the injector class, which sets an object of the service class (CustomerDataAccess) to the client class (CustomerBusinessLogic) either through a constructor, a property, or a method to achieve loose coupling. Let's explore each of these options.

Constructor Injection

As mentioned before, when we provide the dependency through the constructor, this is called a constructor injection.Consider the following example where we have implemented DI using the constructor.



```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _dataAccess;

    public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)
    {
        _dataAccess = custDataAccess;
    }

    public CustomerBusinessLogic()
    {
        _dataAccess = new CustomerDataAccess();
    }

    public string ProcessCustomerData(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public interface ICustomerDataAccess
{
    string GetCustomerData(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess()
    {
    }

    public string GetCustomerName(int id)
    {
        //get the customer name from the db in real application
        return "Dummy Customer Name";
    }
}
```

In the above example, CustomerBusinessLogic includes the constructor with one parameter of type ICustomerDataAccess. Now, the calling class must inject an object of ICustomerDataAccess.

```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic(new CustomerDataAccess());
    }

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }
}
```

As you can see in the above example, the CustomerService class creates and injects the CustomerDataAccess object into the CustomerBusinessLogic class. Thus, the CustomerBusinessLogic class doesn't need to create an object of CustomerDataAccess using the new keyword or using factory class. The calling class (CustomerService) creates and sets the appropriate DataAccess class to the CustomerBusinessLogic class. In this way, the CustomerBusinessLogic and CustomerDataAccess classes become "more" loosely coupled classes.

Property Injection

In the property injection, the dependency is provided through a public property. Consider the following example.

```
public class CustomerBusinessLogic
{
    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        return DataAccess.GetCustomerName(id);
    }

    public ICustomerDataAccess DataAccess { get; set; }
}

public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic();
        _customerBL.DataAccess = new CustomerDataAccess();
    }

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }
}
```

As you can see above, the CustomerBusinessLogic class includes the public property named DataAccess, where you can set an instance of a class that implements

ICustomerDataAccess. So, CustomerService class creates and sets CustomerDataAccess class using this public property.

Method Injection

In the method injection, dependencies are provided through methods. This method can be a class method or an interface method.The following example demonstrates the method injection using an interface based method.

```
interface IDataAccessDependency
{
    void SetDependency(ICustomerDataAccess customerDataAccess);
}

public class CustomerBusinessLogic : IDataAccessDependency
{
    ICustomerDataAccess _dataAccess;

    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }

    public void SetDependency(ICustomerDataAccess customerDataAccess)
    {
        _dataAccess = customerDataAccess;
    }
}

public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic();
        ((IDataAccessDependency)_customerBL).SetDependency(new CustomerDataAccess());
    }

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(id);
    }
}
```

In the above example, the CustomerBusinessLogic class implements the IDataAccessDependency interface, which includes the SetDependency() mehtod. So, the injector class CustomerService will now use this method to inject the dependent class (CustomerDataAccess) to the client class.Thus, you can use DI and strategy pattern to create loose coupled classes.

So far, we have used several principles and patterns to achieve loosely coupled classes. In professional projects, there are many dependent classes and implementing these patterns is time consuming. Here the IoC Container (aka the DI container) helps us. Learn about the IoC Container in the next chapter.

IoC Container

In the previous chapter, we learned how to implement the Dependency Injection pattern to achieve loose coupled classes. IoC Container (a.k.a. DI Container) is a framework for implementing automatic dependency injection. It manages object creation and it's life-time, and also injects dependencies to the class.

The IoC container creates an object of the specified class and also injects all the dependency objects through a constructor, a property or a method at run time and disposes it at the appropriate time. This is done so that we don't have to create and manage objects manually.

All the containers must provide easy support for the following DI life cycle.

- Register: The container must know which dependency to instantiate when it encounters a particular type. This process is called registration. Basically, it must include some way to register type-mapping.
- Resolve: When using the IoC container, we don't need to create objects manually. The container does it for us. This is called resolution. The container must include some methods to resolve the specified type; the container creates an object of the specified type, injects the required dependencies if any and returns the object.
- Dispose: The container must manage the lifetime of the dependent objects. Most IoC containers include different lifetime managers to manage an object's lifecycle and dispose of it.

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans, which we will discuss in the next chapter.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

Spring provides the following two distinct types of containers.

1. **Spring BeanFactory Container**: This is the simplest container providing the basic support for DI and is defined by the *org.springframework.beans.factory.BeanFactory* interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean,

are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.

2. **Spring ApplicationContext Container**: This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the *org.springframework.context.ApplicationContext* interface.

Spring- Bean Definition

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container. For example, in the form of XML <bean/> definitions which you have already seen in the previous chapters.

Bean definition contains the information called configuration metadata, which is needed for the container to know the following –

- How to create a bean
- Bean's life cycle details
- Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

1. **Class**: This attribute is mandatory and specifies the bean class to be used to create the bean.
2. **Name**: This attribute specifies the bean identifier uniquely. In XMLbased configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
3. **Scope**: This attribute specifies the scope of the objects created from a particular bean definition
4. **Constructor-arg**: This is used to inject the dependencies and will be discussed in subsequent chapters.
5. **Properties**: This is used to inject the dependencies.
6. **autowiring mode**: This is used to inject the dependencies and will be discussed in subsequent chapters.
7. **Lazy-initialization mode**: A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup.
8. **initialization method**: A callback to be called just after all necessary properties on the bean have been set by the container.
9. **Destruction mode**: A callback to be used when the container containing the bean is destroyed.

The Singleton scope

If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton. However, when you need one and only one instance of a bean, you can set the scope property to singleton in the bean configuration file, as shown in the following code snippet –

```
              <!-- A bean definition with singleton scope →
           <bean id = "..." class = "..." scope = "singleton">
        <!-- collaborators and configuration for this bean go here →
   </bean>
```

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

      objA.setMessage("I'm object A");
      objA.getMessage();

      HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
      objB.getMessage();
   }
}
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message –

```
Your Message : I'm object A
Your Message : I'm object A
```

The prototype scope

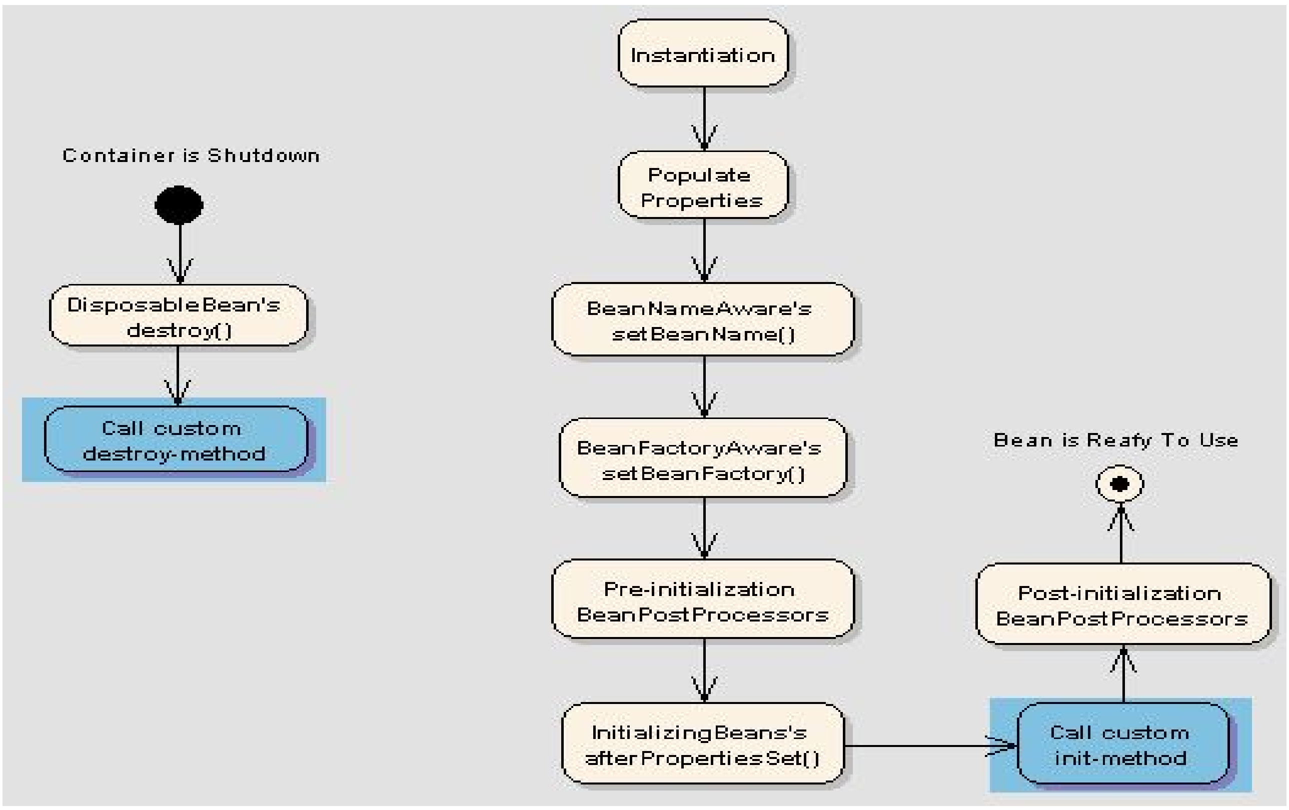
If the scope is set to prototype, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

To define a prototype scope, you can set the scope property to prototype in the bean configuration file, as shown in the following code snippet –

```
<!-- A bean definition with prototype scope -->
<bean id = "..." class = "..." scope = "prototype">
   <!-- collaborators and configuration for this bean go here -->
</bean>
```

```
Your Message : I'm object A
Your Message : null
```

Spring - Bean life cycle


Instantiation
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Pre-initialization BeanPostProcessors
InitializingBeans's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean is Reafy To Use
Container is Shutdown
DisposableBean's destroy()
Call custom destroy-method

When the container starts – a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some post-initialization steps to get it into a usable state. *Same bean life cycle is for spring boot applications as well.*

After that, when the bean is no longer required, it will be removed from the IoC container.

Spring bean factory is responsible for managing the life cycle of beans created through spring containers.

1.1. Life cycle callbacks

Spring bean factory controls the creation and destruction of beans. To execute some custom code, it provides the callback methods which can be categorized broadly in two groups:

- Post-initialization call back methods
- Pre-destruction call back methods

Life Cycle callback method

Spring framework provides following 4 ways for controlling life cycle events of a bean:

1. InitializingBean and DisposableBean callback interfaces
2. *Aware interfaces for specific behavior
3. Custom init() and destroy() methods in bean configuration file
4. @PostConstruct and @PreDestroy annotations

2.1. InitializingBean and DisposableBean

The org.springframework.beans.factory.InitializingBean interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container.

The InitializingBean interface specifies a single method:

```
 void afterPropertiesSet() throws Exception;
```

This is not a preferable way to initialize the bean because it tightly coupled your bean class with the spring container. A better approach is to use the "*init-method*" attribute in bean definition in applicationContext.xml file.

Similarly, implementing the org.springframework.beans.factory.DisposableBean interface allows a bean to get a callback when the container containing it is destroyed.

The DisposableBean interface specifies a single method:

```
void destroy() throws Exception;
```

A sample bean implementing above interfaces would look like this:

```
package com.howtodoinjava.task;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class DemoBean implements InitializingBean, DisposableBean
{
    //Other bean attributes and methods

    @Override
    public void afterPropertiesSet() throws Exception
    {
        //Bean initialization code
    }

    @Override
    public void destroy() throws Exception
    {
        //Bean destruction code
    }
}
```

Spring - Bean Definition Inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

Spring Bean definition inheritance has nothing to do with Java class inheritance but the inheritance concept is the same. You can define a parent bean definition as a template and other child beans can inherit the required configuration from the parent bean.

When you use XML-based configuration metadata, you indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute.

```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
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">
      <property name = "message1" value = "Hello World!"/>
      <property name = "message2" value = "Hello Second World!"/>
   </bean>

   <bean id ="helloIndia" class = "com.tutorialspoint.HelloIndia" parent = "helloWorld">
      <property name = "message1" value = "Hello India!"/>
      <property name = "message3" value = "Namaste India!"/>
   </bean>
</beans>
```

```java
package com.tutorialspoint;

public class HelloWorld {
   private String message1;
   private String message2;

   public void setMessage1(String message){
      this.message1 = message;
   }
   public void setMessage2(String message){
      this.message2 = message;
   }
   public void getMessage1(){
      System.out.println("World Message1 : " + message1);
   }
   public void getMessage2(){
      System.out.println("World Message2 : " + message2);
   }
}

public class HelloIndia {
   private String message1;
   private String message2;
   private String message3;

   public void setMessage1(String message){
      this.message1 = message;
   }
   public void setMessage2(String message){
      this.message2 = message;
   }
   public void setMessage3(String message){
      this.message3 = message;
   }
   public void getMessage1(){
      System.out.println("India Message1 : " + message1);
   }
   public void getMessage2(){
      System.out.println("India Message2 : " + message2);
   }
   public void getMessage3(){
      System.out.println("India Message3 : " + message3);
   }
}
```

Following is the content of the MainApp.java file –

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");
      objA.getMessage1();
      objA.getMessage2();

      HelloIndia objB = (HelloIndia) context.getBean("helloIndia");
      objB.getMessage1();
      objB.getMessage2();
      objB.getMessage3();
   }
}
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message –

```
World Message1 : Hello World!
World Message2 : Hello Second World!
India Message1 : Hello India!
India Message2 : Hello Second World!
India Message3 : Namaste India!
```

What is the difference between BeanFactory and ApplicationContext?

BeanFactory is the **basic container** whereas ApplicationContext is the **advanced container**. ApplicationContext extends the BeanFactory interface. ApplicationContext provides more facilities than BeanFactory such as integration with spring AOP, message resource handling for i18n etc.

Core Spring Framework - Annotations

@Required- This annotation is applied on bean setter methods. Consider a scenario where you need to enforce a required property. The @Required annotation indicates that the affected bean must be populated at configuration time with the required property. Otherwise an exception of type BeanInitializationException is thrown.

@Autowired- This annotation is applied on fields, setter methods, and constructors. The @Autowired annotation injects object dependency implicitly.When you use @Autowired on fields and pass the values for the fields using the property name, Spring will automatically assign the fields with the passed values.

You can even use @Autowired on private properties, as shown below. (This is a very poor practice though!)

```
public class Customer {
  @Autowired
  private Person person;
  private int type;
}
```

When you use @Autowired on setter methods, Spring tries to perform the by Type autowiring on the method. You are instructing Spring that it should initiate this property using a setter method where you can add your custom code, like initializing any other property with this property.

```
public class Customer {
  private Person person;
```

```
   @Autowired
   public void setPerson (Person person) {
    this.person=person;
   }
}
```

When you use @Autowired on a constructor, constructor injection happens at the time of object creation. It indicates the constructor to autowire when used as a bean. One thing to note here is that only one constructor of any bean class can carry the @Autowired annotation.

```
@Component
public class Customer {
   private Person person;
   @Autowired
   public Customer (Person person) {
     this.person=person;
   }
}
```

@Qualifier- This annotation is used along with @Autowired annotation. When you need more control of the dependency injection process, @Qualifier can be used. @Qualifier can be specified on individual constructor arguments or method parameters. This annotation is used to avoid confusion which occurs when you create more than one bean of the same type and want to wire only one of them with a property.

```
@Component
public class BeanA {
 @Autowired
 @Qualifier("beanB2")
 private BeanInterface dependency;
 ...
```

```java
}
```

@Configuration- This annotation is used on classes which define beans. @Configuration is an analog for XML configuration file – it is configuration using Java class. Java class annotated with @Configuration is a configuration by itself and will have methods to instantiate and configure the dependencies.

Here is an example:

```java
@Configuration
public class DataConfig{
 @Bean
 public DataSource source(){
   DataSource source = new OracleDataSource();
   source.setURL();
   source.setUser();
   return source;
 }
 @Bean
 public PlatformTransactionManager manager(){
   PlatformTransactionManager manager = new BasicDataSourceTransactionManager();
   manager.setDataSource(source());
   return manager;
 }
}
```

@ComponentScan- This annotation is used with @Configuration annotation to allow Spring to know the packages to scan for annotated components. @ComponentScan is also used to specify base packages using basePackageClasses or basePackage attributes to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

@Bean- This annotation is used at the method level. @Bean annotation works with @Configuration to create Spring beans. As mentioned earlier, @Configuration will have methods to instantiate and configure dependencies. Such methods will be annotated with @Bean. The method annotated with this annotation works as bean ID and it creates and returns the actual bean.

Here is an example:

```
@Configuration
public class AppConfig{
 @Bean
 public Person person(){
   return new Person(address());
 }
 @Bean
 public Address address(){
   return new Address();
 }
}
```

@Lazy- his annotation is used on component classes. By default all autowired dependencies are created and configured at startup. But if you want to initialize a bean lazily, you can use @Lazy annotation over the class. This means that the bean will be created and initialized only when it is first requested for. You can also use this annotation on @Configuration classes. This indicates that all @Bean methods within that @Configuration should be lazily initialized.

@Value-- This annotation is used at the field, constructor parameter, and method parameter level. The @Value annotation indicates a default value expression for the field or parameter to initialize the property with. As the @Autowired annotation tells Spring to inject an object into another when it loads your application context, you can also use @Value annotation to inject values from a property file into a bean's attribute. It supports both #{...} and ${...} placeholders.

Spring Framework Stereotype Annotations

@Component-This annotation is used on classes to indicate a Spring component. The @Component annotation marks the Java class as a bean or say component so that the component-scanning mechanism of Spring can add into the application context.

@Controller- The @Controller annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC or Spring WebFlux.

@Service- This annotation is used on a class. The @Service marks a Java class that performs some service, such as executing business logic, performing calculations and calling external APIs. This annotation is a specialized form of the @Component annotation intended to be used in the service layer.

@Repository- This annotation is used on Java classes which directly access the database. The @Repository annotation works as a marker for any class that fulfills the role of repository or Data Access Object.This annotation has an automatic translation feature. For example, when an exception occurs in the @Repository there is a handler for that exception and there is no need to add a try catch block.

Spring Boot Annotations

@EnableAutoConfiguration- This annotation is usually placed on the main application class. The @EnableAutoConfiguration annotation implicitly defines a base "search package". This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

@SpringBootApplication- This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the @SpringBootApplication must be kept in the base package. The one thing that the @SpringBootApplication does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with @SpringBootApplication in com.example then @SpringBootApplication will scan all its sub-packages, such as com.example.a, com.example.b, and com.example.a.x. The @SpringBootApplication is a convenient annotation that adds all the following: @Configuratio, @EnableAutoConfiguration, @ComponentScan

Spring Boot Rest and MVC annotations

@Controller- This annotation is used on Java classes that play the role of controller in your application. The @Controller annotation allows autodetection of component classes in the classpath and auto-registering bean definitions for them. To enable autodetection of such annotated controllers, you can add component scanning to your configuration. The Java class annotated with @Controller is capable of handling multiple request mappings.This annotation can be used with Spring MVC and Spring WebFlux.

@RequestMapping- This annotation is used both at class and method level. The @RequestMapping annotation is used to map web requests onto specific handler classes and handler methods. When @RequestMapping is used on class level it creates a base URI for which the controller will be used. When this annotation is used on methods it will give you the URI on which the handler methods will be executed. From this you can infer that the class level request mapping will remain the same whereas each handler method will have their own request mapping.Sometimes you may want to perform different operations based on the HTTP method used, even though the request URI may remain the same. In such situations, you can use the method attribute of @RequestMapping with an HTTP method value to narrow down the HTTP methods in order to invoke the methods of your class.

Here is a basic example on how a controller along with request mappings work:

```
@Controller
@RequestMapping("/welcome")
public class WelcomeController{
 @RequestMapping(method = RequestMethod.GET)
 public String welcomeAll(){
   return "welcome all";
 }
}
```

In this example only GET requests to /welcome is handled by the **welcomeAll**() method.This annotation also can be used with Spring MVC and Spring WebFlux.The @RequestMapping annotation is very versatile.

@CookieValue- This annotation is used at method parameter level. @CookieValue is used as an argument for the request mapping method. The HTTP cookie is bound to the @CookieValue parameter for a given cookie name. This annotation is used in the method annotated with @RequestMapping.

Let us consider that the following cookie value is received with a http request:

JSESSIONID=418AB76CD83EF94U85YD34W

To get the value of the cookie, use @CookieValue like this:

```
@RequestMapping("/cookieValue")
public void getCookieValue(@CookieValue "JSESSIONID" String cookie){
}
```

@CrossOrigin- This annotation is used both at class and method level to enable cross origin requests. In many cases the host that serves JavaScript will be different from the host that serves the data. In such a case Cross Origin Resource Sharing (CORS) enables cross-domain communication. To enable this communication you just need to add the @CrossOrigin annotation. By default the @CrossOrigin annotation allows all origin, all headers, the HTTP methods specified in the @RequestMapping annotation and maxAge of 30 min. You can customize the behavior by specifying the corresponding attribute values.

An example to use @CrossOrigin at both controller and handler method levels is this.

```
@CrossOrigin(maxAge = 3600)
@RestController
```

```
@RequestMapping("/account")
public class AccountController {

@CrossOrigin(origins = "http://example.com")
@RequestMapping("/message")
 public Message getMessage() {
    // ...
  }

@RequestMapping("/note")
       public Note getNote() {
     // ...
  }
}
```

In this example, both **getExample**() and **getNote**() methods will have a maxAge of 3600 seconds. Also, **getExample**() will only allow cross-origin requests from http://example.com, while **getNote**() will allow cross-origin requests from all hosts.

Composed @RequestMapping Variants

Spring framework 4.3 introduced the following method-level variants of @RequestMapping annotation to better express the semantics of the annotated methods. Using these annotations have become the standard ways of defining the endpoints. They act as a wrapper to @RequestMapping.These annotations can be used with Spring MVC and Spring WebFlux.

@GetMapping-This annotation is used for mapping HTTP GET requests onto specific handler methods. @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET)

@PostMapping- This annotation is used for mapping HTTP POST requests onto specific handler methods. @PostMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST)

@PutMapping- This annotation is used for mapping HTTP PUT requests onto specific handler methods. @PutMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PUT)

@PatchMapping- This annotation is used for mapping HTTP PATCH requests onto specific handler methods. @PatchMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PATCH)

@DeleteMapping- This annotation is used for mapping HTTP DELETE requests onto specific handler methods. @DeleteMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.DELETE)

@ExceptionHandler- This annotation is used at method levels to handle exceptions at the controller level. The @ExceptionHandler annotation is used to define the class of exception it will catch. You can use this annotation on methods that should be invoked to handle an exception. The @ExceptionHandler values can be set to an array of Exception types. If an exception is thrown that matches one of the types in the list, then the method annotated with matching @ExceptionHandler will be invoked.

@InitBinder- This annotation is a method level annotation that plays the role of identifying the methods which initialize the WebDataBinder - a DataBinder that binds the request parameter to JavaBean objects. To customise request parameter data binding , you can use @InitBinder annotated methods within our controller. The methods annotated with @InitBinder all argument types that handler methods support.The @InitBinder annotated methods will get called for each HTTP request if you don't specify the value element of this annotation. The value element can be a single or multiple form names or request parameters that the init binder method is applied to.

@Mappings and @Mapping -This annotation is used on fields. The @Mapping annotation is a meta annotation that indicates a web mapping annotation. When mapping different field names, you need to configure the source field to its target field and to do that you have to add the @Mappings annotation. This annotation accepts an array of @Mapping having the source and the target fields.

@MatrixVariable- This annotation is used to annotate request handler method arguments so that Spring can inject the relevant bits of matrix URI. Matrix variables can appear on any segment each separated by a semicolon. If a URL contains matrix variables, the request mapping pattern must represent them with a URI template. The @MatrixVariable annotation ensures that the request is matched with the correct matrix variables of the URI.

@PathVariable- This annotation is used to annotate request handler method arguments. The @RequestMapping annotation can be used to handle dynamic changes in the URI where a certain URI value acts as a parameter. You can specify this parameter using a regular expression. The @PathVariable annotation can be used to declare this parameter.

@RequestAttribute-This annotation is used to bind the request attribute to a handler method parameter. Spring retrieves the named attributes value to populate the parameter annotated with @RequestAttribute. While the @RequestParam annotation is used to bind the parameter values from the query string, the @RequestAttribute is used to access the objects which have been populated on the server side.

@RequestBody- This annotation is used to annotate request handler method arguments. The @RequestBody annotation indicates that a method parameter should be bound to the value of the HTTP request body. The HttpMessageConveter is responsible for converting from the HTTP request message to the object.

@RequestHeader- This annotation is used to annotate request handler method arguments. The @RequestHeader annotation is used to map controller parameters to request header value. When Spring maps the request, @RequestHeader checks the header with the name specified within the annotation and binds its value to the handler method parameter. This annotation helps you to get the header details within the controller class.

@RequestParam- This annotation is used to annotate request handler method arguments. Sometimes you get the parameters in the request URL, mostly in GET requests. In that case, along with the @RequestMapping annotation you can use the @RequestParam annotation to retrieve the URL parameter and map it to the method argument. The @RequestParam annotation is used to bind request parameters to a method parameter in your controller.

@RequestPart- This annotation is used to annotate request handler method arguments. The @RequestPart annotation can be used instead of @RequestParam to get the content of a specific multipart and bind to the method argument annotated with @RequestPart. This annotation takes into consideration the "Content-Type" header in the multipart(request part).

@ResponseBody- This annotation is used to annotate request handler methods. The @ResponseBody annotation is similar to the @RequestBody annotation. The @ResponseBody annotation indicates that the result type should be written straight in the response body in whatever format you specify like JSON or XML. Spring converts the returned object into a response body by using the HttpMessageConveter.

@ResponseStatus- This annotation is used on methods and exception classes. @ResponseStatus marks a method or exception class with a status code and a reason that must be returned. When the handler method is invoked the status code is set to the HTTP response which overrides the status information provided by any other means. A controller class can also be annotated with @ResponseStatus which is then inherited by all @RequestMapping methods.

@RestController- This annotation is used at the class level. The @RestController annotation marks the class as a controller where every method returns a domain object instead of a view. By annotating a class with this annotation you no longer need to add @ResponseBody to all the RequestMapping methods. It means that you no longer use view-resolvers or send html in response. You just send the domain object as an HTTP response in the format that is understood by the consumers like JSON. @RestController is a convenience annotation which combines @Controller and @ResponseBody.

@Transactional- This annotation is placed before an interface definition, a method on an interface, a class definition, or a public method on a class. The mere presence of @Transactional is not enough to activate the transactional behaviour. The @Transactional is simply a metadata that can be consumed by some runtime infrastructure. This infrastructure uses the metadata to configure the appropriate beans with transactional behaviour.The annotation further supports configuration like:The Propagation type of the transaction, The Isolation level of the transaction, A timeout for the operation wrapped by the transaction, A read only flag - a hint for the persistence provider that the transaction must be read only, The rollback rules for the transaction.

Cache-Based Annotations

@Cacheable- This annotation is used on methods. The simplest way of enabling the cache behaviour for a method is to annotate it with @Cacheable and parameterize it with the name of the cache where the results would be stored.

```
@Cacheable("addresses")
public String getAddress(Book book){...}
```

In the snippet above , the method getAddress is associated with the cache named addresses. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated.

@CachePut- This annotation is used on methods. Whenever you need to update the cache without interfering with the method execution, you can use the @CachePut annotation. That is, the method will always be executed and the result cached.

```
@CachePut("addresses")
public String getAddress(Book book){...}
```

Using @CachePut and @Cacheable on the same method is strongly discouraged as the former forces the execution in order to execute a cache update, the latter causes the method execution to be skipped by using the cache.

@CacheEvict- This annotation is used on methods. It is not that you always want to populate the cache with more and more data. Sometimes you may want to remove some cache data so that you can populate the cache with some fresh values. In such a case use the @CacheEvict annotation.

```
@CacheEvict(value="addresses", allEntries="true")
public String getAddress(Book book){...}
```

Here an additional element allEntries is used along with the cache name to be emptied. It is set to true so that it clears all values and prepares to hold new data.

@CacheConfig- This annotation is a class level annotation. The @CacheConfig annotation helps to streamline some of the cache information at one place. Placing this annotation on a class does not turn on any caching operation.

Task Execution and Scheduling Annotations

@Scheduled- This annotation is a method level annotation. The @Scheduled annotation is used on methods along with the trigger metadata. A method with @Scheduled should have void return type and should not accept any parameters. There are different ways of using the @Scheduled annotation:

```
@Scheduled(fixedDelay=5000)
public void doSomething() {
// something that should execute periodically
}
```

In this case, the duration between the end of last execution and the start of next execution is fixed. The tasks always wait until the previous one is finished.

```
@Scheduled(fixedRate=5000)
public void doSomething() {
// something that should execute periodically
}
```

In this case, the beginning of the task execution does not wait for the completion of the previous execution.

```
@Scheduled(initialDelay=1000,fixedRate=5000)
public void doSomething() {
// something that should execute periodically after an initial delay
}
```

The task gets executed initially with a delay and then continues with the specified fixed rate.

@Async- This annotation is used on methods to execute each method in a separate thread. The @Async annotation is provided on a method so that the invocation of that method will occur asynchronously. Unlike methods annotated with @Scheduled, the methods annotated with @Async can take arguments. They will be invoked in the normal way by callers at runtime rather than by a scheduled task.

@Async can be used with both void return type methods and the methods that return a value. However methods with return value must have a Future typed return value.

Event Handling in Spring

You have seen in all the chapters that the core of Spring is the ApplicationContext, which manages the complete life cycle of the beans. The ApplicationContext publishes certain types of events when loading the beans. For example, a *ContextStartedEvent* is published when the context is started and *ContextStoppedEvent* is published when the context is stopped.

Event handling in the *ApplicationContext* is provided through the *ApplicationEvent* class and *ApplicationListener* interface. Hence, if a bean implements the *ApplicationListener*, then every time an *ApplicationEvent* gets published to the ApplicationContext, that bean is notified.

Spring provides the following standard events –

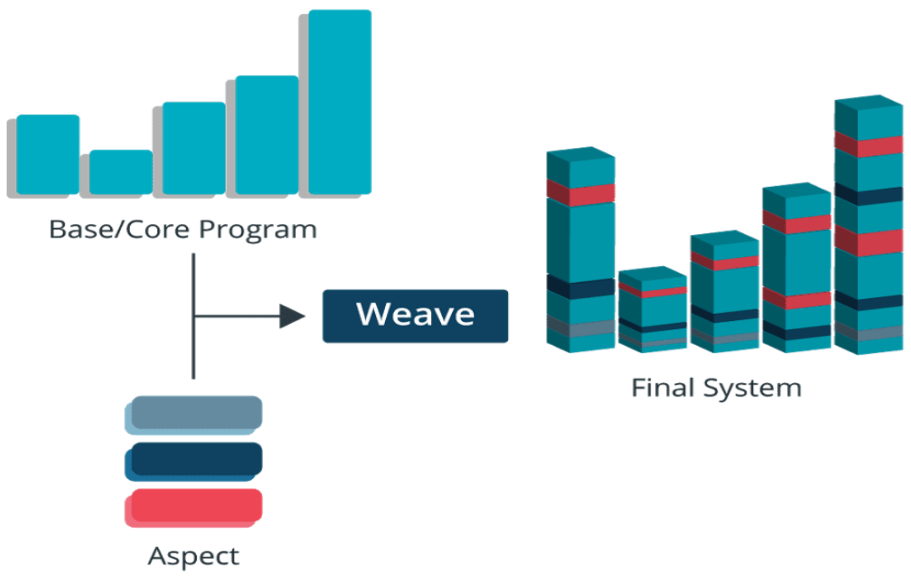
Sr.No.	Spring Built-in Events & Description
1	ContextRefreshedEvent This event is published when the *ApplicationContext* is either initialized or refreshed. This can also be raised using the refresh() method on the *ConfigurableApplicationContext* interface.
2	ContextStartedEvent This event is published when the *ApplicationContext* is started using the start() method on the *ConfigurableApplicationContext* interface. You can poll your database or you can restart any stopped application after receiving this event.
3	ContextStoppedEvent This event is published when the *ApplicationContext* is stopped using the stop() method on the *ConfigurableApplicationContext* interface. You can do required housekeep work after receiving this event.
4	ContextClosedEvent This event is published when the *ApplicationContext* is closed using the close() method on the *ConfigurableApplicationContext* interface. A closed context reaches its end of life; it cannot be refreshed or restarted.

5	RequestHandledEvent This is a web-specific event telling all beans that an HTTP request has been serviced.

Spring's event handling is single-threaded so if an event is published, until and unless all the receivers get the message, the processes are blocked and the flow will not continue. Hence, care should be taken when designing your application if the event handling is to be used.

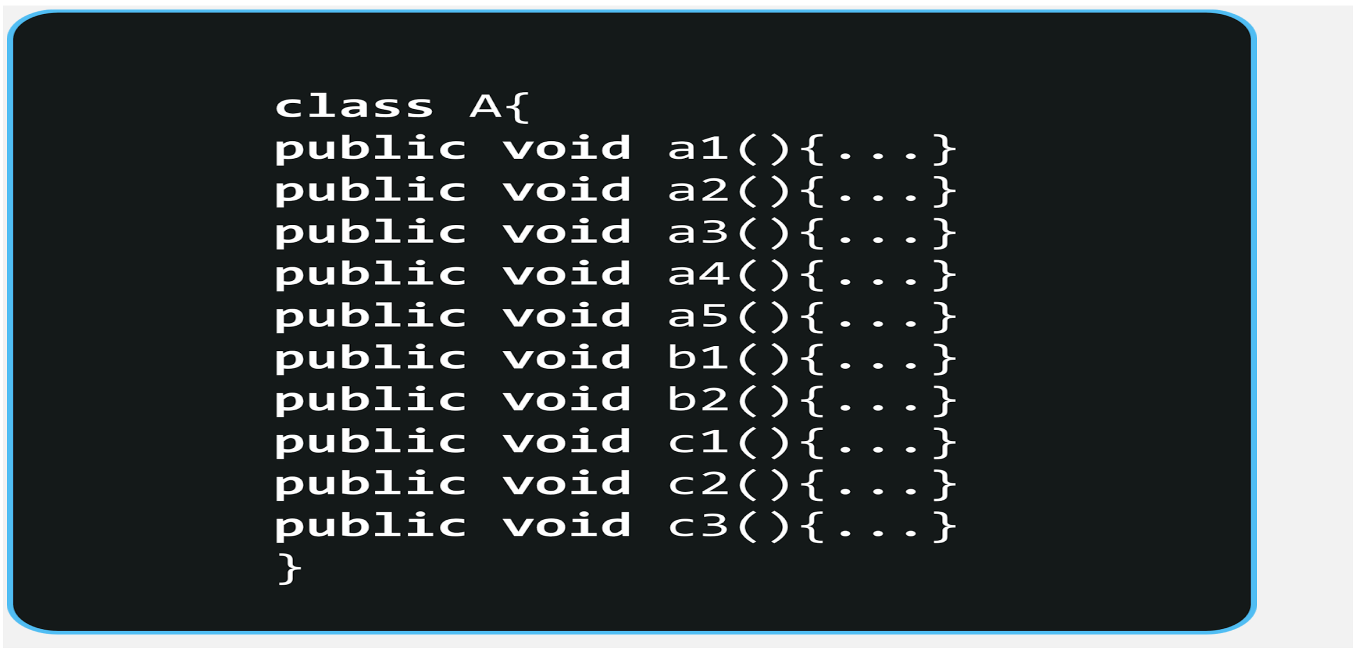
Aspect Oriented Programming (AOP)

Aspect-oriented programming or AOP is a programming technique which allows programmers to modularize crosscutting concerns or behavior that cuts across the typical divisions of responsibility. Examples of cross-cutting concerns can be logging and transaction management. The core of AOP is an *aspect*. It encapsulates behaviors that can affect multiple classes into reusable modules. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect.



Why AOP?

The most important functionality is AOP provides a pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods as shown in the figure:

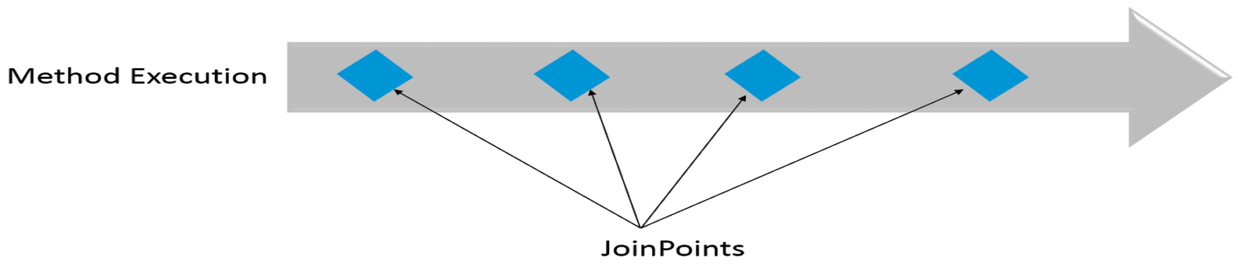


```
class A{
public void a1(){...}
public void a2(){...}
public void a3(){...}
public void a4(){...}
public void a5(){...}
public void b1(){...}
public void b2(){...}
public void c1(){...}
public void c2(){...}
public void c3(){...}
}
```

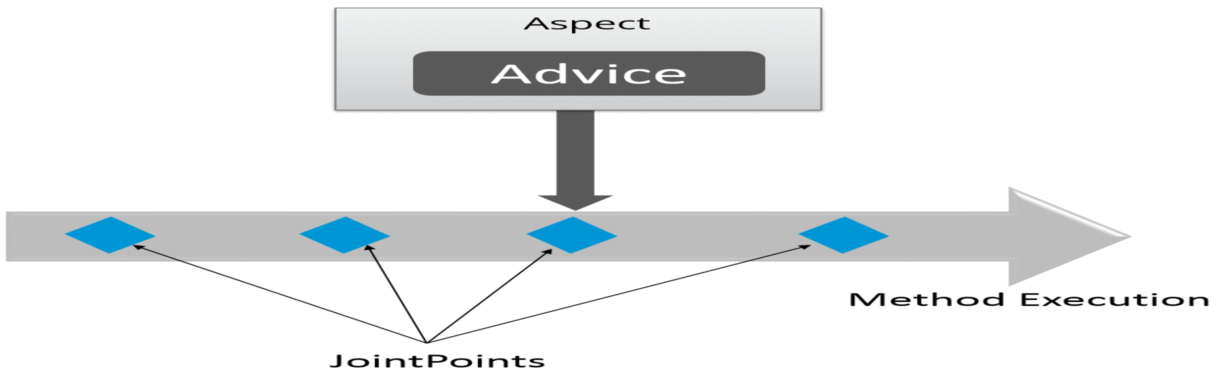
You can see that there are 5 methods that start from a, 2 methods that start from b and 3 methods that start from c. *First, let's understand Scenario-* Here, I have to maintain a log and send notifications after calling methods that start from m. So what is the problem without AOP? Here, We can call methods (that maintain a log and send notification) from the methods starting with a. In such a scenario, we need to write the code in all the 5 methods. But, in case if a client says in future, I don't have to send a notification, you need to change all the methods. It leads to a maintenance problem. So with AOP, we have the solution below.*The solution with AOP-* With AOP, we don't have to call methods from the method. We can simply define the additional concern like maintaining a log, sending notification etc. in the method of a class. Its entry is given in the XML file. Suppose in future, if a client says to remove the notifier functionality, we need to change only in the XML file. So, maintenance is easy in AOP.

1. **Aspect**: Aspect is a modularization of concern which cuts across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. Aspects are implemented using regular classes or regular classes annotated with the @Aspect annotation in Spring Framework.

2. **Joint Point:** A point during the execution of a program is called JoinPoint, such as the execution of a method or the handling of an exception. In Spring AOP, a joinpoint always represents a method execution.



3. Advice: An Action taken by an aspect at a particular joinpoint is known as an Advice. Spring AOP uses advice as an interceptor, maintaining a chain of interceptors "around" the join point.



41. What are the different types of Advices?

Different types of Advices in Spring AOP are:

Before: These types of advice execute before the joinpoint methods and are configured using @Before annotation mark.

After returning: These types of advice execute after the joinpoint methods completes executing normally and are configured using @AfterReturning annotation mark.

After throwing: These types of advice execute only if the joinpoint method exits by throwing an exception and are configured using @AfterThrowing annotation mark.

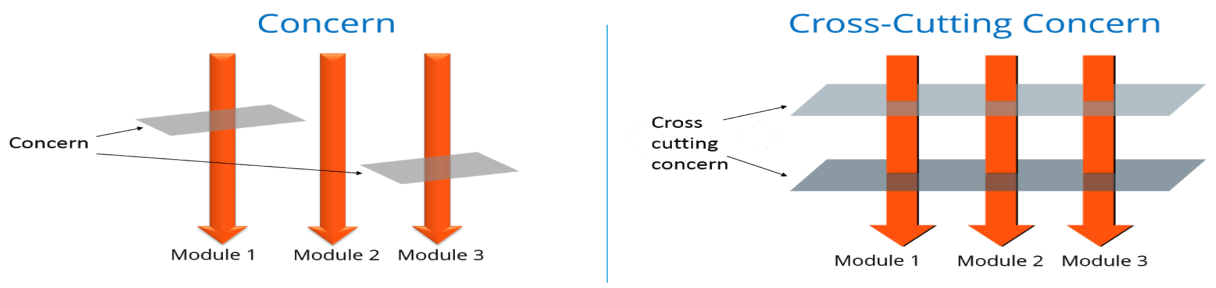
After (finally): These types of advice execute after a joinpoint method, regardless of the method's exit whether normally or exceptional return and are configured using @After annotation mark.

Around: These types of advice execute before and after a joinpoint and are configured using @Around annotation mark.

4. Point out the difference between concern and cross-cutting concern in Spring AOP?

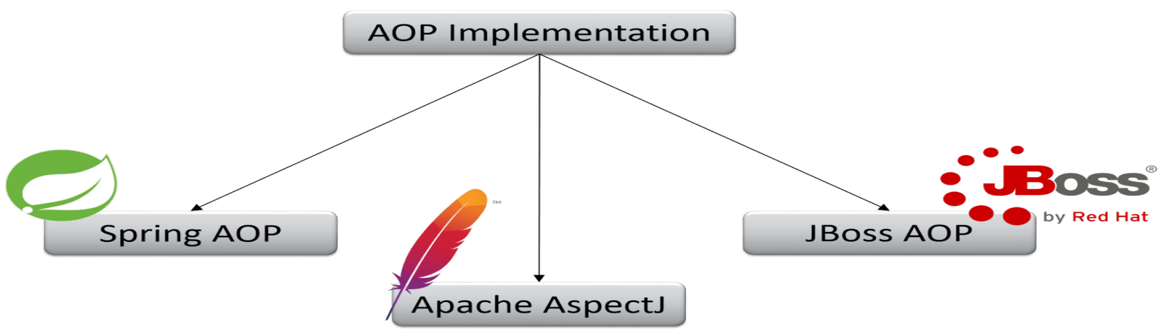
The concern is the behavior we want to have in a particular module of an application. It can be defined as a functionality we want to implement.

The cross-cutting concern is a concern which is applicable throughout the application. This affects the entire application. For example, logging, security and data transfer are the concerns needed in almost every module of an application, thus they are the cross-cutting concerns.



43. What are the different AOP implementations?

Different AOP implementations are depicted by the below diagram:



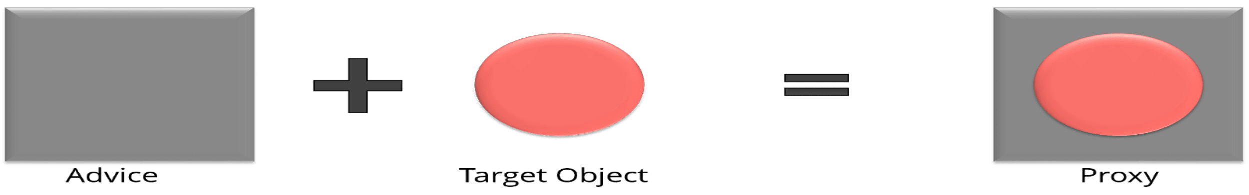
5. What is the difference between Spring AOP and AspectJ AOP?

Spring AOP	**AspectJ AOP**
Runtime weaving through proxy is done	Compile time weaving through AspectJ Java tools is done
It supports only method level PointCut	It supports field level Pointcuts
It is DTD based	It is schema based and Annotation configuration

6. What do you mean by Proxy in Spring Framework?

An object which is created after applying advice to a target object is known as a Proxy.

In case of client objects the target object and the proxy object are the same.



Logging aspect in RESTful web service – spring aop (log requests/responses):

1.Create a RESTFul web service using spring framework.

- Create REST resource BookingController to book hotel & retrieve booking information
- POST resource hotel booking taking required customer information
- GET resource to simulate to retrieval of hotel booking

2. Create a logging aspect.
   - Create logging aspect using spring aop
   - Create join points
   - Create pointcut expressions

3. Usage of pointcut expressions with following advices
    - *Before* advice
    - *After* advice
    - *Around* advice
    - AfterThrowing
    - AfterReturning
4. Generate request response logs.
    - Log request & response information using a logging framework like Log4J.

1.) Spring AOP Logging Aspect:

- Create a logging aspect LoggingHandler using spring aop
- Create pointcut expression
    1. controller
    2. allMethod
    3. loggingPublicOperation
    4. logAnyFunctionWithinResource

```
@Aspect

@Component

public class LoggingHandler {

    Logger log = LoggerFactory.getLogger(this.getClass());

    @Pointcut("within(@org.springframework.stereotype.Controller *)")

    public void controller() {

    }

    @Pointcut("execution(* *.*(..))")

    protected void allMethod() {

    }
```

```
@Pointcut("execution(public * *(..))")

protected void loggingPublicOperation() {

}

@Pointcut("execution(* *.*(..))")

protected void loggingAllOperation() {

}

 @Pointcut("within(org.learn.log..*)")

private void logAnyFunctionWithinResource() {

}

//before -> Any resource annotated with @Controller annotation

//and all method and function taking HttpServletRequest as first parameter

@Before("controller() && allMethod() && args(..,request)")

public void logBefore(JoinPoint joinPoint, HttpServletRequest request) {

     log.debug("Entering in Method :  " + joinPoint.getSignature().getName());

    log.debug("Class Name :  " + joinPoint.getSignature().getDeclaringTypeName());

    log.debug("Arguments :  " + Arrays.toString(joinPoint.getArgs()));

    log.debug("Target class : " + joinPoint.getTarget().getClass().getName());

    if (null != request) {

        log.debug("Start Header Section of request ");

        log.debug("Method Type : " + request.getMethod());

        Enumeration headerNames = request.getHeaderNames();

        while (headerNames.hasMoreElements()) {

            String headerName = headerNames.nextElement();
            String headerValue = request.getHeader(headerName);

            log.debug("Header Name: " + headerName + " Header Value : " + headerValue);

        }

        log.debug("Request Path info :" + request.getServletPath());

        log.debug("End Header Section of request ");

    }

}

//After -> All method within resource annotated with @Controller annotation

// and return a  value

@AfterReturning(pointcut = "controller() && allMethod()", returning = "result")

public void logAfter(JoinPoint joinPoint, Object result) {

    String returnValue = this.getValue(result);

    log.debug("Method Return value : " + returnValue);

}

//After -> Any method within resource annotated with @Controller annotation

// throws an exception ...Log it

@AfterThrowing(pointcut = "controller() && allMethod()", throwing = "exception")

public void logAfterThrowing(JoinPoint joinPoint, Throwable exception) {

    log.error("An exception has been thrown in " + joinPoint.getSignature().getName() + " ()");

    log.error("Cause : " + exception.getCause());

}

//Around -> Any method within resource annotated with @Controller annotation

@Around("controller() && allMethod()")
```

```
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    long start = System.currentTimeMillis();
    try {
        String className = joinPoint.getSignature().getDeclaringTypeName();
        String methodName = joinPoint.getSignature().getName();
        Object result = joinPoint.proceed();
        long elapsedTime = System.currentTimeMillis() - start;
        log.debug("Method " + className + "." + methodName + " ()" + " execution time : "
                + elapsedTime + " ms");

        return result;
    } catch (IllegalArgumentException e) {
        log.error("Illegal argument " + Arrays.toString(joinPoint.getArgs()) + " in "
                + joinPoint.getSignature().getName() + "()");
        throw e;
    }
}
private String getValue(Object result) {
    String returnValue = null;
    if (null != result) {
        if (result.toString().endsWith("@" + Integer.toHexString(result.hashCode()))) {
            returnValue = ReflectionToStringBuilder.toString(result);
        } else {
                returnValue = result.toString();

            }

        }

        return returnValue;

    }

}


@Controller
@RequestMapping("/hotel")
public class BookingController {
    private static final Logger logger = LoggerFactory.getLogger(BookingController.class);

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home(Locale locale, Model model) {
        logger.info("Welcome home! The client locale is {}.", locale);

        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG,
DateFormat.LONG, locale);

        String formattedDate = dateFormat.format(date);

        model.addAttribute("serverTime", formattedDate );

        return "hotel";
    }

    @RequestMapping(value="/book", method=RequestMethod.POST)
    public String bookHotel(
            @RequestParam(value = "name",required=false) String name,
            @RequestParam(value = "city", required = false) String city,
            @RequestParam(value = "purpose",required=false) String purpose,
            @RequestParam(value = "idproof", required = false) String idproof,
            Model model,  HttpServletRequest request){

        //Save the required information in data base
        //......
        //......
        //Send the response back
        model.addAttribute("name", name );
        model.addAttribute("city", city );
        model.addAttribute("purpose", purpose );
        model.addAttribute("idproof", idproof );

        return "customerDetails";
    }

    @RequestMapping(value="/book", method=RequestMethod.GET)
    public String bookHotel(
            @RequestParam(value = "id",required=false) String id,
            Model model,  HttpServletRequest request){

        //get the required information in data base for customer I
        String randomName = UUID.randomUUID().toString().substring(0,4);
        String randomCity = UUID.randomUUID().toString().substring(0,4);
        String randomPurpose = UUID.randomUUID().toString().substring(0,4);
        String randomIdProof = UUID.randomUUID().toString().substring(0,4);

        //Send the response back
        model.addAttribute("name", "Name "+randomName );
        model.addAttribute("city", "City "+randomCity );
        model.addAttribute("purpose", "Purpose "+randomPurpose);
        model.addAttribute("idproof", "IdProof "+randomIdProof );

        return "customerDetails";
    }
}
```
Spring Logging Aspect execution results:

Default Home Page

http://localhost:9095/log/hotel/

Logs are :

```
INFO : Welcome home! The client locale is en_US.

DEBUG: Method org.learn.log.BookingController.home () execution time : 1 ms

DEBUG: Method Return value : hotel
```

Post request RESTFul Service

http://localhost:9095/log/hotel/book

Logs are :

```
DEBUG: Entering in Method : bookHotel
```

```
DEBUG: Class Name : org.learn.log.BookingController

DEBUG: Arguments : [Scott, Customer CityName, Personal, Security Number, {},
org.apache.catalina.connector.RequestFacade@7886596e]

DEBUG: Target class : org.learn.log.BookingController

DEBUG: Start Header Section of request

DEBUG: Method Type : POST

DEBUG: Header Name: host Header Value : localhost:9095

DEBUG: Header Name: connection Header Value : keep-alive

DEBUG: Header Name: content-length Header Value : 0

DEBUG: Header Name: user-agent Header Value : Mozilla/5.0 (Windows NT 6.3; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36

DEBUG: Header Name: cache-control Header Value : no-cache

DEBUG: Header Name: origin Header Value : chrome-extension://mkhojklkhkdaghjjfdnphfphiaiohkef

DEBUG: Header Name: accept Header Value : */*

DEBUG: Header Name: accept-encoding Header Value : gzip, deflate

DEBUG: Header Name: accept-language Header Value : en-US,en;q=0.8

DEBUG: Request Path info :/hotel/book

DEBUG: End Header Section of request

DEBUG: Method org.learn.log.BookingController.bookHotel () execution time : 148 ms

DEBUG: Method Return value : customerDetails
```

GET Request – RESTFul WebService

http://localhost:9095/log/hotel/book?id=1

Logs are :

```
DEBUG: Entering in Method : bookHotel

DEBUG: Class Name : org.learn.log.BookingController
```

```
DEBUG: Arguments : [1, {}, org.apache.catalina.connector.RequestFacade@777d0dd6]

DEBUG: Target class : org.learn.log.BookingController

DEBUG: Start Header Section of request

DEBUG: Method Type : GET

DEBUG: Header Name: accept Header Value : image/jpeg, application/x-ms-application, image/gif,
application/xaml+xml, image/pjpeg, application/x-ms-xbap, */*

DEBUG: Header Name: accept-language Header Value : en-US

DEBUG: Header Name: ua-cpu Header Value : AMD64

DEBUG: Header Name: accept-encoding Header Value : gzip, deflate

DEBUG: Header Name: user-agent Header Value : Mozilla/5.0 (Windows NT 6.2; Win64; x64;
Trident/7.0; MASPJS; rv:11.0) like Gecko

DEBUG: Header Name: host Header Value : localhost:9095

DEBUG: Header Name: connection Header Value : Keep-Alive

DEBUG: Request Path info :/hotel/book

DEBUG: End Header Section of request

DEBUG: Method org.learn.log.BookingController.bookHotel () execution time : 8 ms

DEBUG: Method Return value : customerDetails
```

We can download the complete code. Refer ReadMe.md file under project directory for more details. In our pom we have kept commonly used jars required to develop any web application.

https://makeinjava.com/logging-aspect-restful-web-service-spring-aop-request-response/

Multi-Threading in Spring Boot Using CompletableFuture:

Multi-threading is similar to multi-tasking, but it enables the processing of executing multiple threads simultaneously, rather than multiple processes. CompletableFuture, which was introduced in Java 8, provides an easy way to write asynchronous, non-blocking, and multi-threaded code.The Future interface was introduced in Java 5 to handle asynchronous computations. But, this interface did not have any methods to combine multiple asynchronous computations and handle all the possible errors. The CompletableFuture implements Future interface, it can combine multiple asynchronous computations, handle possible errors and offers much more capabilities.

Let's get down to writing some code and see the benefits.

In this article, we will be using sample data about cars. We will create a JPA entity Car and a corresponding JPA repository.

```
@Data
@EqualsAndHashCode
@Entity
public class Car implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column (name = "ID", nullable = false)
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    private long id;

    @NotNull
    @Column(nullable=false)
    private String manufacturer;

    @NotNull
    @Column(nullable=false)
    private String model;

    @NotNull
    @Column(nullable=false)
    private String type;
}

@Repository
public interface CarRepository extends JpaRepository<Car, Long> {

}
```

Let us now create a configuration class that will be used to enable and configure the asynchronous method execution.

```
@Configuration

@EnableAsync

public class AsyncConfiguration {

    private static final Logger LOGGER = LoggerFactory.getLogger(AsyncConfiguration.class);

    @Bean (name = "taskExecutor")

    public Executor taskExecutor() {

        LOGGER.debug("Creating Async Task Executor");

        final ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

        executor.setCorePoolSize(2);
        executor.setMaxPoolSize(2);

        executor.setQueueCapacity(100);

        executor.setThreadNamePrefix("CarThread-");

        executor.initialize();

        return executor;

    }


}
```

The @EnableAsync annotation enables Spring's ability to run @Async methods in a background thread pool. The bean taskExecutor helps to customize the thread executor such as configuring the number of threads for an application, queue limit size, and so on. Spring will specifically look for this bean when the server is started. If this bean is not defined, Spring will create SimpleAsyncTaskExecutor by default. We will now create a service and @Async methods.

```
@Service

public class CarService {

private static final Logger LOGGER =LoggerFactory.getLogger(CarService.class);

@Autowired

private CarRepository carRepository;

    @Async

    public CompletableFuture<List<Car>> saveCars(final MultipartFile file) throws Exception {

        final long start = System.currentTimeMillis();

        List<Car> cars = parseCSVFile(file);

        LOGGER.info("Saving a list of cars of size {} records", cars.size());

        cars = carRepository.saveAll(cars);

        LOGGER.info("Elapsed time: {}", (System.currentTimeMillis() - start));

        return CompletableFuture.completedFuture(cars);

    }
    private List<Car> parseCSVFile(final MultipartFile file) throws Exception {

      final List<Car> cars=new ArrayList<>();

        try {

            try (final BufferedReader br = new BufferedReader(new
InputStreamReader(file.getInputStream()))) {

                String line;

                while ((line=br.readLine()) != null) {

                    final String[] data=line.split(";");

                    final Car car=new Car();

                    car.setManufacturer(data[0]);

                    car.setModel(data[1]);

                    car.setType(data[2]);

                    cars.add(car);

                }

                return cars;

            }

        } catch(final IOException e) {

            LOGGER.error("Failed to parse CSV file {}", e);

            throw new Exception("Failed to parse CSV file {}", e);

        }

    }

    @Async

    public CompletableFuture<List<Car>> getAllCars() {                    LOGGER.info("Request to get a
list of cars");

        final List<Car> cars = carRepository.findAll();

        return CompletableFuture.completedFuture(cars);
    }

}
```

Here, we have two @Async methods: saveCar() and getAllCars(). The first one accepts a multipart file, parses it, and stores the data in the database. The second method reads the data from the database. Both methods are returning a new CompletableFuture that was already completed with the given values.

Let us create a Rest Controller and provide some endpoints:

```
@RestController

@RequestMapping("/api/car")

public class CarController {

    private static final Logger LOGGER = LoggerFactory.getLogger(CarController.class);

    @Autowired

    private CarService carService;

    @RequestMapping (method = RequestMethod.POST,
consumes={MediaType.MULTIPART_FORM_DATA_VALUE},

            produces={MediaType.APPLICATION_JSON_VALUE})

    public @ResponseBody ResponseEntity uploadFile(

            @RequestParam (value = "files") MultipartFile[] files) {

        try {

            for(final MultipartFile file: files) {

                carService.saveCars(file);

            }

            return ResponseEntity.status(HttpStatus.CREATED).build();

        } catch(final Exception e) {

            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();

        }
    }

    @RequestMapping (method = RequestMethod.GET,
consumes={MediaType.APPLICATION_JSON_VALUE},

       produces={MediaType.APPLICATION_JSON_VALUE})

    public @ResponseBody CompletableFuture<ResponseEntity> getAllCars() {

        return carService.getAllCars().<ResponseEntity>thenApply(ResponseEntity::ok)

                .exceptionally(handleGetCarFailure);

    }

    private static Function<Throwable, ResponseEntity<? extends List<Car>>> handleGetCarFailure =
throwable -> {

        LOGGER.error("Failed to read records: {}", throwable);

        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();

    };

}
```

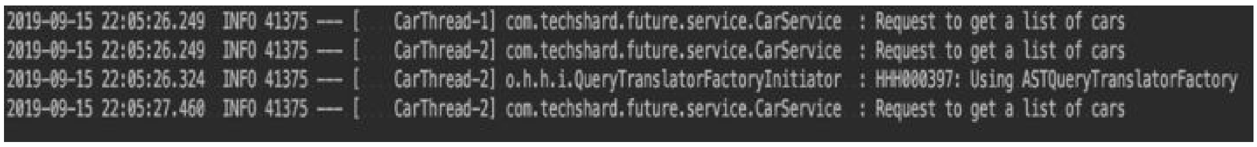
The first REST endpoint accepts a list of multipart files. The second endpoint is to read the data. As you notice the GET endpoint, there is some difference in the return statement. We are returning a list of cars and we are also handling exceptions.The function handleGetCarFailure is invoked when the CompletableFuture completes exceptionally, otherwise, if this CompletableFuture completes normally, it returns a list of cars to the client.

Now, just modify the GET endpoint as follows:

```
@RequestMapping (method = RequestMethod.GET,
consumes={MediaType.APPLICATION_JSON_VALUE},

            produces={MediaType.APPLICATION_JSON_VALUE})

    public @ResponseBody ResponseEntity getAllCars() {

        try {

             CompletableFuture<List<Car>> cars1=carService.getAllCars();

            CompletableFuture<List<Car>> cars2=carService.getAllCars();
            CompletableFuture<List<Car>> cars3=carService.getAllCars();

            CompletableFuture.allOf(cars1, cars2, cars3).join();

            return ResponseEntity.status(HttpStatus.OK).build();

        } catch(final Exception e) {

            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();

        }

    }
```

Here, we are calling the Async method 3 times. The CompletableFuture.allOf() will wait until all the CompletableFutures have been completed, and join() will combine the results. Note that this is just for demonstration purposes. Add Thread.sleep(1000L) in getAllCars() of the CarService class. We are giving a delay of 1 second just for testing purposes.

Restart the application and test GET endpoint again.



```
2019-09-15 22:05:26.249  INFO 41375 --- [    CarThread-1] com.techshard.future.service.CarService  : Request to get a list of cars
2019-09-15 22:05:26.249  INFO 41375 --- [    CarThread-2] com.techshard.future.service.CarService  : Request to get a list of cars
2019-09-15 22:05:26.324  INFO 41375 --- [    CarThread-2] o.h.h.i.QueryTranslatorFactoryInitiator  : HHH000397: Using ASTQueryTranslatorFactory
2019-09-15 22:05:27.460  INFO 41375 --- [    CarThread-2] com.techshard.future.service.CarService  : Request to get a list of cars
```

As you see in the above screenshot, the first two calls to the Async method have started simultaneously. The third call has started with a delay of 1 second.

Remember that we have configured only 2 threads that can be used simultaneously. When at least one of the two threads becomes free, the third request to the Async method will be made.

https://dzone.com/articles/multi-threading-in-spring-boot-using-completablefu

Spring Boot Actuator

Spring Boot Actuator is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use HTTP and JMX endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the Spring Boot actuator.

Spring Boot Actuator Features

There are three main features of Spring Boot Actuator:

- Endpoints
- Metrics
- Audit

Endpoint: The actuator endpoints allow us to monitor and interact with the application. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application chooses HTTP, where the Id of the endpoint, along with the prefix of /actuator, is mapped to a URL.

For example, the /health endpoint provides the basic health information of an application. The actuator, by default, mapped it to /actuator/health.

Metrics: Spring Boot Actuator provides dimensional metrics by integrating with the micrometer. The micrometer is integrated into Spring Boot. It is the instrumentation library powering the delivery of application metrics from Spring. It provides vendor-neutral interfaces for timers, gauges, counters, distribution summaries, and long task timers with a dimensional data model.

Audit: Spring Boot provides a flexible audit framework that publishes events to an AuditEventRepository. It automatically publishes the authentication events if spring-security is in execution.

ORM

ORM stands for Object-Relational Mapping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.

An ORM system has the following advantages over plain JDBC –

1. Let's business code access objects rather than DB tables.
2. Hides details of SQL queries from OO logic.
3. Based on JDBC 'under the hood.
4. No need to deal with the database implementation.
5. Entities based on business concepts rather than database structure.
6. Transaction management and automatic key generation.

An ORM solution consists of the following four entities –

1. An API to perform basic CRUD operations on objects of persistent classes.
2. A language or API to specify queries that refer to classes and properties of classes.
3. A configurable facility for specifying mapping metadata.
4. A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.

What is JDBC?

JDBC stands for Java Database Connectivity. It provides a set of Java API for accessing the relational databases from the Java program. These Java APIs enable Java programs to execute SQL statements and interact with any SQL compliant database.

JDBC provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification.

Why Object Relational Mapping (ORM)?

When we work with an object-oriented system, there is a mismatch between the object model and the relational database. RDBMSs represent data in a tabular format whereas object-oriented languages, such as Java or C# represent it as an interconnected graph of objects.

Consider the following Java Class with proper constructors and associated public function –

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;
  public Employee() {}
  public Employee(String fname, String lname, int salary) {
    this.first_name = fname;
    this.last_name = lname;
    this.salary = salary;
  }
```

```
   public int getId() {
   return id;
   }

   public String getFirstName() {
      return first_name;
   }

   public String getLastName() {
      return last_name;
   }

   public int getSalary() {
      return salary;
   }
}
```

Consider the above objects are to be stored and retrieved into the following RDBMS table –

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

First problem, what if we need to modify the design of our database after having developed a few pages or our application? Second, loading and storing objects in a relational database exposes us to the following five mismatch problems –

- **Granularity**:Sometimes you will have an object model, which has more classes than the number of corresponding tables in the database.
- **Inheritance**: RDBMSs do not define anything similar to Inheritance, which is a natural paradigm in object-oriented programming languages.
- **Identity**: An RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity (a==b) and object equality (a.equals(b)).

- **Associations**: Object-oriented languages represent associations using object references whereas an RDBMS represents an association as a foreign key column.
- **Navigation**: The ways you access objects in Java and in RDBMS are fundamentally different.

The Object-Relational Mapping (ORM) is the solution to handle all the above impedance mismatches.

Java ORM Frameworks

There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database.

- Enterprise JavaBeans Entity Beans
- Java Data Objects
- Castor
- TopLink
- Spring DAO
- Hibernate
- And many more

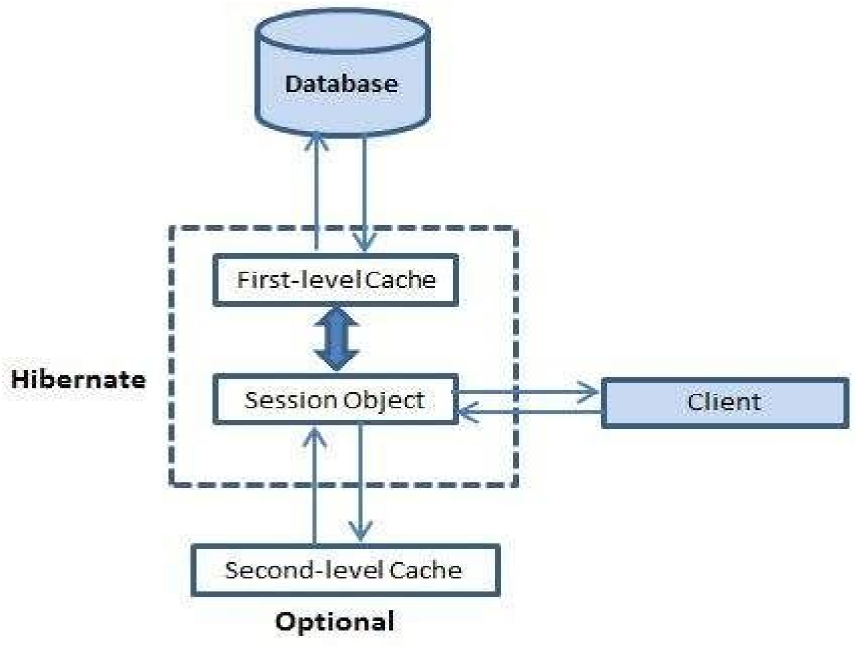
Hibernate

Hibernate is a high-performance Object/Relational persistence and query service. Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities.

Hibernate Advantages

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
- Provides simple APIs for storing and retrieving Java objects directly to and from the database.
- If there is change in the database or in any table, then you need to change the XML file properties only.
- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- Manipulates Complex associations of objects of your database.
- Minimizes database access with smart fetching strategies.
- Provides simple querying of data.

Hibernate - Caching



First-level Cache: The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

Second-level Cache: Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

Any third-party cache can be used with Hibernate. An org.hibernate.cache.CacheProvider interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.

Query-level Cache: Hibernate also implements a cache for query resultsets that integrates closely with the second-level cache.

This is an optional feature and requires two additional physical cache regions that hold the cached query results and the timestamps when a table was last updated. This is only useful for queries that are run frequently with the same parameters.

JDBC vs Hibernate:

JDBC is an acronym for Java DataBase Connectivity and is a technology for interaction of java application and its objects with a database. Hibernate on the other hand is a java based framework which also facilitates the interaction of application objects with a database but *in a completely different approach*.

Below are listed some differences between the two. The points do not intend to indicate which is better, they just compare the various aspects of the method of operation of both. *The headers before each row indicate the area of comparison*.

1. **Type**: JDBC is a technology for persistence of data whereas Hibernate is a technology for persistence of Objects.
2. **Query Language:** JDBC uses SQL (Structured Query Language) to interact with the database whereas Hibernate uses HQL(Hibernate Query Language) and SQL.
3. **Database Dependence:** JDBC uses SQL queries for database interaction which vary with the type of database. Hence, when the underlying database changes, queries have to be updated whereas Hibernate uses HQL which is independent of databases. Hence, changing application databases requires no change in application code.
4. **Caching**: JDBC provides no caching support. It has to be separately implemented and integrated with the application whereas Hibernate provides two levels of caching. Separate implementation is not required.
5. **Connection pooling:** JDBC by itself does not provide any connection pooling facility. It has to be implemented by the developer using java or third party libraries such as DBCP from Apache whereas With Hibernate you can use connection pooling merely by adding a dependency in your application build file and configuring it in the hibernate configuration file. Example is c3p0, which has a hibernate package.
6. **Multiple Table data fetch:** If you have nested objects (*whose data is distributed over multiple tables*), then SQL queries to fetch the data can become too complex. For example, suppose you have employee's official details in one table and his personal details in some other table then the query to fetch complete employee data will be quite long and cumbersome whereas Hibernate's *One-To-One*, *One-To-Many* and *Many-To-Many* relations manage this very well and fetching top level entities will automatically fetch the data from all related tables automatically.
7. **Object-Database table Relationship:** In order to convert database records into an object, explicit code needs to be written where each column data will be mapped

to its respective java class field whereas Hibernate automatically does this via configuration file or annotations. Moreover, column names are automatically mapped with the respective entity fields if they share the same name.

8. **Primary Key Generation:** With JDBC we cannot auto generate primary key value for a record, we need to explicitly set it before saving whereas Hibernate provides automatic primary key generation so that when we save a record, primary key value for that record is automatically generated.
9. **Cascading**: An object may contain another object such as a Test Script and a list of its test steps. When you save (or delete) a test script, you will obviously want the test steps to be saved (or deleted) as well. This is called Cascading. With JDBC, this has to be manually handled by the developer. Save the test script, then set the id of this test script in each of the test steps as a foreign key; delete the test script, then delete all test steps related to the given test script id and so on whereas Hibernate manages all this stuff by itself. You just need to apply appropriate Cascade and CascadeType either in the form of annotations or in configuration files.
10. **Automatic Database Creation:** You cannot create a non-existent database or table if you are using JDBC provided you don't execute a query to do that whereas With Hibernate you can create a database or table at application startup by using appropriate value of configuration property hbm2ddl.auto.
11. **Query Statistics:** With JDBC, it is not possible to figure out the time taken by a query unless you manually calculate the time or set logging at database level whereas Hibernate can easily generate the statistics for query execution in an application. It can also tell the number of database connections, statements utilized and the time taken for these operations, the cache usage and number of flushes. All required is the configuration hibernate.generate_statistics to be set.
12. **Row Versioning:** Versioning of rows is not possible using JDBC. Although this can also be managed explicitly by incrementing a particular column value after every update. But again it's a manual effort whereas Hibernate manages versioning automatically provided that you have configured a field to be used as a version by annotating it with @Version or by setting a field as versioned using in XML configuration.

Annotated Class Example

As I mentioned above while working with Hibernate Annotation, all the metadata is clubbed into the POJO java file along with the code, this helps the user to understand the table structure and POJO simultaneously during the development.

Consider we are going to use the following EMPLOYEE table to store our objects –

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Following is the mapping of Employee class with annotations to map objects with the defined EMPLOYEE table –

```
import javax.persistence.*;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;

   @Column(name = "first_name")
   private String firstName;

   @Column(name = "last_name")
   private String lastName;

   @Column(name = "salary")
   private int salary;

   public Employee() {}

   public int getId() {
      return id;
   }

   public void setId( int id ) {
      this.id = id;
   }

   public String getFirstName() {
      return firstName;
   }

   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }

   public String getLastName() {
      return lastName;
   }

   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }

   public int getSalary() {
      return salary;
   }

   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

Hibernate detects that the @Id annotation is on a field and assumes that it should access properties of an object directly through fields at runtime. If you placed the @Id annotation on the getId() method, you would enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy.

Following section will explain the annotations used in the above class.

@Entity Annotation: The EJB 3 standard annotations are contained in the javax.persistence package, so we import this package as the first step. Second, we used the @Entity annotation to the Employee class, which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

@Table Annotation: The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.

The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now, we are using just table names, which is EMPLOYEE.

@Id and @GeneratedValue Annotations: Each entity bean will have a primary key, which you annotate on the class with the @Id annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the @GeneratedValue annotation, which takes two parameters strategy and generator that I'm not going to discuss here, so let us use only the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

@Column Annotation : The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes −

- name attribute permits the name of the column to be explicitly specified.

- length attribute permits the size of the column used to map a value particularly for a String value.
- nullable attribute permits the column to be marked NOT NULL when the schema is generated.
- a unique attribute permits the column to be marked as containing only unique values.

What is JPA?

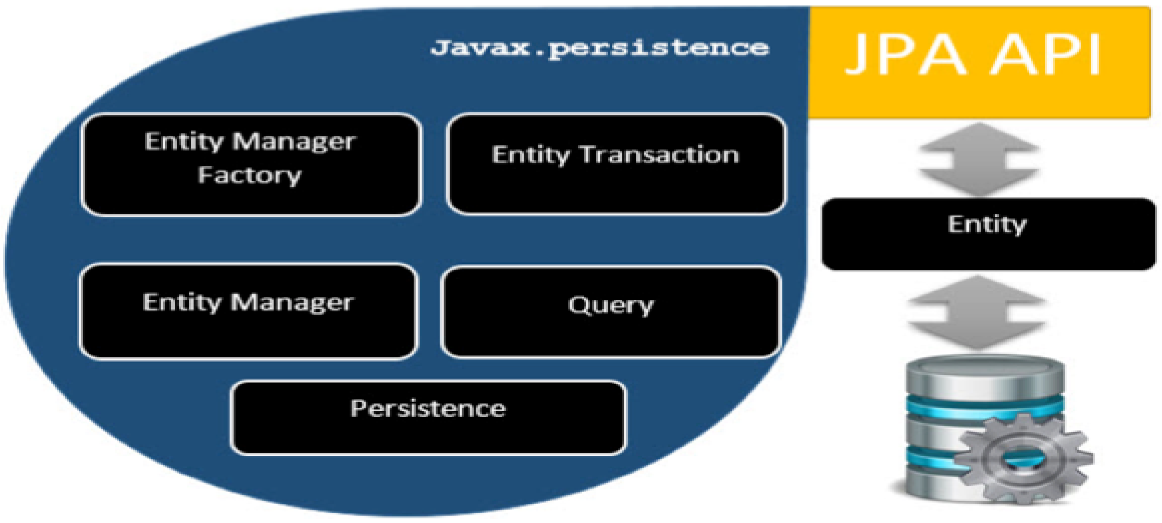
Not all Java objects need to be persisted, but most applications persist key business objects. The JPA specification lets you define *which* objects should be persisted, and *how* those objects should be persisted in your Java applications.

By itself, JPA is not a tool or framework; rather, it defines a set of concepts that can be implemented by any tool or framework. While JPA's object-relational mapping (ORM) model was originally based on Hibernate, it has since evolved. Likewise, while JPA was originally intended for use with relational/SQL databases, some JPA implementations have been extended for use with NoSQL datastores.

The following table describes each of the units shown in the above architecture.

Units	**Description**
EntityManagerFactory	This is a factory class of EntityManager. It creates and manages multiple EntityManager instances.
EntityManager	It is an Interface, it manages the persistence operations on objects. It works like a factory for Query instances.
Entity	Entities are the persistence objects, stored as records in the database.
EntityTransaction	It has a one-to-one relationship with EntityManager. For each EntityManager, operations are maintained by the EntityTransaction class.

Persistence	This class contains static methods to obtain EntityManagerFactory instances.
Query	This interface is implemented by each JPA vendor to obtain relational objects that meet the criteria.



Entity: A class which should be persisted in a database must be annotated with javax.persistence.Entity. Such a class is called Entity. JPA uses a database table for every entity. Persisted instances of the class will be represented as one row in the table.

All entity classes must define a primary key, must have a non-arg constructor and or not allowed to be final. Keys can be a single field or a combination of fields.

JPA allows auto-generating the primary key in the database via the @GeneratedValue annotation.By default, the table name corresponds to the class name. You can change this with the addition to the annotation @Table(name="NEWTABLENAME").

Persistence of fields: The fields of the Entity will be saved in the database. JPA can use either your instance variables (fields) or the corresponding getters and setters to access the fields. You are not allowed to mix both methods. If you want to use the setter and getter methods the Java class must follow the JavaBean naming conventions. JPA persists per default on all fields of an Entity, if fields should not be saved they must be marked with @Transient.

By default each field is mapped to a column with the name of the field. You can change the default name via @Column (name="newColumnName").

The following annotations can be used.

@Id	**Identifies the unique ID of the database entry**
@GeneratedValue	Together with an ID this annotation defines that this value is generated automatically.
@Transient	Field will not be saved in database

Relationship Mapping- JPA allows to define relationships between classes, e.g. it can be defined that a class is part of another class (containment). Classes can have one to one, one to many, many to one, and many to many relationships with other classes.
A relationship can be bidirectional or unidirectional, e.g. in a bidirectional relationship both classes store a reference to each other while in an unidirectional case only one class has a reference to the other class. Within a bidirectional relationship you need to specify the owning side of this relationship in the other class with the attribute "mappedBy", e.g. @ManyToMany(mappedBy="attributeOfTheOwningClass".

Relationship annotations:

- @OneToOne
- @OneToMany
- @ManyToOne

- @ManyToMany

Entity Manager: The entity manager javax.persistence.EntityManager provides the operations from and to the database, e.g. find objects, persist them, remove objects from the database, etc.

In a JavaEE application the entity manager is automatically inserted in the web application. Outside JavaEE you need to manage the entity manager yourself.

Entities which are managed by an Entity Manager will automatically propagate these changes to the database (if this happens within a commit statement). If the Entitymanager is closed (via close()) then the managed entities are in a detached state. If synchronize them again with the database a Entity Manager provides the merge() method.

Spring Data

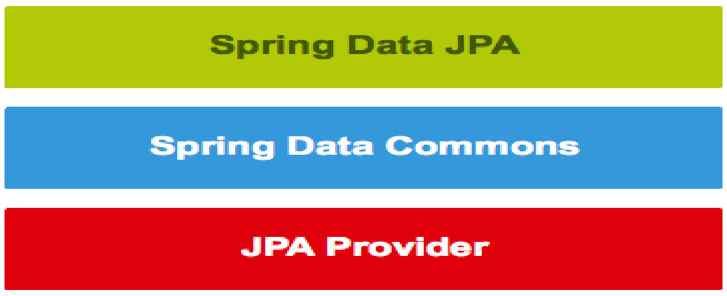
Spring Data is a Spring-based programming model for data access. It reduces the amount of code needed for working with databases and datastores. It consists of several modules. The Spring Data JPA simplifies the development of Spring applications that use JPA technology.With Spring Data, we define a repository interface for each domain entity in the application. A repository contains methods for performing CRUD operations, sorting and pagination data. @Repository is a marker annotation, which indicates that the underlying interface is a repository. A repository is created by extending specific repository interfaces, such as CrudRepository, PagingAndSortingRepository, or JpaRepository.Spring Data has advanced integration with Spring MVC controllers and provides dynamic query derivation from repository method names.

What is Spring Data JPA?

Spring Data JPA is not a JPA provider. It is a library / framework that adds an extra layer of abstraction on the top of our JPA provider. If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following:

- Spring Data JPA provides support for creating JPA repositories by extending the Spring Data repository interfaces.
- Spring Data Commons provides the infrastructure that is shared by the datastore specific Spring Data projects.
- The JPA Provider implements the Java Persistence API.

The following figure illustrates the structure of our repository layer:



Additional Reading: Spring Data JPA versus JPA: What's the difference?

At first it seems that Spring Data JPA makes our application more complicated, and in a way that is true. It does add an additional layer to our repository layer, but at the same time it frees us from writing any boilerplate code.

That sounds like a good tradeoff. Right?

Introduction to Spring Data Repositories

The power of Spring Data JPA lies in the repository abstraction that is provided by the Spring Data Commons project and extended by the datastore specific sub projects.

We can use Spring Data JPA without paying any attention to the actual implementation of the repository abstraction, but we have to be familiar with the Spring Data repository interfaces. These interfaces are described in the following:

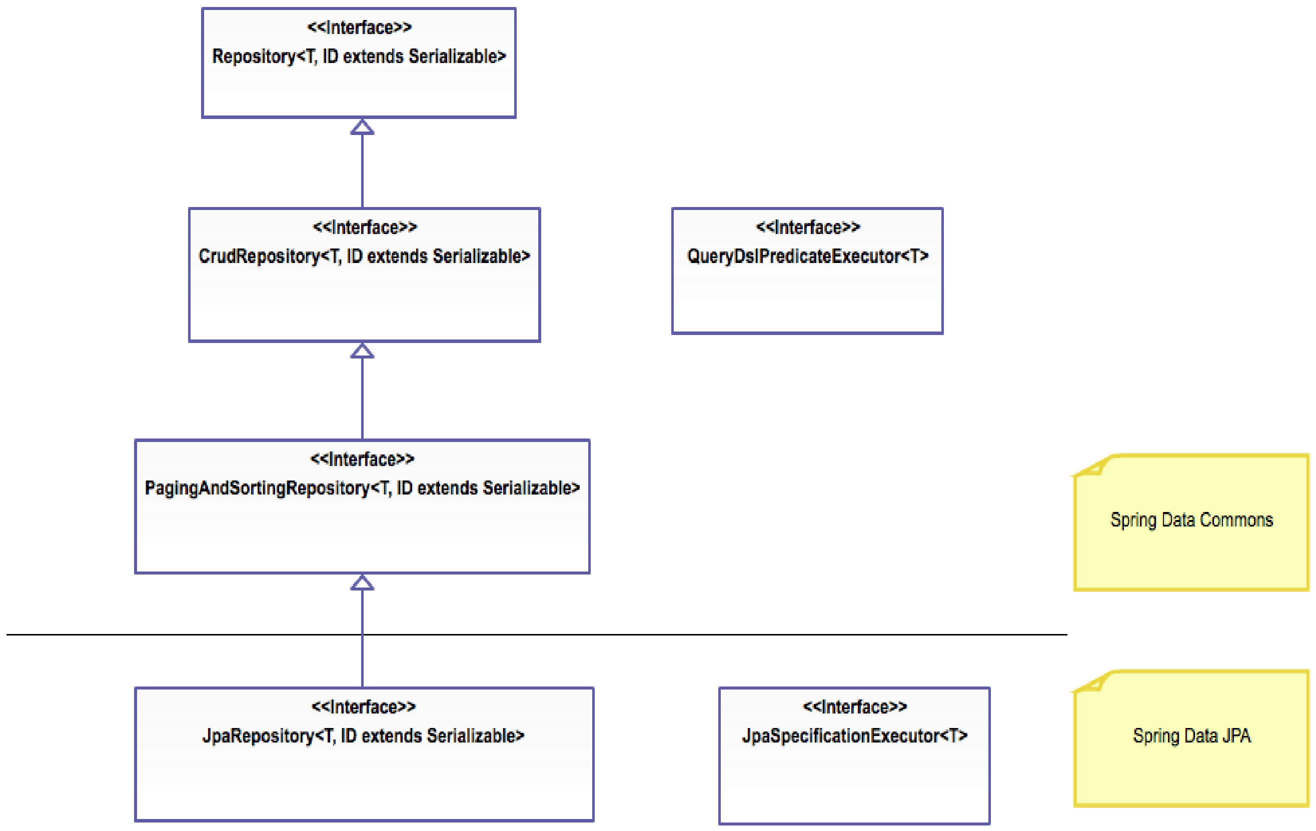
First, the Spring Data Commons project provides the following interfaces:

- The *Repository<T, ID extends Serializable>* interface is a marker interface that has two purposes:
    1. It captures the type of the managed entity and the type of the entity's id.
    2. It helps the Spring container to discover the "concrete" repository interfaces during classpath scanning.
- The *CrudRepository<T, ID extends Serializable>* interface provides CRUD operations for the managed entity.
- The *PagingAndSortingRepository<T, ID extends Serializable>* interface declares the methods that are used to sort and paginate entities that are retrieved from the database.
- The *QueryDslPredicateExecutor<T>* interface is not a "repository interface". It declares the methods that are used to retrieve entities from the database by using QueryDsl *Predicate* objects.

Second, the Spring Data JPA project provides the following interfaces:

- The *JpaRepository<T, ID extends Serializable>* interface is a JPA specific repository interface that combines the methods declared by the common repository interfaces behind a single interface.
- The *JpaSpecificationExecutor<T>* interface is not a "repository interface". It declares the methods that are used to retrieve entities from the database by using *Specification<T>* objects that use the JPA criteria API.

The repository hierarchy looks as follows:



<<Interface>>
Repository<T, ID extends Serializable>
<<Interface>>
CrudRepository<T, ID extends Serializable>
<<Interface>>
QueryDslPredicateExecutor<T>
<<Interface>>
PagingAndSortingRepository<T, ID extends Serializable>
Spring Data Commons
<<Interface>>
JpaRepository<T, ID extends Serializable>
<<Interface>>
JpaSpecificationExecutor<T>
Spring Data JPA

That is nice, but how can we use them?

That is a fair question. The next parts of this tutorial will answer to that question, but essentially we have to follow these steps:

1. Create a repository interface and extend one of the repository interfaces provided by Spring Data.
2. Add custom query methods to the created repository interface (if we need them that is).
3. Inject the repository interface to another component and use the implementation that is provided automatically by Spring.

For more information visit https://www.petrikainulainen.net/spring-data-jpa-tutorial/

JpaRepository

JpaRepository is a JPA specific extension of Repository. It contains the full API of CrudRepository and PagingAndSortingRepository. So it contains API for basic CRUD operations and also API for pagination and sorting.

CrudRepository Vs JPARepository

Sr. No.	Key	JPARepository	CrudRepository
1	Hierarchy	JPA extend crudRepository and PagingAndSorting repository	Crud Repository is the base interface and it acts as a marker interface.
2	Batch support	JPA also provides some extra methods related to JPA such as delete records in batch and flushing data directly to a database.	It provides only CRUD functions like findOne, saves, etc.
3	Pagination support	JPA repository also extends the PagingAndSorting repository. It provides all the methods for which are useful for implementing pagination.	Crud Repository doesn't provide methods for implementing pagination and sorting.
4	Use Case	JpaRepository ties your repositories to the JPA persistence technology so it should be avoided.	We should use CrudRepository or PagingAndSortingRepository depending on whether

			you need sorting and paging or not.

An Introduction to OAuth 2

Introduction

OAuth 2 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, GitHub, and DigitalOcean. It works by delegating user authentication to the service that hosts the user account, and authorizing third-party applications to access the user account. OAuth 2 provides authorization flows for web and desktop applications, and mobile devices.

This informational guide is geared towards application developers, and provides an overview of OAuth 2 roles, authorization grant types, use cases, and flows.

Let’s get started with OAuth Roles!

OAuth Roles

OAuth defines four roles:

Resource Owner

Client

Resource Server

Authorization Server

We will detail each role in the following subsections.

Resource Owner: User

The resource owner is the user who authorizes an application to access their account. The application’s access to the user’s account is limited to the “scope” of the authorization granted (e.g. read or write access).

Resource / Authorization Server: API

The resource server hosts the protected user accounts, and the authorization server verifies the identity of the user then issues access tokens to the application.

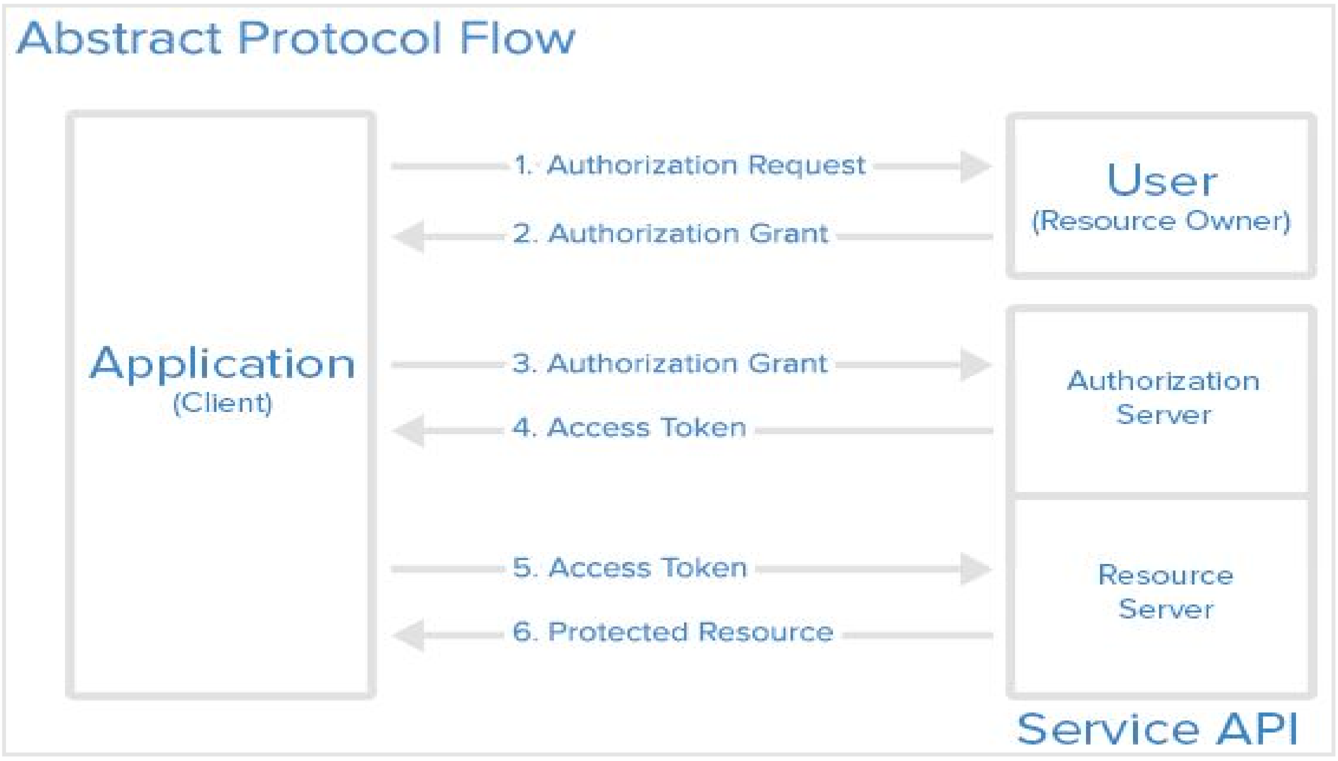
From an application developer's point of view, a service's API fulfills both the resource and authorization server roles. We will refer to both of these roles combined, as the Service or API role.

Client: Application

The client is the application that wants to access the user's account. Before it may do so, it must be authorized by the user, and the authorization must be validated by the API.

Abstract Protocol Flow

Now that you have an idea of what the OAuth roles are, let's look at a diagram of how they generally interact with each other:



Here is a more detailed explanation of the steps in the diagram:

1. The application requests authorization to access service resources from the user
2. If the user authorized the request, the application receives an authorization grant

3. The application requests an access token from the authorization server (API) by presenting authentication of its own identity, and the authorization grant
4. If the application identity is authenticated and the authorization grant is valid, the authorization server (API) issues an access token to the application. Authorization is complete.
5. The application requests the resource from the resource server (API) and presents the access token for authentication
6. If the access token is valid, the resource server (API) serves the resource to the application.

The actual flow of this process will differ depending on the authorization grant type in use, but this is the general idea. We will explore different grant types in a later section.

Application Registration

Before using OAuth with your application, you must register your application with the service. This is done through a registration form in the "developer" or "API" portion of the service's website, where you will provide the following information (and probably details about your application):

Application Name

Application Website

Redirect URI or Callback URL

The redirect URI is where the service will redirect the user after they authorize (or deny) your application, and therefore the part of your application that will handle authorization codes or access tokens.

Client ID and Client Secret

Once your application is registered, the service will issue “client credentials” in the form of a client identifier and a client secret. The Client ID is a publicly exposed string that is used by the service API to identify the application, and is also used to build authorization URLs that are presented to users. The Client Secret is used to authenticate the identity of the application to the service API when the application requests to access a user’s account, and must be kept private between the application and the API.

Authorization Grant

In the Abstract Protocol Flow above, the first four steps cover obtaining an authorization grant and access token. The authorization grant type depends on the method used by the application to request authorization, and the grant types supported by the API. OAuth 2 defines four grant types, each of which is useful in different cases:

Authorization Code: used with server-side Applications

Implicit: used with Mobile Apps or Web Applications (applications that run on the user’s device)

Resource Owner Password Credentials: used with trusted Applications, such as those owned by the service itself

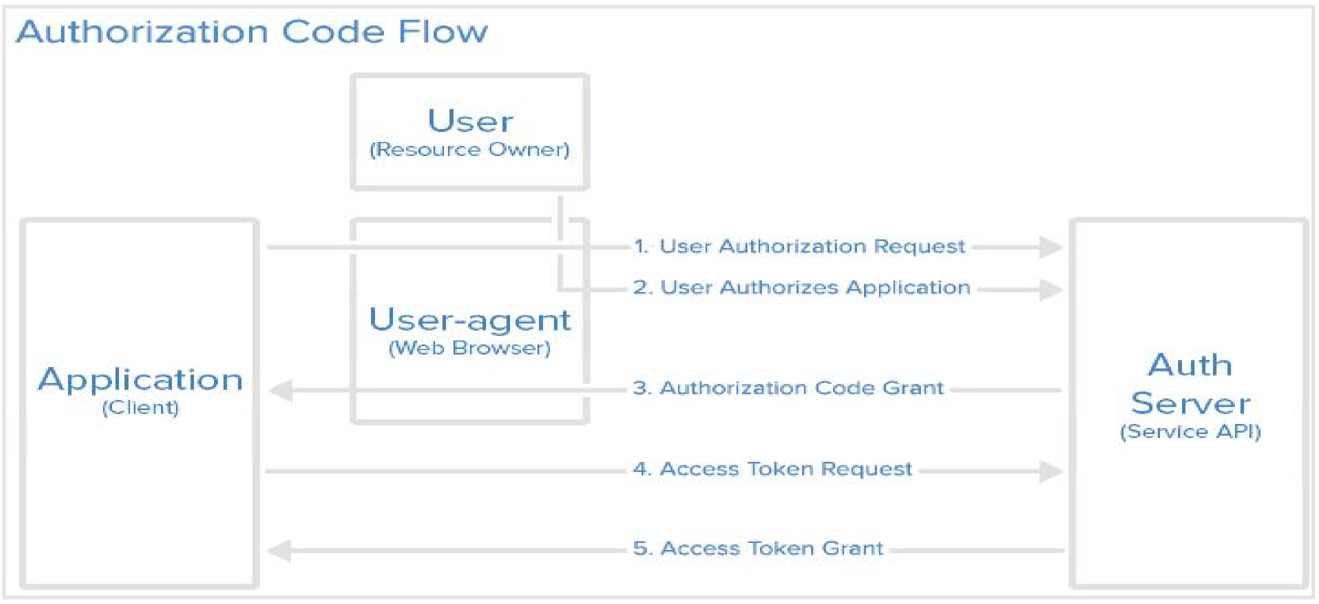
Client Credentials: used with Applications API access

Now we will describe grant types in more detail, their use cases and flows, in the following sections.

Grant Type: Authorization Code

The authorization code grant type is the most commonly used because it is optimized for server-side applications, where source code is not publicly exposed, and Client Secret confidentiality can be maintained. This is a redirection-based flow, which means that the application must be capable of interacting with the user-agent (i.e. the user's web browser) and receiving API authorization codes that are routed through the user-agent.

Now we will describe the authorization code flow:



Step 1: Authorization Code Link

First, the user is given an authorization code link that looks like the following:

https://cloud.digitalocean.com/v1/oauth/authorize?response_type=code&client_id=CLIENT_ID&redirect_uri=CALLBACK_URL&scope=read

Here is an explanation of the link components:

- **https://cloud.digitalocean.com/v1/oauth/authorize**: the API authorization endpoint
- **client_id=client_id**: the application's *client ID* (how the API identifies the application)
- **redirect_uri=CALLBACK_URL**: where the service redirects the user-agent after an authorization code is granted
- **response_type=code**: specifies that your application is requesting an authorization code grant

- **scope=read**: specifies the level of access that the application is requesting.

Step 2: User Authorizes Application

When the user clicks the link, they must first log in to the service, to authenticate their identity (unless they are already logged in). Then they will be prompted by the service to authorize or deny the application access to their account. Here is an example authorize application prompt:

This particular screenshot is of DigitalOcean's authorization screen, and we can see that "The Droplet Book App" is requesting authorization for "read" access to the account of "manicas@digitalocean.com".

Step 3: Application Receives Authorization Code

If the user clicks "Authorize Application", the service redirects the user-agent to the application redirect URI, which was specified during the client registration, along with an authorization code. The redirect would look something like this (assuming the application is "dropletbook.com"):

https://dropletbook.com/callback?code=AUTHORIZATION_CODE

Step 4: Application Requests Access Token

The application requests an access token from the API, by passing the authorization code along with authentication details, including the client secret, to the API token endpoint. Here is an example POST request to DigitalOcean's token endpoint:

https://cloud.digitalocean.com/v1/oauth/token?client_id=CLIENT_ID&client_secret=CLIENT_SECRET&grant_type=authorization_code&code=AUTHORIZATION_CODE&redirect_uri=CALLBACK_URL

Step 5: Application Receives Access Token

If the authorization is valid, the API will send a response containing the access token (and optionally, a refresh token) to the application. The entire response will look something like this:

```
{"access_token":"ACCESS_TOKEN","token_type":"bearer","expires_in":2592000,"refresh_token":"REFRESH_TOKEN","scope":"read","uid":100101,"info":{"name":"Mark E. Mark","email":"mark@thefunkybunch.com"}}
```

Now the application is authorized! It may use the token to access the user's account via the service API, limited to the scope of access, until the token expires or is revoked. If a refresh token was issued, it may be used to request new access tokens if the original token has expired.

What is JSON Web Token?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

Authorization: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

Information Exchange: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

What is the JSON Web Token structure?

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

Header

Payload

Signature

Therefore, a JWT typically looks like the following.

xxxxx.yyyyy.zzzzz

Let's break down the different parts.

Header

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

```
{

  "alg": "HS256",

  "typ": "JWT"

}
```

Then, this JSON is Base64Url encoded to form the first part of the JWT.

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

Registered claims: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.

Notice that the claim names are only three characters long as JWT is meant to be compact.

Public claims: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

Private claims: These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

An example payload could be:

```
{

  "sub": "1234567890",

  "name": "John Doe",

  "admin": true

}
```

The payload is then Base64Url encoded to form the second part of the JSON Web Token.

Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)
```

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

Putting all together

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

If you want to play with JWT and put these concepts into practice, you can use jwt.io Debugger to decode, verify, and generate JWTs.

How do JSON Web Tokens work?

In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required.

You also should not store sensitive session data in browser storage due to lack of security.

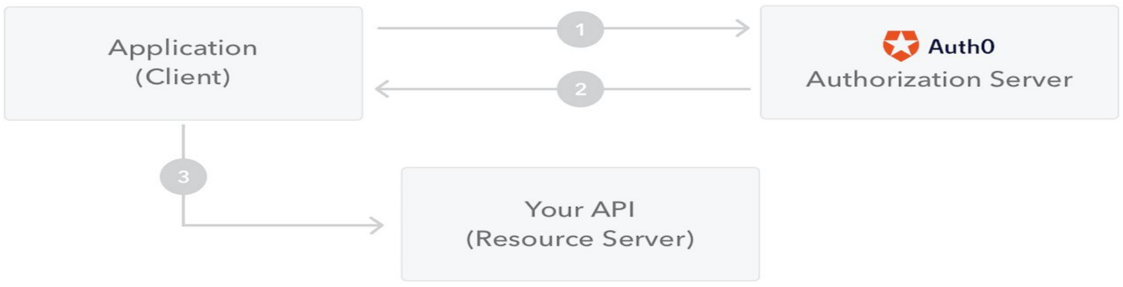
Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

Authorization: Bearer <token>

This can be, in certain cases, a stateless authorization mechanism. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. If the JWT contains the necessary data, the need to query the database for certain operations may be reduced, though this may not always be the case.

If the token is sent in the Authorization header, Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies.

The following diagram shows how a JWT is obtained and used to access APIs or resources:



1. The application or client requests authorization to the authorization server. This is performed through one of the different authorization flows. For example, a typical OpenID Connect compliant web application will go through the /oauth/authorize endpoint using the authorization code flow.
2. When the authorization is granted, the authorization server returns an access token to the application.
3. The application uses the access token to access a protected resource (like an API).

Do note that with signed tokens, all the information contained within the token is exposed to users or other parties, even though they are unable to change it. This means you should not put secret information within the token.

Spring Boot Security Oauth2 Jwt Auth Example

Here, we will be creating a sample spring security OAUTH2 application using JwtTokenStore.Using JwtTokenStore as a token provider allows us to customize the token generated with TokenEnhancer to add additional claims.

https://www.devglan.com/spring-security/spring-boot-oauth2-jwt-example

https://blog.marcosbarbero.com/centralized-authorization-jwt-spring-boot2/

https://www.youtube.com/watch?v=X80nJ5T7YpE

Difference Between Web server and Application server

A server is a central repository where information and computer programs are held and accessed by the programmer within the network. Web server and Application server are kinds of the server which are employed to deliver sites and therefore the latter deals with application operations performed between users and back-end business applications of the organization. "Web server" can refer to hardware or software, or both of them working together.

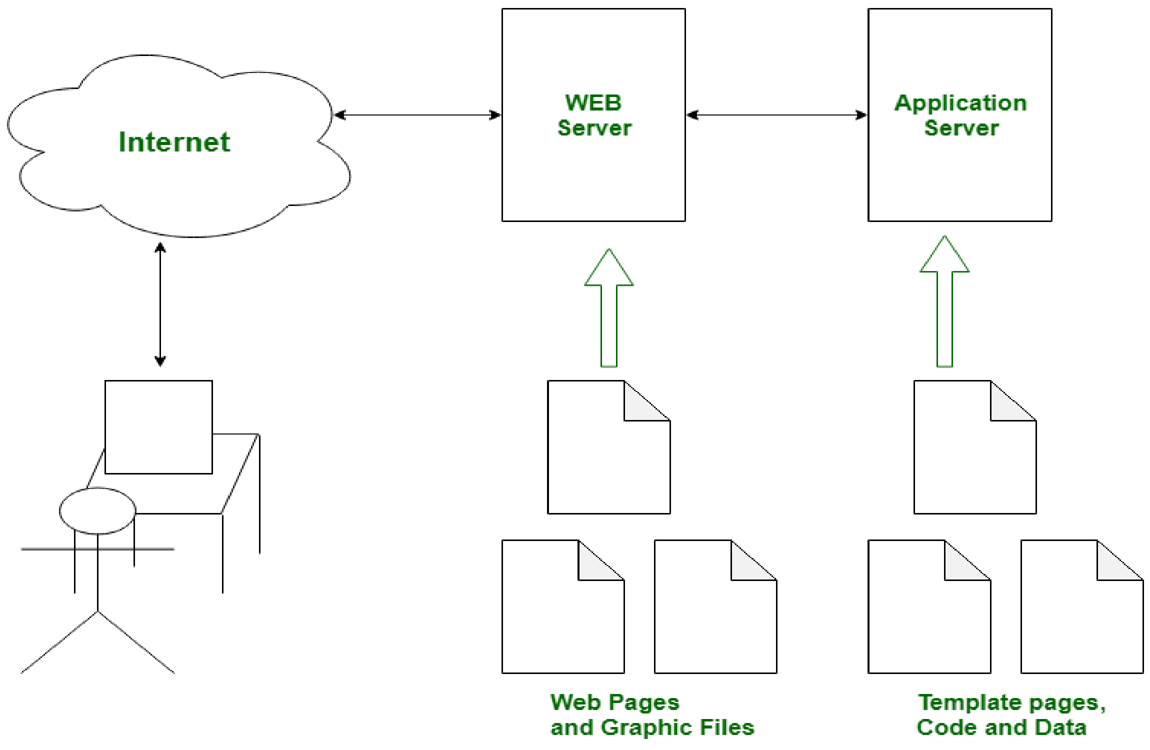
On the hardware side, a web server is a computer that stores web server software and a website's component files (e.g. HTML documents, images, CSS stylesheets, and JavaScript files). It is connected to the Internet and supports physical data interchange with other devices connected to the web.

On the software side, a web server includes several parts that control how web users access hosted files, at minimum an HTTP server. An HTTP server is a piece of software that understands URLs (web addresses) and HTTP (the protocol your browser uses to view webpages). It can be accessed through the domain names (like mozilla.org) of websites it stores, and delivers their content to the end-user's device.

At the most basic level, whenever a browser needs a file which is hosted on a web server, the browser requests the file via HTTP. When the request reaches the correct web server (hardware), the HTTP server (software) accepts the request, finds the requested document (if it doesn't then a 404 response is returned), and sends it back to the browser, also through HTTP.

Basic representation of a client/server connection through HTTP. To publish a website, you need either a static or a dynamic web server.A static web server, or stack, consists of a computer (hardware) with an HTTP server (software). We call it "static" because the server sends its hosted files as it is to your browser. A dynamic web server consists of a static web server plus extra software, most commonly an application server and a database. We call it "dynamic" because the application server updates the hosted files before sending them to your browser via the HTTP server.

Application server: It encompasses Web container as well as EJB container. Application servers organize the run atmosphere for enterprises applications. Application server may be a reasonable server that means how to put an operating system, hosting the applications and services for users, IT services and organizations. In this, user interface similarly as protocol and RPC/RMI protocols are used. Examples of Application Servers are Weblogic, JBoss, Websphere



1. Web server encompasses web containers only whereas Application server encompasses Web container as well as EJB container.
2. Web server is useful or fitted for static content whereas application servers are fitted for dynamic content.

3. Web servers consume or utilize less resources whereas application servers utilize more resources.
4. Web servers arrange the run environment for web applications whereas application servers arrange the run environment for enterprises applications.
5. In web servers, multithreading is not supported while in application server, multithreading is supported.
6. In web server, HTML and HTTP protocols are used whereas in this, GUI as well as HTTP and RPC/RMI protocols are used

Understanding Gradle: the Build Lifecycle

What do you understand by Gradle?

Gradle is an open-source built system whose main job is to take the project's resources, source code, and other things related to it into an APK file. Gradle uses a stable programming language known as Groovy rather than using HTML configuration file. Gradle build can easily handle anything related to the code of the program as Gradle's build scripts are coded in a superior language. They are a regular program that uses Groovy instead of Java to write scripts. It also has directed acyclic graph that determines the task's order.

Explain Groovy?

Gradle uses a programming language that is written in a script form, and the name of that script is Groovy. The features of this language are:
It interoperates with Java easily as Groovy operates on JVM (Java Virtual Machine).
To write a build script, you don't have to learn Groovy.
It is simple to write and read a Groovy due to its smaller codes than Java.
It is a dynamic and flexible language which works somewhat similar to Java. It is also compatible with the byte code of JVM.

What is Gradle Framework?

It is a type of automated build system which is open source and creates builds on the concepts of Apache Ant and Maven. It uses a domain specific language (DSL) which is based on Groovy to declare about the configuration of the project. It doesn't use the XML form that Apache Maven uses for this declaration.

How do you create build scan in Gradle?

With the help of build scan one can develop key insights into what happened during a build, and why it happened. It is more like troubleshooting a failed build with the help of a log file. In order to create a build scan, one needs to run the command "gradle build --scan", and can check the output from scan.gradle.com.

What do you mean by Gradle Wrapper?

The Gradle wrapper is the most suitable way to initiate a Gradle build. A Gradle wrapper is a Window's batch script which has a shell script for the OS (operating system). Once you start the Gradle build via the wrapper, you will see an auto download which runs the build.

What is Dependency Configuration?

A configuration dependency is a set of dependency that includes external dependency that you require to install and ensure that the downloading is happening via the web. Some key features of dependency configuration are:

Compilation: The initial project that you will start and work on should be well-compiled. Also, ensure that you maintain it in good condition.
Runtime: Runtime is the preferred time that you need to complete the work dependency in a collection form.
Test Compile: It requires a complete collection for making the project run.
Test runtime: It's the final process which requires the checking to complete for running a test which is the default runtime mode.

Gradle Daemon

It is a background built server which you can use to build quickly. Gradle Daemon's best feature is the fact that its speed becomes faster as you use it more. It is a Java process that acts as a server/client that communicates with a local TCP. It also executes and finds build actions.

Explain in detail the reason Gradle is the first choice amongst developers?

Traditionally, the building was all about packaging and compiling the source code. But now the work of a builder is more than that. They have to perform test run, merge code resources with multiple sources, provide documentation, manage dependencies and publish applications.

Thus, the builder became software where you can release apps and change tests. And with Gradle, you can perform all these functions in a single place.

For instance, Gradle is the build tool for Android Studio. It makes Gradle more efficient than other tools. In fact, the whole process of building an Android app is now with Gradle tool. It can work with multiple platforms and has a powerful and compact build language Groovy.

Gradle VS Maven:

What is it: Gradle an open-source build tool is a combination of Maven and Ant. Maven uses Java as its basis to build automation projects.
Language: It works with a domain-specific language with Groovy as its basis. While Apache Maven uses XML for configuring its projects

Approach: Gradle uses the graph task dependency approach. In this, the tasks are the things that perform the work. Maven uses the approach of a linear and fixed model of phases.
Performance: Performance wise both Gradle and Maven allow multi-projects build. But In Gradle, you can execute only incremental projects. The reason is that it checks for updated tasks and if they are present, there is no execution and the span of build time becomes shorter.

What About Publishing Artifacts To Maven Central?

This allows for maximum convenience for the majority of Spring users, given that most users have Maven-based builds and Maven resolves artifacts by default from Maven Central.

The preferred way of releasing artifacts to Maven Central is via Sonatype's Nexus server at oss.sonatype.org (OSO). This is explained in detail in Sonatype's OSS usage guide.

The Spring Artifactory repository has been customized with a "nexus-push" plugin that allows for automatic propagation of builds from Artifactory to the Nexus server at OSO for publication into Maven Central.

All Spring projects -- that is, all projects having groupid org.springframework -- can publish to OSO under the shared 'springsource' account. This has already been set up in the nexus-push plugin, so there's no additional setup necessary at OSO, even for new projects.

The Artifactory Bamboo plugin supports use of the nexus-push plugin through it's UI. Step 3 of the the FAQ entry above on publishing releases described the process for promoting a build out of staging. If the build is a GA release, simply choose the 'Push to Nexus' option, and select 'libs-release-local' as the target repository.

What Is Gradle Multi-Project Build?

Multi-project builds helps with modularization. It allows a person to concentrate on one area of work in a larger project, while Gradle takes care of dependencies from other parts of the project A multi-project build in Gradle consists of one root project, and one or more subprojects that may also have subprojects. While each subproject could configure itself in complete isolation of the other subprojects, it is common that subprojects share common traits. It is then usually preferable to share configurations among projects, so the same configuration affects several subprojects.

lets agree on some capital definitions in the Gradle universe.

Project: This is one of the most important concept for Gradle. In fact, a project is a representation of what needs to be built and leads to an artifact at the end of the build. For example, on Android the :app module is a Gradle project. If you have another module in your Android project (don't get confused here), that module also is a project for Gradle. Projects are registered in the settings.gradle file Most of the time, a projet has a build.gradle file

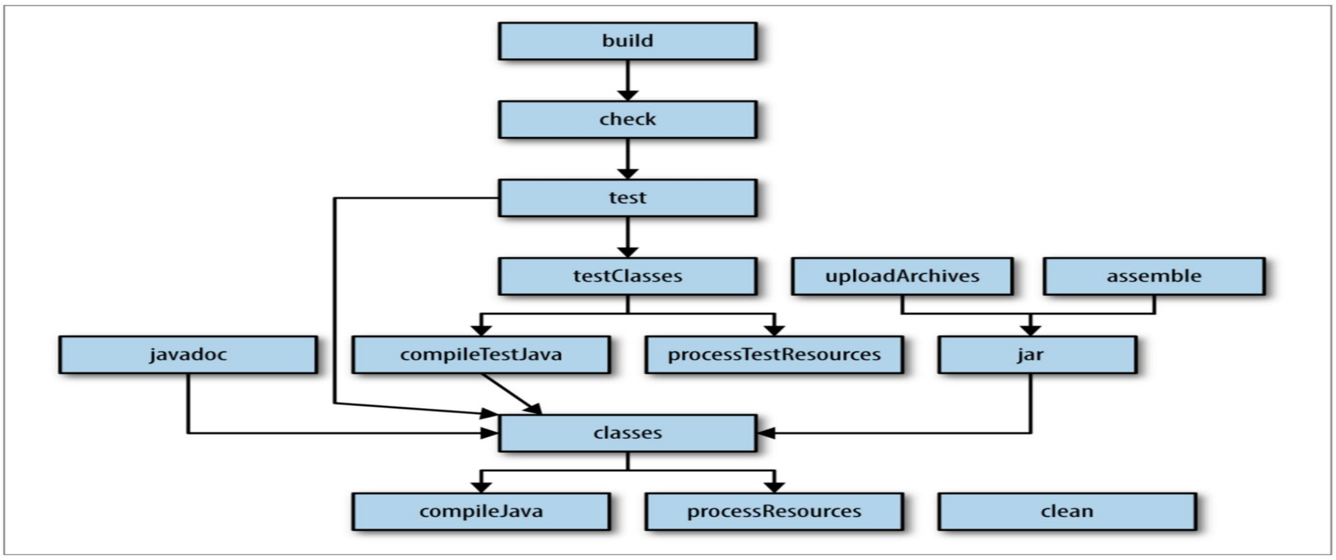
Task: A task, as the name suggests, is a representation of actions (default or custom) that need to be executed during the build process. For example, the compilation of Java code is started by a task. Tasks are defined in the project build script and can have dependencies with each other.

Now that we know these capital terms let's dive into the main subject

Build Phases: Every Gradle build goes through 3 different lifecycle phases following the same order every time.

The initialization phase: In this phase, Gradle tries to identify all the projects involved in the build process. It is very important for Gradle to know whether it's a Single-project build or a Multi-project build. In a Multi-project build there are several projects to evaluate. Hence, several build scripts. Gradle looks at the settings.gradle file in order to identify the different projects. At the end of the initialization phase, Gradle creates an instance of org.gradle.api.Project corresponding to each of these projects.

The configuration phase: During this phase, Gradle executes the build script of each project identified in the previous phase. Actually, it is very important to know that just because we say "Gradle executes the build scripts" does not mean that the Tasks in those build scripts are executed too. Instead, after evaluating those scripts as simple Groovy scripts and identify the tasks in it, Gradle builds a Directed Acyclic Graph (DAG) of task objects. A DAG is a mathematical algorithm for representing a graph that contains no cycles. The "directed" term means each dependency arrow goes in one direction. "Acyclic" means that there are no loops in the graph.



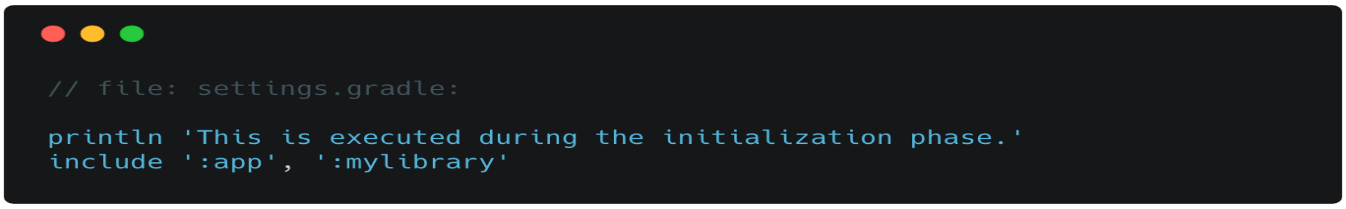
Directed acyclic graph for the Java plug-in tasks from the book Gradle Recipes for Android. Also one thing to mention in the Configuration phase is that Gradle introduced a feature called configuration on demand that gives it the ability to configure only the relevant and necessary projects during the build process. This is very useful in large Multi-project builds because it could considerably decrease build time.

The execution phase; This is the last phase. During this phase, Gradle identifies the tasks that need to be executed based on the DAG of task objects created in the previous phase, and executes them according to their dependency order. All the build work and activities are actually done in this phase. For example: compiling source code and generating .class files, copying files, cleaning build directory, uploading archives, archiving files etc.

This is cool but what does it look like in real life ?

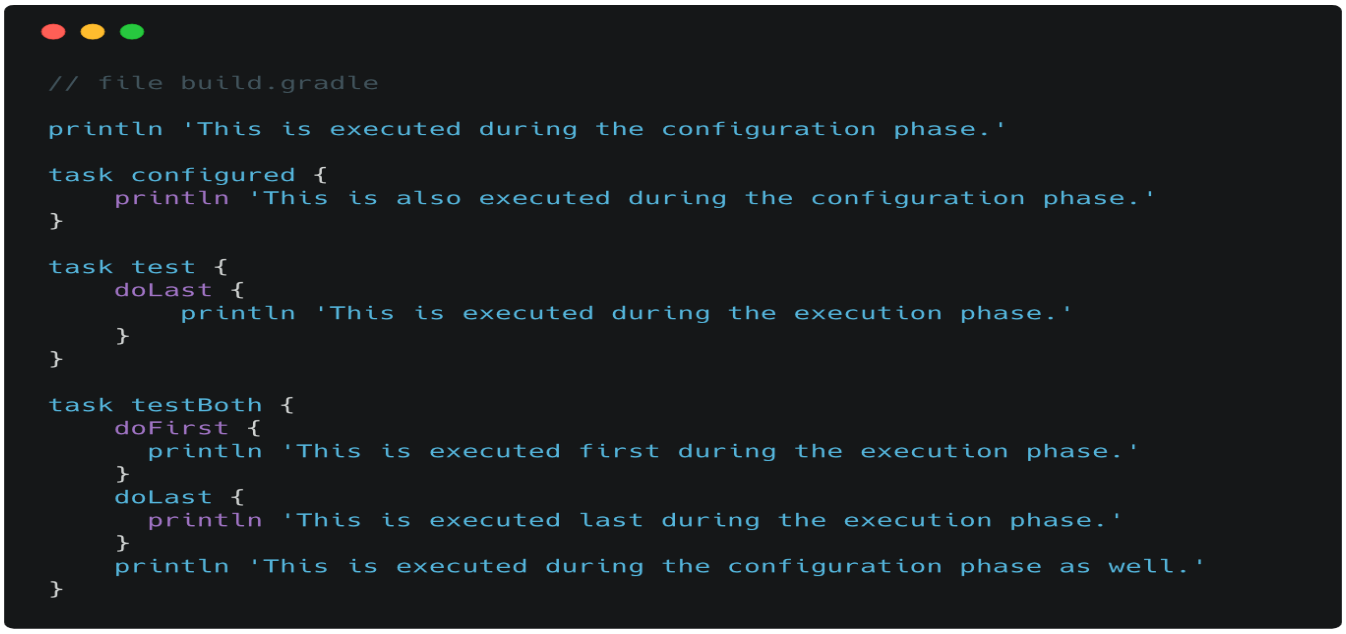
Well, in "real life" the phases could easily be identified in your gradle files. For example if you put a code in the settings.gradle file, it is evaluated in the initialization phase. The code in your build script files that are not related to actions of your tasks are evaluated in the configuration phase. And finally the code in the actual actions of your tasks like the doLast closures of your tasks are evaluated in the execution phase.

Let's take a look:



```
// file: settings.gradle:

println 'This is executed during the initialization phase.'
include ':app', ':mylibrary'
```

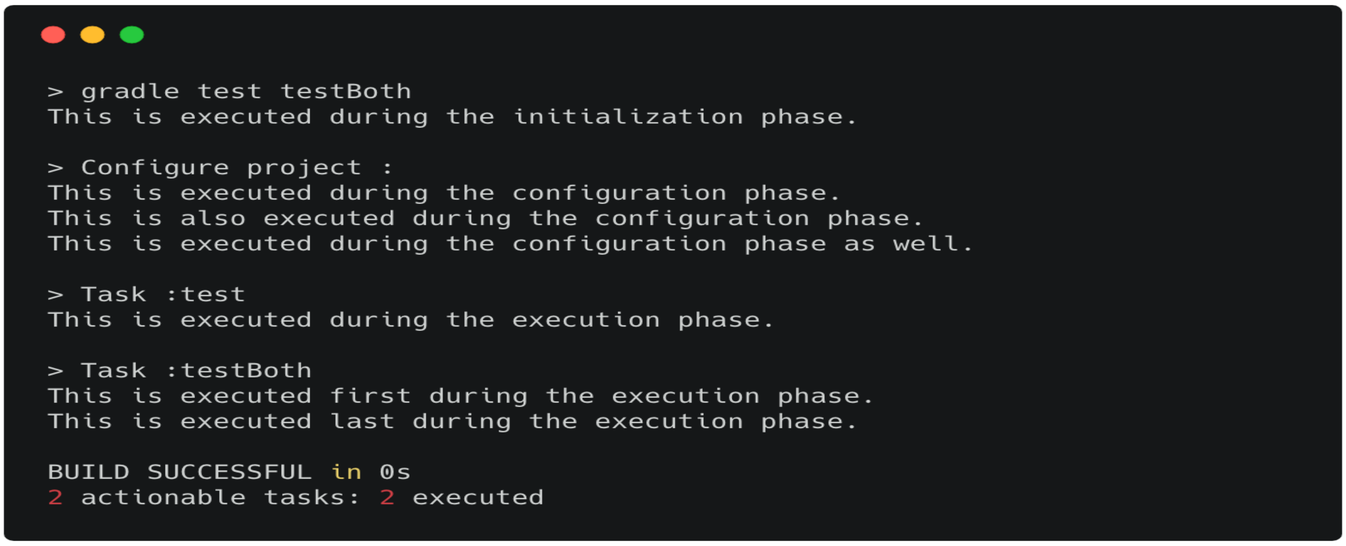
settings.gradle



```
// file build.gradle

println 'This is executed during the configuration phase.'

task configured {
    println 'This is also executed during the configuration phase.'
}

task test {
    doLast {
        println 'This is executed during the execution phase.'
    }
}

task testBoth {
    doFirst {
      println 'This is executed first during the execution phase.'
    }
    doLast {
      println 'This is executed last during the execution phase.'
    }
    println 'This is executed during the configuration phase as well.'
}
```

build.gradle

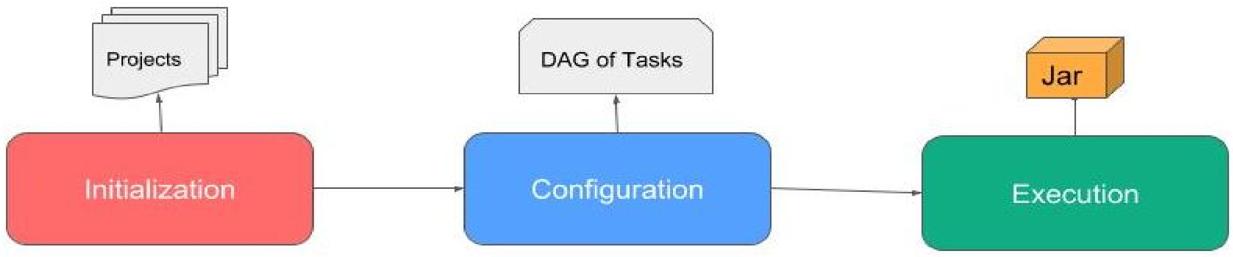
And the result of the build after entering the command gradle test testBoth



```
> gradle test testBoth
This is executed during the initialization phase.

> Configure project :
This is executed during the configuration phase.
This is also executed during the configuration phase.
This is executed during the configuration phase as well.

> Task :test
This is executed during the execution phase.

> Task :testBoth
This is executed first during the execution phase.
This is executed last during the execution phase.

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Output of gradle test testBoth

Let's wrap it up. This single image can give you an idea of what we've been talking about throughout this article. We represent here in a very simple and stupid way the build process that takes us to the generation of a jar but it could be anything else like aar, apk, war etc.



The 3 build lifecycle phases with their output

Thank you for reading, i hope i helped you learn something new in one way or another. Perhaps i forgot something, perhaps i was wrong somewhere, if so, just leave a comment so we can discuss about it.

For more reading:

https://www.oreilly.com/library/view/gradle-beyond-the/9781449373801/ch03.html

SPRING FRAMEWORK INTERVIEW QUESTION

1. What is a Spring framework?
Spring is a development framework for Java programming. It is an open source development framework for Enterprise Java. The core features of Spring Framework can be used in developing a Java Enterprise application. It has many extensions and jars for developing web applications on top of the Java EE platform. With Spring we can develop large-scale complex Java applications very easily. It is also based on good design patterns like Dependency Injection, Aspect oriented programming for developing extensible feature rich software.

2. What are the benefits of Spring framework in software development?

Many benefits of Spring framework are: Lightweight Framework: Basic Spring framework is very small in size. It is easy to use and does not add a lot of overhead on software. It just has 2 MB in the basic version. Container: Spring framework provides the basic container that creates and manages the life cycle of application objects like Plain old Java objects (POJO). It also stores the configuration files of
application objects to be created. Dependency Injection (DI): Spring provided loose coupling is application by Dependency Injection. It uses the Inversion of Control technique by which objects specify their dependencies to the Spring container instead of creating new objects themselves. Aspect Oriented Programming (AOP): Spring framework promotes and provides support for Aspect oriented programming in Java. This helps in separating application business logic from system services that are common across all the business logic. E.g. Logging can be a cross cutting concern in an Application. Transaction Management: Spring provides a framework for transaction management. So a developer does not have to implement it from scratch. Spring Transaction Management is so powerful that we can scale it from one local transaction to global transactions in a cluster. MVC Framework: For Web applications, Spring provides MVC framework. This framework is based on MVC design pattern and has better features compared to other web frameworks. Exception Handling: Spring also gives support for a common API to handle exceptions in various technologies like- Hibernate, JDBC
etc.

3. What are the modules in the Core Container of Spring framework?
Spring framework has a Core Container. Modules in Core Container are:

Core module
Bean module
Context module
Spring Expression Language module

4. What are the modules in the Data Access/Integration layer of Spring framework?
Modules in Data Access/Integration Layer of Spring framework are:
JDBC module: An abstraction layer to remove tedious JDBC coding.
ORM module Integration layers for Object Relational Mapping
OXM module: An abstraction layer to support Object XML mapping.
Java Messaging Service (JMS) module: Module for producing and consuming messages.
Transactions module: Transaction Management for POJO classes

5. What are the modules in the Web layer of Spring framework?
Modules in Web Layer of Spring framework are:
Web module: This provides basic web-oriented integration features.
Servlet module: Support for Servlet Listeners.
WebSocket module: Support for Web Socket style messaging.
Portlet module: MVC implementation for Portlet environment.

6. What is the main use of the Core Container module in the Spring framework?
As the name suggests, Spring Core Container is the core of Spring framework. It gives the basic functionality of the Spring. All the parts of Spring Framework are built on top of Core Container. Its main use is to provide Dependency Injection (DI) and Inversion of control (IOC) features.

7. What kind of testing can be done in the Spring Test Module?
Spring Test Module provides support for Unit testing as well as Integration testing of Spring components. It allows using JUnit or TestNG testing frameworks. It also gives the ability to mock objects to use the test code.

8. What is the use of BeanFactory in the Spring framework?
BeanFactory is the main class that helps in implementing Inversion of Control pattern in Spring. It is based on the factory design pattern. It separates the configuration and dependencies of an application from the rest of application code. Implementations of BeanFactory like XmlBeanFactory class are used by applications built with Spring.

9. Which is the most popular implementation of BeanFactory in Spring?
XMLBeanFactory is the most popular implementation of BeanFactory in Spring.

10. What is XMLBeanFactory in Spring framework?
XMLBeanFactory is one of the most useful implementations of BeanFactory in Spring. This factory loads its beans based on the definitions mentioned in an XML file. Spring container reads bean configuration metadata from an XML file and creates a fully configured application with the help of XMLBeanFactory class.

11. What are the uses of the AOP module in the Spring framework?
AOP module is also known as Aspect Oriented Programming module. Its uses are: Development of aspects in a Spring based application Provides interoperability between Spring and other AOP frameworks Supports metadata programming to Spring.

12. What are the benefits of JDBC abstraction layer modules in Spring framework?
Spring provides a JDBC abstraction layer module. Main benefits of this module are: Helps in keeping the database code clean and simple. Prevents problems that result from a failure to close database resources. Provides a layer of useful exceptions on top of the error messages given by different database servers. Based on Spring's AOP module Provides transaction management services for objects in a Spring application.

13. How does Spring support Object Relational Mapping (ORM) integration?
Spring supports Object Relational Mapping (ORM) by providing an ORM Module. This module helps in integrating with popular ORM frameworks like Hibernate, JDO, and iBATIS SQL Maps etc. The Transaction Management module of Spring framework supports all of these ORM frameworks as well as JDBC.

14. How does the Web module work in the Spring framework?
Spring provides support for developing web applications by using Web modules. This module is built on an application context module that provides context for web-based applications. This module also supports web-oriented integration features like transparently handling multipart requests for uploading files, programmatically binding request parameters to business objects etc. This module also supports integration with popular web frameworks like Jakarta Struts, JSF, and Tapestry etc.

15. What are the main uses of the Spring MVC module?
Spring-webmvc module is also known as Web-servlet module. It is based on the Web Model View Controller pattern. Main uses of this module are: Integration of Spring with other MVC frameworks Supports IoC to provide clean separation of controller logic from business objects Provides clean separation between domain model code and web forms Allows developers to declaratively bind request parameters to business objects.

16. What is the purpose of Spring configuration file?
Spring applications can be configured by an XML file. This file contains information of classes and how these classes are configured and introduced to each other. Spring IoC container uses some kind of configuration metadata. This configuration metadata represents how an application developer tells the Spring container to instantiate, configure, and assemble the objects in your application. This configuration metadata is stored in the Spring configuration file. The other ways of specifying configuration metadata are Java based configuration and Annotation based configuration.

17. What is the purpose of the Spring IoC container?
The Spring IoC Container is responsible for: Creating the objects, Configuring the objects, Managing dependency between objects (with dependency injection (DI)) , Wiring the objects together Managing complete lifecycle of objects.

18. What is the main benefit of the Inversion of Control (IOC) principle?
Inversion of Control (IOC) principle is the base of Spring framework. It supports dependency injection in an application. With Dependency Injection, a programmer has to write minimal code. It also makes it easier to test an application. Most important benefit is that it leads to loose coupling within objects. With loose coupling it is easier to change the application with new requirements.

19. Do IOC containers support Eager Instantiation or Lazy loading of beans?
IOC Container in Spring supports both the approaches. Eager instantiation as well as lazy loading of beans.

20. What are the benefits of ApplicationContext in Spring?
ApplicationContext in Spring provides following benefits: Bean factory methods: These are used to access application components , Load File Resources: It helps in loading file resources in a generic fashion, Publish Events: It enables publishing events to registered

listeners Internationalization Support: Ability to resolve messages to support internationalization, Parent Context: Ability to inherit from a parent context.

21. How will you implement ApplicationContext in Spring framework?
ApplicationContext in Spring can be implemented in one of the following three ways:
FileSystemXmlApplicationContext: If we want to load the definitions of beans from an XML file then FileSystemXmlApplicationContext is used. The full path of the XML bean configuration file is provided to the constructor.
ClassPathXmlApplicationContext: To load the definitions of beans from an XML file in the CLASSPATH, we use ClassPathXmlApplicationContext. It is used for application context embedded in jars.
WebXmlApplicationContext: To provide configuration for a web application WebXmlApplicationContext is used. While the application is running, it is read only. But it can be reloaded if the underlying application supports it.

22. Explain the difference between ApplicationContext and BeanFactory in Spring?
Main differences between ApplicationContext and BeanFactory are: Automatic BeanPostProcessor registration: BeanFactory does not support BeanPostProcessor registration. Whereas ApplicationContext supports this. Automatic BeanFactoryPostProcessor registration: BeanFactory also does not allow Automatic BeanFactoryPostProcessor registration. Whereas ApplicationContext allows this. MessageSource access: BeanFactory is not convenient for MessageSource access. ApplicationContext is quite convenient for MessageSource access. ApplicationEvent: We cannot publish ApplicationEvent with BeanFactory. But ApplicationContext provides the ability to publish ApplicationEvent.

23. Between ApplicationContext and BeanFactory which one is preferable to use in Spring?
Spring documentation recommends using ApplicationContext in almost all the cases. ApplicationContext has all the functionality of BeanFactory.

24. What are the main components of a typical Spring based application?
In a Spring based application, main components are: Spring configuration XML file: This is used to configure Spring application
API Interfaces: Definition of API interfaces for functions provided by application
Implementation: Application code with implementation of APIs
Aspects: Spring Aspects implemented by application
Client: Application at client side that is used for accessing functions

25. What are the different roles in Dependency Injection (DI)?
There are four roles in Dependency Injection: Service object(s) to be used Client object that depends on the service Interface that defines how client uses services
Injector responsible for constructing services and injecting them into clients.
26. Spring framework provides what kinds of Dependency Injection mechanism?

Spring framework provides two types of Dependency Injection mechanism:
Constructor-based Dependency Injection: Spring container can invoke a class constructor with a number of arguments. This represents a dependency on another class.
Setter-based Dependency Injection: Spring container can call a setter method on a bean after creating it with a no-argument constructor or no-argument static factory method to instantiate another bean.

27. In the Spring framework, which Dependency Injection is better? Constructor-based DI or Setter-based DI?
Spring framework provides support for both Constructor-based and Setter-based Dependency Injection. There are different scenarios in which these options can be used. It is recommended to use Constructor-based DI for mandatory dependencies. Whereas Setter-based DI is used for optional dependencies.

29. What are the advantages of Dependency Injection (DI)?
Dependency Injection (DI) pattern has following advantages: Dependency Injection reduces coupling between a class and its dependencies. With Dependency Injection (DI), we can do concurrent or independent software development. Two teams can work parallel on classes that will be used by each other. In Dependency Injection (DI), the client can be configured in multiple ways. It needs to just work with the given interface. Rest of the implementation can be changed and configured for different features. Dependency injection is also used to export a system's configuration details into configuration files. So we can configure the same application run in different environments based on configuration. E.g. Run in the Test environment, UAT environment, and Production environment. Dependency Injection (DI) applications provide more ease and flexibility of testing. These can be tested in isolation in Unit Tests. Dependency injection (DI) isolates clients from the impact of design and implementation changes. Therefore, it promotes reusability, testability and maintainability.

30. What is a Spring Bean?
A Spring Bean is a plain old Java object (POJO) that is created and managed by a Spring container. There can be more than one bean in a Spring application. But all these Beans are instantiated and assembled by a Spring container. Developer provides configuration metadata to Spring container for creating and managing the lifecycle of Spring Bean. In general a Spring Bean is singleton. Evert bean has an attribute named "singleton". If its value is true then the bean is a singleton. If its value is false then bean is a prototype bean. By default the value of this attribute is true. Therefore, by default all the beans in the spring framework are singleton in nature.

31. What does the definition of a Spring Bean contain?
A Spring Bean definition contains configuration metadata for beans. This configuration metadata is used by Spring container to:
Create the bean
Manage its lifecycle

Resolve its dependencies

32. What are the different ways to provide configuration metadata to a Spring Container?
Spring supports three ways to provide configuration metadata to Spring Container: **XML based configuration**: We can specify configuration data in an XML file. **Annotation-based configuration**: We can use Annotations to specify configuration. This was introduced in Spring 2.5.
Java-based configuration: This is introduced from Spring 3.0. We can embed annotations like @Bean, @Import, @Configuration in Java code to specify configuration metadata.

33. What are the different scopes of a Bean supported by Spring?
Spring framework supports seven types of scopes for a Bean. Out of these only five scopes are available for a web-aware ApplicationContext application:
singleton: This is the default scope of a bean. Under this scope, there is a single object instance of bean per Spring IoC container.
prototype: Under this scope a single bean definition can have multiple object instances.
request: In this scope, a single bean definition remains tied to the lifecycle of a single HTTP request. Each HTTP request will have its own instance of a bean for a single bean definition. It is only valid in the context of a web-aware Spring ApplicationContext.
session: Under this scope, a single bean definition is tied to the lifecycle of an HTTP Session. Each HTTP Session will have one instance of bean. It is also valid in the context of a web-aware Spring ApplicationContext.
globalSession: This scope, ties a single bean definition to the lifecycle of a global HTTP Session. It is generally valid in a Portlet context. It is also valid in the context of a web-aware Spring ApplicationContext.
application: This scope limits a single bean definition to the lifecycle of a ServletContext. It is also valid in the context of a web-aware Spring ApplicationContext.
websocket: In this scope, a single bean definition is tied to the lifecycle of a WebSocket. It is also valid in the context of a webaware Spring ApplicationContext.

34. Is it safe to assume that a Singleton bean is thread safe in Spring Framework?
No, Spring framework does not guarantee anything related to multithreaded behavior of a singleton bean. Developer is responsible for dealing with concurrency issues and maintaining thread safety of a singleton bean.

35. What are the design-patterns used in the Spring framework?
Spring framework uses many Design patterns. Some of these
patterns are:
Singleton – By default beans defined in spring config files are singleton. These are based on the Singleton pattern.
Template – This pattern is used in many classes like- JdbcTemplate, RestTemplate, JmsTemplate, JpaTemplate etc.

Dependency Injection – This pattern is the core behind the design of BeanFactory and ApplicationContext.
Proxy – Aspect Oriented Programming (AOP) heavily uses proxy design patterns.
Front Controller – DispatcherServlet in Spring is based on Front Controller pattern to ensure that incoming requests are dispatched to other controllers.
Factory pattern – To create an instance of an object, BeanFactory is used. This is based on Factory pattern.
View Helper – Spring has multiple options for separating core code from presentation in views. Like- Custom JSP tags, Velocity macros etc.

36. What is the lifecycle of a Bean in Spring framework?
A Bean in Spring framework goes through the following phases in its lifecycle.Initialization and creation: Spring container gets the definition ofBean from XML file and instantiates the Bean. It populates all the properties of Bean as mentioned in the bean definition. Setting the Behavior of Bean: In case a Bean implements BeanNameAware interface, Spring uses setBeanName() method to pass the bean's id. In case a Bean implements BeanFactoryAware interface, Spring uses setBeanFactory() to pass the BeanFactory to Bean. Post Processing: Spring container uses postProcesserBeforeInitialization() method to call BeanPostProcessors associated with the bean. Spring calls afterPropertySet() method to call the specific initialization methods. In case there are any BeanPostProcessors of a bean, the postProcessAfterInitialization() method is called. Destruction: During the destruction of a bean, if bean implements DisposableBean, Spring calls destroy() method.

37. What are the two main groups of methods in a Bean's life cycle?
A Bean in Spring has two main groups of lifecycle methods. Initialization Callbacks: Once all the necessary properties of a Bean are set by the container, Initialization Callback methods are used for performing initialization work. A developer can implement a method afterPropertiesSet() for this work. Destruction Callbacks: When the Container of a Bean is destroyed, it calls the methods in DisposableBean to do any cleanup work. There is a method called destroy() that can be used for this purpose to make Destruction Callbacks.
Recent recommendation from Spring is to not use these methods, since it can strongly couple your code to Spring code.

38. Can we override main lifecycle methods of a Bean in Spring?
Yes, Spring framework allows developers to override the lifecycle methods of a Bean. This is used for writing any custom behavior for Bean.

39. What are Inner beans in Spring?
A bean that is used as a property of another bean is known as Inner bean. It can be defined as a <bean/> element in <property/> or <constructor-arg/> tags. It is not mandatory for an Inner bean to have id or a name. These are always anonymous. Inner bean does not need a scope. By default it is of prototype scope.

40. What is Bean wiring in Spring?
A Spring container is responsible for injecting dependencies between beans. This process of connecting beans is called wiring. Developer mentions in the configuration file, the dependencies between beans. And Spring container reads these dependencies and wires the beans on creation.

41. What is Autowiring in Spring?
Autowiring is a feature of Spring in which containers can automatically wire/connect the beans by reading the configuration file. Developer has to just define the "autowire" attribute in a bean. Spring resolves the dependencies automatically by looking at this attribute of beans that are autowired.

42. What are the different modes of Autowiring supported by Spring?
There are five modes of Autowiring supported by Spring framework:
no: This is the default setting for Autowiring. In this case, we use "ref" mode to mention the explicit bean that is being referred for wiring. E.g. In this example the Employee bean refers to the Manager bean.

```
<bean id="employee" class="com.dept.Employee">
<property name="manager" ref="manager" />
</bean>
<bean id="manager" class="com.dept.Manager" />
```

byName: In this case, Spring container tries to match beans by name during Autowiring. If the name of a bean is the same as the name of bean referred to in autowire byname, then it automatically wires it. E.g. In the following example, Manager bean is wired to Employee bean by Name.

```
<bean id="employee" class="com.dept.Employee"
autowire="byName" />
<bean id="manager" class="com.dept.Manager" />
```

byType: In this case, Spring container checks the properties of beans referred with the attribute byType. Then it matches the type of bean and wires. If it finds more than one such bean of that type, it throws a fatal exception. E.g. In the following example, Manager bean is wired by type to Employee bean.

```
<bean id="employee" class="com.dept.Employee"
autowire="byType" />
<bean id="manager" class="com.dept.Manager" />
```

constructor: In this case, Spring container looks for the byType attribute in the constructor argument. It tries to find the bean with the exact name. If it finds more than one bean of the same name, it throws a fatal exception. This case is similar to byType case. E.g. In the following example "constructor" mode is used for autowiring.

```
<bean id="employee" class="com.dept.Employee"
autowire="constructor" />
<bean id="manager" class="com.dept.Manager" />
```

autodetect: This is an advanced mode for autowiring. In this case, by default Spring tries to find a constructor match. If it does not find a constructor then it uses autowire by Type. E.g. This is an example of auto detect Autowiring.

```
<bean id="employee" class="com.dept.Employee"
autowire="autodetect" />
<bean id="manager" class="com.dept.Manager" />
```

43. What is the purpose of @Configuration annotation?
This annotation is used in a class to indicate that this class is the primary source of bean definitions. This class can also contain inter-bean dependencies that are annotated by @Bean annotation.

44. In Spring framework, what is Annotation-based container configuration?
From the Spring 2.5 version it is possible to provide configuration by using annotation. To turn this configuration on, we need to mention <context:annotation-config/> in spring XML file. Now developers can use annotations like @Required, @Autowired, @Qualifier etc. in a class file to specify the configuration for beans. Spring container can use this information from annotation for creating and wiring the beans.

45. What is @Required annotation?
We use @Required annotation to a property to check whether the property has been set or not. Spring container throws BeanInitializationException if the @Required annotated property is not set.

46. How does the Spring framework make JDBC coding easier for developers?
Spring provides a mature JDBC framework to provide support for JDBC coding. Spring JDBC handled resource management as well as error handling in a generic way. This reduces the work of software developers. They just have to write queries and related statements to fetch the data or to store the data in database.

47. What is the purpose of JdbcTemplate?
Spring framework provides a JdbcTemplate class that contains many convenient methods for regular tasks like- converting data into primitives or objects, executing prepared or callable statements etc. This class makes it very easy to work with database in our Application and it also provides good support for custom error handling in database access code.

48. What are the benefits of using Spring DAO?
Some of the benefits of using Spring DAO are: It makes it easier to work on different data access methods like- JDBC, Hibernate etc. It provides a consistent and common way to deal with different data access methods. Spring DAO makes it easier to switch between different data persistence frameworks. No need for catching framework specific exceptions.

49. What are the benefits provided by Spring Framework's Transaction Management?
Main benefits provided by Spring Transaction Management are: Consistent: By using Spring Transaction management, we can use a consistent programming model across different transaction APIs like- JPA, JDBC, JTA, Hibernate, JPA, JDO etc. Simplicity: Spring TM provides a simple API for managing the transaction programmatically. Declarative: Spring also supports annotation or xml based declarative transaction management. Integration: Spring Transaction management is easier to integrate with other data access abstractions of Spring.

50. What is Aspect Oriented Programming (AOP)
Aspect Oriented Programming (AOP) is a programming paradigm that promotes programmers to develop code in different modules that can be parallel or in crosscutting concerns. E.g. To develop banking software, one team can work on business logic for Money withdrawal, Money deposit, Money Transfer etc. The other team can work on Transaction Management for committing the transaction across multiple accounts.
In an Auto company, one team can work on software to integrate with different components of a car. The other team can work on how all the components will send signal and current information to a common dashboard.

51. What is an Aspect in Spring?
An Aspect is the core construct of AOP. It encapsulates the behavior that affects multiple classes in a reusable module. An Aspect can have a group of APIs that provide cross-cutting features. E.g. A logging module can be an Aspect in an Application. An application can have multiple of Aspects based on the different requirements. An Aspect can be implemented by using annotation @Aspect on a class.

52. In Spring AOP, what is the main difference between a Concern and a Cross cutting concern?
A Concern in Spring is the behavior or expectation from an application. It can be the main feature that we want to implement in the application. A Cross cutting concern is also a type of Concern. It is the feature or functionality that is spread throughout the application in a thin way. E.g. Security, Logging, Transaction Management etc. are cross cutting concerns in an application.

53. What is a Joinpoint in Spring AOP?
In Spring AOP, Joinpoint refers to a candidate point in application where we can plug in an Aspect. Joinpoint can be a method or an exception or a field getting modified. This is the place where the code of an Aspect is inserted to add new behavior in the existing execution flow.

54. What is an Advice in Spring AOP?

An Advice in Spring AOP, is an object containing the actual action that an Aspect introduces. An Advice is the code of cross cutting concern that gets executed. There are multiple types of Advice in Spring AOP.
1. Before Advice: This type of advice runs just before a method executes. We can use @Before annotation for this.
2. After (finally) Advice: This type of advice runs just after a method executes. Even if the method fails, this advice will run. We can use @After annotation here.
3. After Returning Advice: This type of advice runs after a method executes successfully. @AfterReturning annotation can be used here.
4. After Throwing Advice: This type of advice runs after a method executes and throws an exception. The annotation to be used is @AfterThrowing.
5. Around Advice: This type of advice runs before and after the method is invoked. We use @Around annotation for this.

55. What is a Pointcut in Spring AOP?
A Pointcut in Spring AOP refers to the group of one or more Joinpoints where an advice can be applied. We can apply Advice to any Joinpoint. But we want to limit the places where a specific type of Advice should be applied. To achieve this we use Pointcut. We can use class names, method names or regular expressions to specify the Pointcuts for an Advice.

56. What is a Target object in Spring AOP?
A Target object is the object that gets Advice from one or more Aspects. This is also known as an advised object. In most cases it is a proxy object.

57. What is Weaving in Spring AOP?
In Aspect oriented programming, linking Aspects with the other application types creates an Advised object. This process is known as Weaving. Without Weaving, we just have a definition of Aspects. Weaving makes use realize the full potential of the AOP. Weaving can be done at compile time, load time or at run time.

50. What is DispatcherServlet?
In Spring MVC, DispatcherServlet is the core servlet that is responsible for handling all the requests and dispatching these to handlers. Dispatcher servlet knows the mapping between the method to be called and the browser request. It calls the specific method and combines the results with the matching JSP to create an html document, and then sends it back to the browser. In case of RMI invocation, it sends back response to the client application.a Spring MVC web application can have more than one DispatcherServlets. Each DispatcherServlet has to operate in its own namespace. It has to load its own ApplicationContext with mappings, handlers, etc. Only the root application context will be shared among these Servlets.

51. What is WebApplicationContext in Spring MVC?

WebApplicationContext is the child of plain ApplicationContext. It is used in web applications. It provides features to deal with webrelated components like- controllers, view resolvers etc. A Web Application can have multiple WebApplicationContext to handle requests. Each DispatcherServlet is associated with one WebApplicationContext.

52. What is Controller in Spring MVC framework?
Controller is an interface in Spring MVC. It receives HttpServletRequest and HttpServletResponse in web app just like an HttpServlet, but it is able to participate in an MVC flow. Controllers are similar to a Struts Action in a Struts based Web application. Spring recommends that the implementation of Controller interface should be a reusable, thread-safe class, capable of handling multiple HTTP requests throughout the lifecycle of an application. It is preferable to implement a Controller by using a JavaBean. Controller interprets user input and transforms it into a model. The model is represented to the user by a view. Spring implements a controller in a very generic way. This enables us to create a wide variety of controllers. We use @ Controller annotation to indicate that a class is a Controller in Spring MVC. The dispatcher in Spring scans for @Controller annotated classes for mapped methods and detects @RequestMapping.

53. What is @RequestMapping annotation in Spring?
In Spring MVC, we use @RequestMapping annotation to map a web request to either a class or a handler method. In @RequestMapping we can specify the path of URL as well as HTTP method like- GET, PUT, POST etc. @RequestMapping also supports specifying HTTP Headers as attributes. We can also map different media types produced by a controller in @RequestMapping. We use HTTP Header Accepts for this purpose. E.g. @RequestMapping(value = "/test/mapping", method = GET, headers = "Accept=application/json").

54. Name some popular Spring framework annotations that you use in your project?
Spring has many Annotations to serve different purposes. For regular use we refer following popular Spring annotations:
@Controller: This annotation is for creating controller classes in a Spring MVC project.
@RequestMapping: This annotation maps the URI to a controller handler method in Spring MVC.
@ResponseBody: For sending an Object as response we use this annotation.
@PathVariable: To map dynamic values from a URI to handler method arguments, we use this annotation.
@Autowired: This annotation indicates to Spring for auto-wiring dependencies in beans.
@Service: This annotation marks the service classes in Spring.
@Scope: We can define the scope of Spring bean by this annotation.
@Configuration: This an annotation for Java based Spring configuration.
@Aspect, @Before, @After, @Around, @Joinpoint, @Pointcut: These are the annotations in Spring for AspectJ AOP.

55. What are the different types of events provided by Spring framework?

Spring framework provides the following five events for Context:
ContextRefreshedEvent: Whenever ApplicationContext is initialized or refreshed, Spring publishes this event. We can also raise it by using refresh() method on ConfigurableApplicationContext interface.
ContextStartedEvent: When ApplicationContext is started using start() method on ConfigurableApplicationContext interface, ContextStartedEvent is published. We can poll databases or restart any stopped application after receiving this event.
ContextStoppedEvent: Spring publishes this event when ApplicationContext is stopped using the stop() method on the ConfigurableApplicationContext interface. This is used for doing any cleanup work.
ContextClosedEvent: Once the ApplicationContext is closed using close() method, ContextClosedEvent is published. Once a context is closed, it is the last stage of its lifecycle. After this it cannot be refreshed or restarted.
RequestHandledEvent: This is a web specific event that informs to all beans that an HTTP request has been serviced.

56. What is the difference between DispatcherServlet and ContextLoaderListener in Spring?
DispatcherServlet is the core of Spring MVC application. It loads the Spring bean configuration file and initializes all the beans mentioned in the config file. In case we have enabled annotations in the Spring config file, it also scans the packages and configures any bean annotated with @Component, @Controller, @Repository or @Service annotations. ContextLoaderListener is a listener to start up and shut down Spring's root WebApplicationContext. ContextLoaderListener links the lifecycle of ApplicationContext to the lifecycle of the ServletContext. It automates the creation of ApplicationContext. It can also be used to define shared beans used across different spring contexts.

57. What is Spring Boot?
Spring Boot is a ready made solution to create Spring applications with production grade features. It favors convention over configuration. We can embed Tomcat or Jetty in an application created with Spring Boot. Spring Boot automatically configures Spring in an Application. It does not require any code generation or xml configuration. It is an easy solution to create applications that can run stand-alone.

58. What is the Hibernate framework?
Hibernate is a popular Object Relational Mapping (ORM) framework of Java. It helps in mapping the Object Oriented Domain model to Relational Database tables. Hibernate is a free software distributed under GNU license. Hibernate also provides implementation of Java Persistence API (JPA). In simple words, it is a framework to retrieve and store data from database tables from Java.

59. What is an Object Relational Mapping (ORM)?
Object Relational Mapping (ORM) is a programming technique to map data from a relational database to an Object oriented domain model. This is the core of the

Hibernate framework. In the case of Java, most of the software is based on OOPS design. But the data stored in Database is based on the Relation Database Management System (RDBMS). ORM helps in data retrieval in an Object Oriented way from an RDBMS. It reduces the effort of developers in writing queries to access and insert data.

60. What is the purpose of Configuration Interface in Hibernate?
Configuration interface can be implemented in an application to specify the properties and mapping documents for creating a SessionFactory in Hibernate. By default, a new instance of Configuration uses properties mentioned in hibernate.properties file. Configuration is mainly an initialization time object that loads the properties and helps in creating SessionFactory with these properties. In short, Configuration interface is used for configuring Hibernate framework in an application.

61. What are the key characteristics of Hibernate?
Hibernate has the following key characteristics:
Object/Relational Mapping (ORM): Hibernate provides ORM capabilities to developers. So then I can write code in the Object model for connecting with data in the Relational model.
JPA Provider: Hibernate provides an excellent implementation of Java Persistence API (JPA) specification.
Idiomatic persistence: Hibernate provides persistence based on natural Object-oriented idioms with full support for inheritance, polymorphism, association, composition, and the Java collections framework. It can work with any data for persistence.
High Performance: Hibernate provides a high level of performance supporting features like- lazy initialization, multiple fetching strategies, optimistic locking etc. Hibernate does not need its own database tables or fields. It can generate SQL at system initialization to provide better performance at runtime.
Scalability: Hibernate works well in multi server clusters. It has built in scalability support. It can work well for small projects as well as for large business software.
Reliable: Hibernate very reliable and stable framework. This is the reason for its worldwide acceptance and popularity among the developer community.
Extensible: Hibernate is quite generic in nature. It can be configured and extended as per the use case of application.

62. Can you tell us about the core interfaces of Hibernate framework?
The core interfaces of Hibernate framework are as follows:
Configuration: Configuration interface can be implemented in an application to specify the properties and mapping documents for creating a SessionFactory in Hibernate. Hibernate application bootstraps by using this interface.
SessionFactory: In Hibernate, SessionFactory is used to create and manage Sessions. Generally, there is one SessionFactory created for one database. It is a thread-safe interface that works well in multithreaded applications.

Session: Session is a lightweight object that is used at runtime between a Java application and Hibernate. It contains methods to create, read and delete operations for entity classes. It is a basic class that abstracts the concept of persistence.
Transaction: This is an optional interface. It is a short lived object that is used for encapsulating the overall work based on the unit of work design pattern. A Session can have multiple Transactions.
Query: This interface encapsulates the behavior of an object oriented query in Hibernate. It can accept parameters and execute the queries to fetch results. Same query can be executed multiple times.
Criteria: This is a simplified API to retrieve objects by creating Criterion objects. It is very easy to use for creating Search like features.

63. What are the steps for creating a SessionFactory in Hibernate?

```
Configuration config = new Configuration();
config.addResource("testInstance/configuration.hbm.xml");
config.setProperties( System.getProperties() );
SessionFactory sessions = config.buildSessionFactory();
```

64. Why do we use POJO in Hibernate?
POJO stands for Plain Old Java Objects. A POJO is a java bean with getter and setter methods for each property of the bean. It is a simple class that encapsulates an object's properties and provides access through setters and getters. Some of the reasons for using POJO in Hibernate are:
POJO emphasizes the fact that this class is a simple Java class, not a heavy class like EJB.
POJO is a well-constructed class, so it works well with Hibernate proxies.
POJO also comes with a default constructor that makes it easier to persist with a default constructor.

65. What is Hibernate Query Language (HQL)?
Hibernate Query Language is also known as HQL. It is an Object Oriented language. But it is similar to SQL. HQL works well with persistent objects and their properties. HQL does not work on database tables. HQL queries are translated into native SQL queries specific to a database. HQL supports direct running of native SQL queries also. But it creates an issue in Database portability.

66. What are the advantages of the Hibernate framework over JDBC?
Main advantages of Hibernate over JDBC are as follows: Database Portability: Hibernate can be used with multiple types of database with easy portability. In JDBC, developers have to write database specific native queries. These native queries can reduce the database portability of the code. Connection Pool: Hibernate handles connection pooling very well. JDBC requires connection pooling to be defined by the developer. Complexity: Hibernate handles complex query scenarios very well with its internal API like Criteria. So

developers need not gain expertise in writing complex SQL queries. In JDBC application developer writes most of the queries.

67. What is the use of Dirty Checking in Hibernate?
Dirty Checking is a very useful feature of Hibernate for writing to database operations. Hibernate monitors all the persistent objects for any changes. It can detect if an object has been modified or not. By Dirty Checking, only those fields of an object are updated that require any change in them. It reduces the time-consuming database write operations.

68. What are the different ORM levels in Hibernate?
There are four different ORM levels in Hibernate:
Pure Relational ORM: At this level the entire application is designed around the relational model. All the operations are SQL based at this level.
Light Object Mapping: At this level entity classes are mapped manually to relational tables. Business logic code is hidden from data access code. Applications with less number of entities use this level.
Medium Object Mapping: In this case, application is designed around an object model. Most of the SQL code is generated at compile time. Associations between objects are supported by the persistence mechanism. Object-oriented expression language is used to specify queries.
Full Object Mapping: This is one of the most sophisticated object modeling levels. It supports composition, inheritance, polymorphism and persistence. The persistent classes do not inherit any special base class at this level. There are efficient fetching and caching strategies implemented transparently to the application.

69. What is Query Cache in Hibernate?
Hibernate provides a Query Cache to improve the performance of queries that run multiple times with the same parameters. At times Query Caching can reduce the performance of Transactional processing. By default Query Cache is disabled in Hibernate. It has to be used based on the benefits gained by it in performance of the queries in an application.

70. What is the Unit of Work design pattern?
Unit of Work is a design pattern to define business transactions. A Unit of Work is a list of ordered operations that we want to run on a database together. Either all of these go together or none of these goes. Most of the time, we use term business transactions in place of the Unit of Work. Egg. In case of money transfer from account A to B, the unit of work can be two operations: Debit account A and Credit account B in a sequence. Both these operations should happen together and in the right sequence.

71. How does Transaction management work in Hibernate?
In Hibernate we use the Session interface to get a new transaction. Once we get the transaction we can run business operations in that transaction. At the end of successful

business operations, we commit the transaction. In case of failure, we rollback the transaction. Sample code is a follows:

```
Session s = null;
Transaction trans = null;
try {
s = sessionFactory.openSession();
trans = s.beginTransaction();
doTheAction(s);
trans.commit();
} catch (RuntimeException exc) {
trans.rollback();
} finally {
s.close();
}
```

72. How can we mark an entity/collection as immutable in Hibernate?
In Hibernate, by default an entity or collection is mutable. We can add, delete or update an entity/collection. To mark an entity/collection as immutable, we can use one of the Following: @Immutable: We can use the annotation @Immutable to mark an entity/collection immutable. XML file: We can also set the property mutable=false in the XML file for an entity to make it immutable.

73. How can we auto-generate the primary key in Hibernate?
We can use the primary key generation strategy of type GenerationType.AUTO to auto-generate primary key while persisting an object in Hibernate. Egg.

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int id;
```

We can leave it null/0 while persisting and Hibernate automatically generates a primary key for us. Sometimes, AUTO strategy refers to a SEQUENCE instead of an IDENTITY .

74. What is the first level of cache in Hibernate?
A Hibernate Session is the first level of cache for persistent data in a transaction. The second level of cache is at JVM or SessionFactory level. In Hibernate, we can use different cache providers for implementing second level cache at JVM/SessionFactory level. Some of these are:

Hashtable
EHCache
OSCache
SwarmCache
JBoss Cache 1.x
JBoss Cache 2

75. What are the different fetching strategies in Hibernate?

Hibernate 3 onwards there are following fetching strategies to retrieve associated objects:
Join fetching: In Join strategy Hibernate uses OUTER join to retrieve the associated instance or collection in the same SELECT.
Select fetching: In Select strategy, Hibernate uses a second SELECT to retrieve the associated entity or collection. We can explicitly disable lazy fetching by specifying lazy="false". By default lazy fetching is true.
Subselect fetching: In Subselect strategy, Hibernate uses a second SELECT to retrieve the associated collections for all entities retrieved in a previous query or fetch.
Batch fetching: In Batch strategy, Hibernate uses a single SELECT to retrieve a batch of entity instances or collections by specifying a list of primary or foreign keys. This is a very good performance optimization strategy for select fetching.

76. What is the difference between Immediate fetching and Lazy collection fetching?
In Immediate fetching an association, collection or attribute is retrieved at the same time when the owner is loaded. But in Lazy collection fetching, a collection is fetched only when an operation is invoked on that collection by client application. This is the default fetching strategy for collections in Hibernate. Lazy fetching is better from a performance perspective.

77. What are the different strategies for cache mapping in Hibernate?
Hibernate provides following strategies for cache mapping:
Read only: If an application requires caching only for read but not for write operations, then we can use this strategy. It is very simple to use and give very good performance benefit. It is also safe to use in a cluster environment.
Read/Write: If an application also needs caching for write operations, then we use Read/Write strategy. Read/write cache strategy should not be used if there is requirement for serializable transaction isolation level. If we want to use it in a cluster environment, we need to implement locking mechanism.
Nonstrict Read/Write: If an application only occasionally updates the data, then we can use this strategy. It cannot be used in systems with serializable transaction isolation level requirement.
Transactional: This strategy supports full transactional cache providers like JBoss TreeCache.

78. What is the difference between a Set and a Bag in Hibernate?
A Bag in Hibernate is an unordered collection. It can have duplicate elements. When we persist an object in a bag, there is no guarantee that bag will maintain any order.
A Set in Hibernate can only store unique objects. If we add the same element to set second time, it just replaces the old one. By default a Set is unordered collection in Hibernate.

79. How can we monitor the performance of Hibernate in an application?

We can use following ways to monitor Hibernate performance: Monitoring SessionFactory: Since there is one SessionFactory in an application, we can collect the statistics of a SessionFactory to monitor the performance. Hibernate provides sessionFactory.getStatistics() method to get the statistics of SessionFactory. Hibernate can also use JMX to publish metrics.
Metrics: In Hibernate we can also collect other metrics likenumber of open sessions, retrieved JDBC connections, cache hit, miss etc. These metrics give great insight into the performance of Hibernate. We can tune Hibernate settings and strategies based on these metrics.

80. What is ORM metadata?
ORM uses metadata for its internal work. ORM maintains metadata to generate code used for accessing columns and tables. ORM maps classes to tables and stores this information in Metadata. It maps fields in classes to columns in tables. These kinds of mappings are also part of Metadata. Application developers can also access Hibernate Metadata by using ClassMetadata and CollectionMetadata interfaces and Type hierarchy.

81. What is the difference between load() and get() method in Hibernate?
In Hibernate, load() and get() methods are quite similar in functionality. The main difference is that load() method will throw an ObjectNotFoundException if row corresponding to an object is not found in the database.
On the other hand, get() method returns null value when an object is not found in the database. It is recommended that we should use load() method only when we are sure that object exists in database.

82. What are the two locking strategies in Hibernate?
There are two popular locking strategies that can be used in Hibernate:
Optimistic: In Optimistic locking we assume that multiple transactions can complete without affecting each other. So we let the transactions do their work without locking the resources initially. Just before the commit, we check if any of the resource has changed by another transaction, then we throw exception and rollback the transaction.
Pessimistic: In Pessimistic locking we assume that concurrent transactions will conflict while working with same resources. So a transaction has to first obtain lock on the resources it wants to update. The other transaction can proceed with same resource only after the lock has been released by previous transaction.

83. What is the use of version number in Hibernate?
Version number is used in optimistic locking in Hibernate. When a transaction modifies an object, it increments its version. Based on version number, second transaction can determine if the object it has read earlier has changed or not. If the version number at the time of write is different than the version number at the time of read, then we should not commit the transaction.

84. What inheritance mapping strategies are supported by Hibernate?

Hibernate supports following inheritance mapping strategies between classes and tables: Table per class hierarchy: In case of multiple types of books, we can have one book class and one book table. We can store all child classes of book like- HardCoverBook, PaperBackBook etc in same table book. But we can identify the subclasses by a BookType column in Book table. Table per subclass: In this case we can have separate table for each kind of book. HardCoverBook table for HardCoverBook book class. PaperBackBook table for PaperBackBook book class. And there will be a parent table, Book for Book class. Table per concrete class: In this case also we have separate table for each kind of book. But in this case we have even inherited properties defined inside each table. There is no parent table Book for Book class, since it is not a concrete class.

85. What is a Microservice?

A Microservice is a small and autonomous piece of code that does one thing very well. It is focused on doing well one specific task in a big system. It is also an autonomous entity that can be designed, developed and deployed independently. Generally, it is implemented as a REST service on HTTP protocol, with technology-agnostic APIs. Ideally, it does not share database with any other service.

86. What is the role of architect in Microservices architecture?

Architects, in Microservices architecture, play the role of Town planners. They decide in broad strokes about the layout of the overall software system. They help in deciding the zoning of the components. They make sure components are mutually cohesive but not tightly coupled. They need not worry about what is inside each zone. Since they have to remain up to date with the new developments and problems, they have to code with developers to learn the challenges faced in day-to-day life. They can make recommendations for certain tools and technologies, but the team developing a micro service is ultimately empowered to create and design the service. Remember, a micro service implementation can change with time. They have to provide technical governance so that the teams in their technical development follow principles of Microservice. At times they work as custodians of overall Microservices architecture.

88. What are the disadvantages of using Shared libraries approach to decompose a monolith application?

You can create shared libraries to increase reuse and sharing of features among teams. But there are some downsides to it. Since shared libraries are implemented in same language, it constrains you from using multiple types of technologies. It does not help you with scaling the parts of system that need better performance. Deployment of shared libraries is same as deployment of Monolith application, so it comes with same deployment issues. Shared libraries introduce shared code that can increase coupling in software.

89. What is the preferred type of communication between Microservices? Synchronous or Asynchronous?

Synchronous communication is a blocking call in which client blocks itself from doing anything else, till the response comes back. In Asynchronous communication, client can move ahead with its work after making an asynchronous call. Therefore client is not Blocked. In synchronous communication, a Microservice can provide instant response about success or failure. In real-time systems, synchronous service is very useful. In Asynchronous communication, a service has to react based on the response received in future. Synchronous systems are also known as request/response based. Asynchronous systems are event-based. Synchronous Microservices are not loosely coupled. Depending on the need and critical nature of business domain, Microservices can choose synchronous or asynchronous form of communication.

90. What are the issues in using REST over HTTP for Microservices?
In REST over HTTP, it is difficult to generate a client stub. Some Web-Servers also do not support all the HTTP verbs like- GET, PUT, POST, DELETE etc. Due to JSON or plain text in response, performance of REST over HTTP is better than SOAP. But it is not as good as plain binary communication. There is an overhead of HTTP in each request for communication. HTTP is not well suited for low-latency communications. There is more work in consumption of payload. There may be overhead of serialization, deserialization in HTTP.

VERSION CONTROLLING SYSTEM: GIT

GIT is a mature Distributed Version Control System (DVCS). It is used for Source Code Management (SCM). It is open source software. It was developed by Linus Torvalds, the creator of Linux operating system. GIT works well with a large number of IDEs (Integrated Development Environments) like- Eclipse, InteliJ etc. GIT can be used to handle small and large projects. Most of the GIT distributions are written in C language with Bourne shell. Some of the commands are written in Perl language. There are many GUI for GIT that we can use. Some of these are: GitHub Desktop, GITX-dev, Gitbox, Git-cola, SourceTree, Git Extensions, SmartGit, GitUp.

GIT is made very secure since it contains the source code of an organization. All the objects in a GIT repository are encrypted with a hashing algorithm called SHA1. This algorithm is quite strong and fast. It protects source code and other contents of repository against the possible malicious attacks. This algorithm also maintains the integrity of GIT repository by protecting the change history against accidental changes.

There are following main benefits of GIT:

1. **Distributed System**: GIT is a Distributed Version Control System (DVCS). So you can keep your private work in version control but completely hidden from others. You can work offline as well.
2. **Flexible Workflow**: GIT allows you to create your own workflow. You can use the process that is suitable for your project. You can go for centralized or master-slave or any other workflow.
3. **Fast**: GIT is very fast when compared to other version control systems.

4. **Data Integrity**: Since GIT uses SHA1, data is not easier to corrupt.
5. **Free**: It is free for personal use. So many amateurs use it for their initial projects. It also works very well with large size project.
6. **Collaboration:** GIT is very easy to use for projects in which collaboration is required. Many popular open source software across the globe use GIT.

GIT has very few disadvantages. These are the scenarios when GIT is difficult to use. Some of these are:
1. **Binary Files**: If we have a lot binary files (non-text) in our project, then GIT becomes very slow. E.g. Projects with a lot of images or Word documents.
2. **Steep Learning Curve**: It takes some time for a newcomer to learn GIT. Some of the GIT commands are non-intuitive to a fresher.
3. **Slow remote speed**: Sometimes the use of remote repositories in slow due to network latency. Still GIT is better than other VCS in speed.

The main differences between GIT and SVN are:
1. **Decentralized**: GIT is decentralized. You have a local copy that is a repository in which you can commit. In SVN you have to always connect to a central repository for check-in.
2. **Complex to learn**: GIT is a bit difficult to learn for some developers. It has more concepts and commands to learn. SVN is much easier to learn.
3. **Unable to handle Binary files**: GIT becomes slow when it deals with large binary files that change frequently. SVN can handle large binary files easily.
4. **Internal directory**: GIT creates only .git directory. SVN creates .svn directory in each folder.
5. **User Interface**: GIT does not have good UI. But SVN has good user interfaces.

git init- We use git init command in an existing project directory to start version control for our project. After this we can use git add and git commit commands to add files
to our GIT repository.

git clone- In GIT, we use git clone command to create a copy of an existing GIT repository in our local. This is the most popular way to create a copy of the repository among developers.

We can start work in GIT in following ways: New Project: To create a new repository we use git init command. Existing Project: To work on an existing repository we use git clone command.

git pull- In GIT, git pull internally does a git fetch first and then does a git merge. So pull is a combination of two commands: fetch and merge. We use git pull command to bring our local branch up to date with its remote version. A pull request in GIT is the list of changes that have been pushed to GIT repository. Generally these changes are pushed in a feature branch or hotfix branch. After pushing these changes we create a pull request that contains the changes between master and our feature branch. This pull request is sent to reviewers for reviewing the code and then merging it into develop or release branch.

git push- In GIT, git push command does following two commands:
1. fetch: First GIT, copies all the extra commits from server into local repo and moves origin/master branch pointer to the end of commit chain.
2. merge: Then it merges the origin/master branch into the master branch. Now the master branch pointer moves to the newly created commit. But the origin/master pointer remains there.

git stash- In GIT, sometimes we do not want to commit our code but we do not want to lose also the unfinished code. In this case we use git stash command to record the current state of the working directory and index in a stash. This stores the unfinished work in a stash, and cleans the current branch from uncommitted changes. Now we can work on a clean working directory. Later we can use the stash and apply those changes back to our working directory. At times we are in the middle of some work and do not want to lose the unfinished work, we use git stash command.In case we do not need a specific stash, we use git stash drop command to remove it from the list of stashes. By default, this command removes to latest added stash To remove a specific stash we specify as argument in the git stash drop <stashname> command.

we use git stash apply command to bring back the unfinished work. So the command to apply a stash is: git stash apply Or we can use git stash apply <stashname>

Stage- In GIT, stage is a step before commit. To stage means that the files are ready for commit. Let say, you are working on two features in GIT. One of the features is finished and the other is not yet ready. You want to commit and leave for home in the evening. But you can commit since both of them are not fully ready. In this case you can just stage the feature that is ready and commit that part. Second feature will remain as work in progress.

git config- We can set the configuration options for GIT installation by using git config command. 'git config --list' command to print all the GIT configuration settings in GIT installation.

git add.- GIT gives us a very good feature of staging our changes before commit. To stage the changes we use git add command. This adds our changes from working directory to the index. When we are working on multiple tasks and we want to just commit the finished tasks, we first add finished changes to staging area and then commit it. At this time git add command is very helpful.

git reset- We use git reset command to reset current HEAD to a specific state. By default it reverses the action of git add command. So we use git reset command to undo the changes of git add command.

git commit - $/> git commit –m <message>. GIT commit object contains following information:
SHA1 name: A 40 character string to identify a commit

Files: List of files that represent the state of a project at a specific point of time
Reference: Any reference to parent commit objects
A commit message is a comment that we add to a commit. We can provide meaningful information about the reason for commit by using a commit message. In most of the organizations, it is mandatory to put a commit message along with each commit. Often, commit messages contain JIRA ticket, bug id, defect id etc. for a project.

to see the differences between two commits: git diff <commit#1> <commit#2>

git log- We can use git log command to see the latest commits. To see the three most recent commits we use following command: git log -3. We can search git history by author, date or content. It can even list the commits that were done x days before or after a specific date.A shortlog in GIT is a command that summarizes the git log output.The output of git shortlog is in a format suitable for release announcements.

HEAD: A HEAD is a reference to the currently checked out commit. It is a symbolic reference to the branch that we have checked out. At any given time, one head is selected as the 'current head' This head is also known as HEAD (always in uppercase). There can be any number of heads in a repository. By default there is one head known as HEAD in each repository in GIT.

Branching in GIT: If we are simultaneously working on multiple tasks, projects, defects or features, we need multiple branches. In GIT we can create a separate branch for each separate purpose. Let say we are working on a feature, we create a feature branch for that. In between we get a defect to work on then we create another branch for defect and work on it. Once the defect work is done, we merge that branch and come back to work on feature branch again. So working on multiple tasks is the main reason for using multiple branches. We can create different kinds of branches for following purposes in GIT:
Feature branches: These are used for developing a feature.
Release branches: These are used for releasing code to production.
Hotfix branches: These are used for releasing a hotfix to production for a defect or emergency fix.

We use following command to create a new branch in GIT: $/> git checkout –b <branchname>

We do the development work on a feature branch that is created from master branch. Once the development work is ready we use git merge command to merge it into master branch.

There are many ways to do branching in GIT. One of the popular ways is to maintain two branches:
master: This branch is used for production. In this branch HEAD is always in production ready state.
develop: This branch is used for development. In this branch we store the latest code developed in project. This is work in progress code.

Once the code is ready for deployment to production, it is merged into master branch from develop branch.

To get all the list of branch merged in master We can use following commands for this purpose:
git branch --merged master : This prints the branches merged into master
git branch --merged lists : This prints the branches merged into HEAD (i.e. tip of current branch)
git branch --no-merged : This prints the branches that have not been merged By default this applies only to local branches.
We can use -a flag to show both local and remote branches. Or we can use -r flag to show only the remote branches.

To delete an unwanted branch we use following command: git branch –d <branchname>
To forcibly delete an unwanted branch with unmerged changes:git branch –D <branchname>
In GIT, we can use git checkout <new branchname> command to switch to a new branch.

git rebase: Another alternative of merging in GIT is rebasing. It is done by git rebase command.Rebasing is the process of moving a branch to a new base commit. It is like rewriting the history of a branch. In Rebasing, we move a branch from one commit to another. By this we can maintain linear project history. Once the commits are pushed to a public repository, it is not a good practice to use Rebasing.
Git command for rebasing is: git rebase <new-commit>
Tom compress last n commits a single commit, we use git rebase command. This command compresses multiple commits and creates a new commit. It overwrites the history of commits. It should be done carefully, since it can lead to unexpected results.

Merge conflict: A merge conflict in GIT is the result of merging two commits. Sometimes the commit to be merged and current commit have changes in same location. In this scenario, GIT is not able to decide which change is more important. Due to this GIT reports a merge conflict. It means merge is not successful. We may have to manually check and resolve the merge conflict.When GIT reports merge conflict in a file, it marks the lines as follows: E.g. the business days in this week are

```
<<<<<<< HEAD
five
=======
six
>>>>>>> branch-feature
```

To resolve the merge conflict in a file, we edit the file and fix the conflicting change. In above example we can either keep five or six. After editing the file we run git add command followed by git commit command. Since GIT is aware that it was merge conflict, it links this change to the correct commit.

git rerere- In GIT, rerere is a hidden feature. The full form of rerere is “reuse recorded resolution”. By using rerere, GIT remembers how we’ve resolved a hunk conflict. The next time GIT sees the same conflict, it can automatically resolve it for us.

git clone vs git remote- The main difference between git clone and git remote is that git clone is used to create a new local repository whereas git remote is used in an existing repository. git remote adds a new reference to existing remote repository for tracking further changes. git clone creates a new local repository by copying another repository from a URL.

To put a local repository on GitHub, we first add all the files of working directory into local repository and commit the changes. After that we call git remote add <Remote Repo URL> command to add the local repository on GitHub server. Once it is added, we use git push command to push the contents of local repository to remote GitHub server.We can use command git remote rename for changing the name of a remote repository. This changes the short name associated with a remote repository in your local. Command would look as follows:
git remote rename repoOldName repoNewName

git diff- In GIT, git diff command is used to display the differences between 2 versions, or between working directory and an index, or between index and most recent commit. It can also display changes between two blob objects, or between two files on disk in GIT. It helps in finding the changes that can be used for code review for a feature or bug fix. Three most popular git diff commands are as follows:
git diff: It displays the differences between working directory and the index.
git diff –cached: It displays the differences between the index and the most recent commit.
git diff HEAD: It displays the differences between working directory and the most recent commit

git status- In GIT, git status command mainly shows the status of working tree. It shows following items:
1. The paths that have differences between the index file and the current HEAD commit.
2. The paths that have differences between the working tree and the index file
3. The paths in the working tree that are not tracked by GIT. Among the above three items, first item is the one that we commit by using git commit command. Item two and three can be committed only after running git add command.

In GIT, git diff shows the differences between different commits or between the working directory and index. Whereas, git status command just shows the current status of working tree.

git rm –r- We use git rm –r to recursively remove all files from a leading directory.

git hooks- Git hooks are scripts that can run automatically on the occurrence of an event in a Git repository. These are used for automation of workflow in GIT. Git hooks also help in customizing the internal behavior of GIT. These are generally used for enforcing a GIT commit policy.Git hooks are generally written in shell and PERL scripts. But these can be written in any other language as long as it has an executable. Git hooks can also be written in Python script.

Pre-receive hook is invoked when a commit is pushed to a destination repository. Any script attached to this hook is executed before updating any reference. This is mainly used to enforce development best practices and policies.
Update hook is similar to pre-receive hook. It is triggered just before any updates are done. This hook is invoked once for every commit that is pushed to a destination repository.
Post-receive hook is invoked after the updates have been done and accepted by a destination repository. This is mainly used to configure deployment scripts. It can also invoke Continuous Integration (CI) systems and send notification emails to relevant parties of a repository.

git init –bare- A repository created with git init –bare command is a bare repository in GIT. The bare repository does not contain any working or checked out copy of source files. A bare repository stores git revision history in the root folder of repository instead of in a .git subfolder. It is mainly used for sharing and collaborating with other developers. We can create a bare repository in which all developers can push their code. There is no working tree in bare repository, since no one directly edits files in a bare repository.

git revert- Internally, git revert command creates a new commit with patches that reverse the changes done in previous commits.

git clean –x-git clean command to recursively clean the working tree. It removes the files that are not under version control in GIT.

git tag- We use git tag command to add, delete, list or verify a tag object in GIT. Tag objects created with options –a, -s, -u are also known as annotated tags. Annotated tags are generally used for release.In GIT, we can create two types of Tags.
Lightweight Tag: A lightweight tag is a reference that never moves. We can make a lightweight tag by running a command similar to following:
$ git update-ref refs/tags/v1.0 dad0dab538c970e37ea1e769cbbde608743bc96d
Annotated Tag: An annotated tag is more complex object in GIT. When we create an annotated tag, GIT creates a tag object and writes a reference to point to it rather than directly to the commit. We can create an annotated tag as follows:
$ git tag -a v1.1 1d410eabc13591cb07496601ebc7c059dd55bfe9 - m 'test tag'

git cherry-pick-A git cherry-pick is a very useful feature in GIT. By using thiscommand we can selectively apply the changes done by existing commits. In case we want to selectively release a feature, we can remove the unwanted files and apply only selected commits.

git diff-tree -r <hash of commit>-Every commit in GIT has a hash code. This hash code uniquely represents the GIT commit object. We can use git diff-tree command to list the name of files that were changed in a commit. The command will be as follows: git diff-tree -r <hash of commit> By using -r flag, we just get the list of individual files.

git bisect-In GIT we can use git bisect command to find the commit that has introduced a bug in the system. GIT bisect command internally uses binary search algorithm to find the commit that introduced a bug. We first tell a bad commit that contains the bug and a good commit that was present before the bug was introduced. Then git bisect picks a commit between those two endpoints and asks us whether the selected commit is good or bad. It continues to narrow down the range until it discovers the exact commit responsible for introducing the bug.

git fork-In case of projects where we do not have push access, we can just fork the repository. By running git fork command, GIT will create a personal copy of the repository in our namespace. Once our work is done, we can create a pull request to merge our changes on the real project.

git grep- GIT is shipped along with a grep command that allows us to search for a string or regular expression in any committed tree or the working directory.
By default, it works on the files in your current working directory

If we don't want to type username/password with every single timepush, we can set up a "credential cache". It is kept in memory for a few minutes. We can set it by running:
git config --global credential.helper cache.

Is origin a special branch in GIT?
No, origin is not a special branch in GIT. Branch origin is similar to branch master. It does not have any special meaning in GIT. Master is the default name for a starting branch when we run git init command. Origin is the default name for a remote when we run git clone command. If we run git clone -o myOrigin instead, then we will have myOrigin/master as our default remote branch.

Git protocol is a mechanism for transferring data in GIT. It is a special daemon. It comes pre-packaged with GIT. It listens on a dedicated port 9418. It provides services similar to SSH protocol. But Git protocol does not support any authentication. So on plus side, this is a very fast network transfer protocol. But it lacks authentication.

GIT maintains following three trees: HEAD: This is the last commit snapshot. Index: This is the proposed next commit snapshot. Working Directory: This is the sandbox for doing changes

GIT has following three main steps in a simple workflow: Checkout the project from HEAD to Working Directory. Stage the files from Working Directory to Index. Commit the changes from Index to HEAD.

GIT provides an option ignore-space-change in git merge command to ignore the conflicts related to whitespaces. The command to do so is as follows: git merge -Xignore-space-change whitespace.

git blame- In GIT, git blame is a very good option to find the person who changed a specific line. When we call git blame on a file, it displays the commit and name of a person responsible for making change in that line. Following is a sample:

```
$ git blame -L 12,19 HelloWorld.java
```