VIGNAN'S LARA
INSTITUTE OF TECHNOLOGY & SCIENCE
Approved by AICTE New Delhi & Affiliated to JNTUK Kakinada
Accredited by NAAC 'A+' and NBA | ISO 9001 : 2015
Vadlamudi - 522 213, Guntur District

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SUBJECT: API & Micro Services

QUESTION BANK WITH ANSWERS

Regulation: R20 **A.Y.**:2023-24

Name of the faculty: Asha Priyadarshini.M **Year & Sem**: IV- I

UNIT - I

1. Briefly explain about the need of Spring Framework.

Imagine you are building a customer service application for a company. The application should handle customer inquiries, create support tickets, and store customer data in a database. Without using the Spring Framework, you might start with a straightforward implementation.

Customer Class (Plain Java Approach):

```
public class Customer {
    private int id;
    private String name;
    private String email;
    // Getters and setters
}
```

CustomerDAO Class (Plain Java Approach):

```
public class CustomerDAO {
    public void saveCustomer(Customer customer) {
        // Code to save the customer data to the database
    }
    public Customer getCustomerById(int id) {
        // Code to retrieve the customer from the database using the provided ID
        return null;
    }
    public List<Customer> getAllCustomers() {
        // Code to retrieve all customers from the database
        return new ArrayList<>();
    }
}
```

CustomerService Class (Plain Java Approach):

```
public class CustomerService {
    private CustomerDAO customerDAO;
    public CustomerService() {
        customerDAO = new CustomerDAO();
    }

    public void addCustomer(Customer customer) {
        // Perform validation or business logic if needed
        customerDAO.saveCustomer(customer);
    }

    public Customer getCustomerById(int id) {
        // Perform business logic if needed
        return customerDAO.getCustomerById(id);
    }

    public List<Customer> getAllCustomers() {
        // Perform business logic if needed
        return customerDAO.getAllCustomers();
    }
}
```

As the application grows, you might have more components, such as logging, security, and transaction management, which can make your codebase complex and harder to maintain.
Now, let's see how the Spring Framework can simplify this customer service application:

Customer Class (Spring Approach):

```
public class Customer {
    private int id;
    private String name;
    private String email;
    // Getters and setters
}
```

CustomerDAO Class (Spring Approach):

```
@Repository // Annotation to indicate that this class handles data access
public class CustomerDAO {
    public void saveCustomer(Customer customer) {
        // Code to save the customer data to the database
    }
    public Customer getCustomerById(int id) {
        // Code to retrieve the customer from the database using the provided ID
        return null;
    }
    public List<Customer> getAllCustomers() {
        // Code to retrieve all customers from the database
        return new ArrayList<>();
    }
}
```

CustomerService Class (Spring Approach):

```
@Service // Annotation to indicate that this class provides a service/business logic
public class CustomerService {
    private final CustomerDAO customerDAO;
   @Autowired // Annotation for automatic dependency injection
    public CustomerService(CustomerDAO customerDAO) {
        this.customerDAO = customerDAO;
    }
   public void addCustomer(Customer customer) {
        // Perform validation or business logic if needed
        customerDAO.saveCustomer(customer);
    }
    public Customer getCustomerById(int id) {
        // Perform business logic if needed
        return customerDAO.getCustomerById(id);
    }
    public List<Customer> getAllCustomers() {
        // Perform business logic if needed
        return customerDAO.getAllCustomers();
    }
}
```

With the Spring Framework:

Dependency Injection (DI): The CustomerService class no longer creates the CustomerDAO instance manually. Instead, it relies on Spring to inject the appropriate CustomerDAO instance when the CustomerService is constructed. This makes your code more modular and allows easier unit testing.

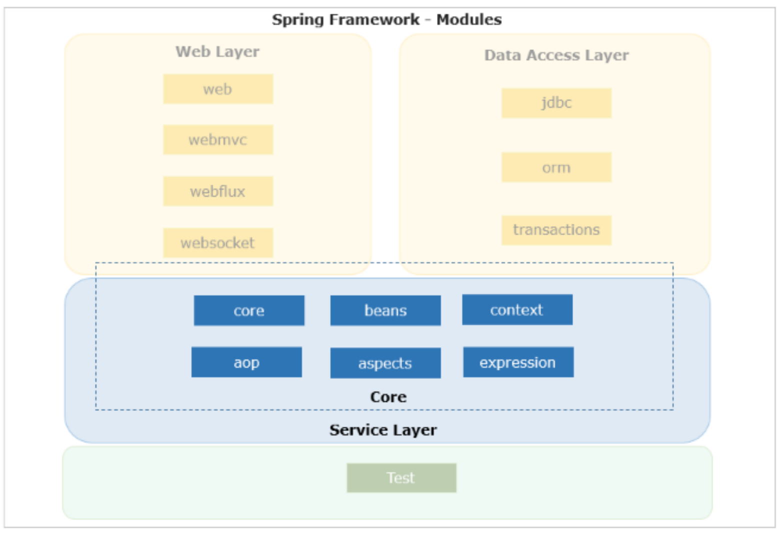
Inversion of Control (IoC): The flow of control in managing objects and their lifecycles is shifted to Spring. You don't have to worry about managing the creation and initialization of objects anymore.

Data Access and Transactions: The CustomerDAO class can utilize Spring's data access and transaction management features, making it easier to interact with the database and ensure data integrity.

By using the Spring Framework, your customer service application becomes more maintainable, scalable, and easier to extend with additional functionalities, such as adding new services, applying cross-cutting concerns like logging, and managing transactions without much hassle. Spring enables you to focus on the core business logic of your application, while it takes care of many of the underlying infrastructure concerns.

2. **A. Explain about core container modules and data access modules in the Spring Framework.**

   Spring Modules - Core Container

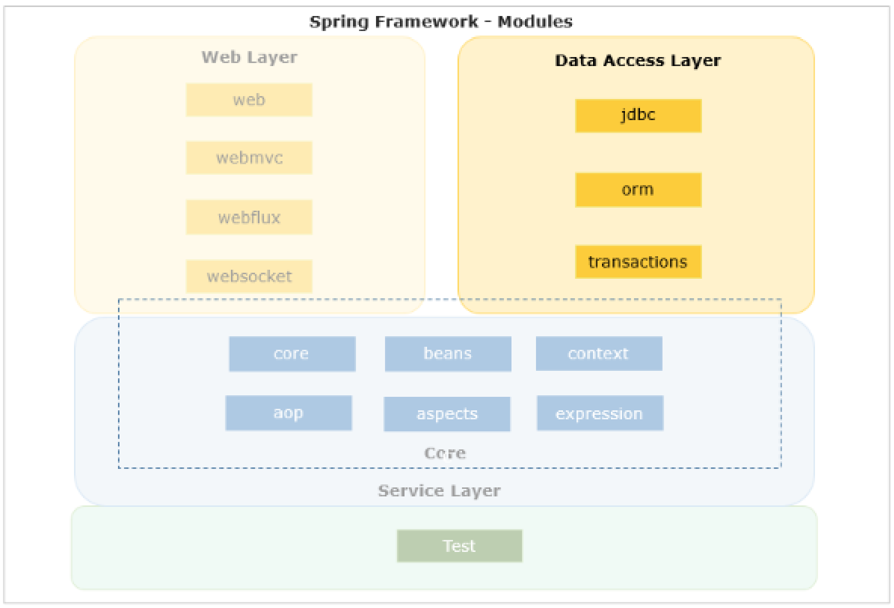


In the Spring Framework, the core container provides the foundational modules that form the backbone of the entire Spring ecosystem. Here are the key modules of the Spring core container:

1. **Spring Core:** This module is the foundation of the Spring Framework and provides the core functionality for dependency injection and IoC. It includes the BeanFactory interface, which is responsible for managing and providing beans (objects) and their dependencies. The core module is found in the spring-core JAR.
2. **Spring Beans:** This module defines the concept of a "bean" in the Spring context. A bean is an object that is managed by the Spring container and is configured using XML, annotations, or Java-based configuration. The @Component, @Autowired, and other related annotations are part of this module. The Spring Beans module is found in the spring-beans JAR.
3. **Spring Context:** The Spring Context module builds on top of the core container and provides a more advanced application context. It includes features like internationalization, event propagation, resource loading, and environment profiles. The Spring Context module is found in the spring-context JAR.
4. **Spring Expression Language (SpEL):** SpEL is a powerful expression language that allows you to define expressions for evaluating values at runtime. It is widely used in annotations such as @Value and @Conditional expressions. The SpEL module is found in the spring-expression JAR.
5. **Spring AOP (Aspect-Oriented Programming):** The AOP module enables aspect-oriented programming in Spring applications. AOP allows you to separate cross-cutting concerns, such as logging, security, and caching, from the core business logic. The Spring AOP module is found in the spring-aop JAR.
6. **Spring Test:** The Spring Test module provides support for testing Spring applications. It includes integration testing features for loading Spring contexts, managing transactional behavior, and more. The Spring Test module is found in the spring-test JAR.

By utilizing the core container modules, developers can achieve loose coupling between components, easily manage dependencies, and take advantage of various Spring features, making the development process more efficient and maintainable.

Spring Modules - Data Access/Integration

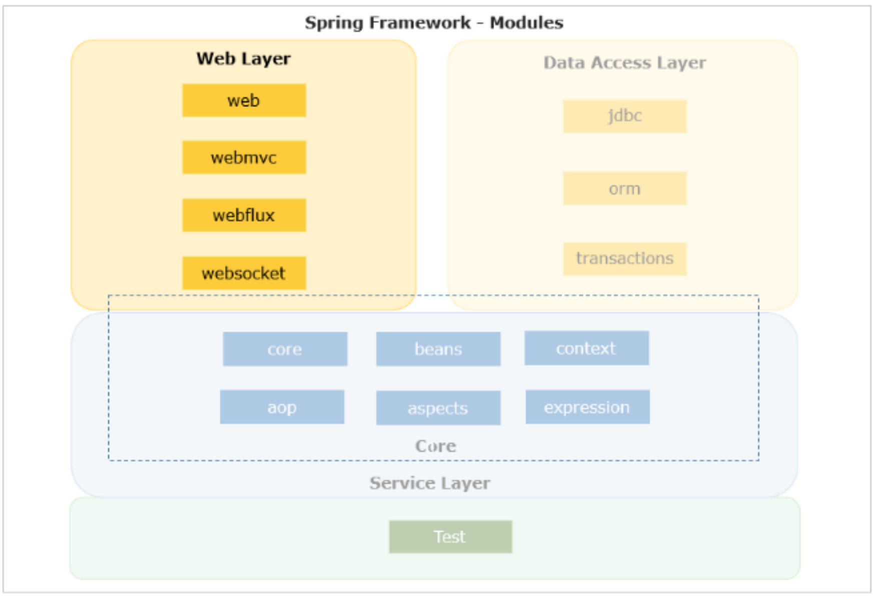


In the Spring Framework, data access is a critical aspect of building enterprise-level applications. Here are the key data access modules in the Spring Framework:

1. **Spring JDBC:** The Spring JDBC module provides support for working with traditional JDBC (Java Database Connectivity) to interact with relational databases. It simplifies common JDBC operations like connection management, query execution, and exception handling. It offers features such as JdbcTemplate and NamedParameterJdbcTemplate to execute SQL queries and map results to Java objects efficiently.
2. **Spring ORM (Object-Relational Mapping):** The Spring ORM module integrates the Spring Framework with popular ORM frameworks like Hibernate, JPA (Java Persistence API), and JDO (Java Data Objects). It provides seamless integration and configuration options to work with ORM frameworks, allowing developers to focus on business logic rather than database interactions.
3. **Spring Transaction Management:** The Spring Transaction Management module offers declarative and programmatic transaction management capabilities. It supports both local and distributed transactions. By using annotations or XML-based configuration, developers can define transaction boundaries, and the Spring container handles transaction management transparently.
4. **Spring Data JPA:** Spring Data JPA is part of the larger Spring Data project and simplifies working with JPA-based data repositories. It provides powerful repository abstractions, reducing the amount of boilerplate code required for database interactions.
5. **Spring Data MongoDB:** For NoSQL databases like MongoDB, Spring Data MongoDB provides integration and a convenient way to work with MongoDB collections. It offers repository abstractions and query methods tailored to MongoDB's document-based nature.
6. These data access modules in the Spring Framework provide a unified and consistent approach to working with various data sources, making it easier for developers to manage data interactions and switch between different data technologies seamlessly.

2 B. Explain about Spring web modules and Spring Test modules.

Spring Modules – Web



In the Spring Framework, web modules provide powerful capabilities for building web applications and RESTful web services. Here are the key web modules in the Spring Framework:

Spring Web (spring-web): It provides basic web-related functionalities, such as handling HTTP requests and responses, serving static resources, and managing servlet and filter lifecycles. This module includes the DispatcherServlet, which acts as the front controller in Spring web applications and handles incoming requests.

Spring Web MVC (spring-webmvc): Spring Web MVC is an extension of the Spring Web module and focuses on building Model-View-Controller (MVC) web applications. It provides a powerful and flexible MVC architecture that separates the application into model (data), view (UI), and controller (request processing). Developers can use annotations or XML configuration to define mappings between URLs and controller methods, enabling easy handling of HTTP requests and rendering responses.

Spring WebFlux (spring-webflux): Spring WebFlux is a reactive web framework introduced in Spring 5 that supports building reactive web applications. It is designed to handle high concurrency with a non-blocking, event-driven approach. Developers can choose between the traditional annotation-based programming model (with @Controller and @RestController) or the functional programming model using RouterFunctions.

Spring Web Services (spring-ws): Spring Web Services provides support for building SOAP-based web services. It simplifies the development of contract-first web services using XML schemas (XSD) to define the message formats. The module includes classes for handling XML marshaling and unmarshaling, as well as endpoints for processing SOAP requests.

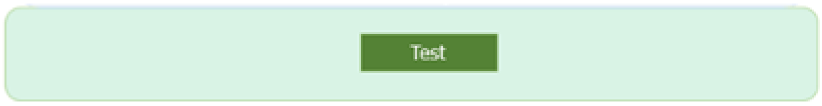
Spring Security (spring-security): Spring Security is a powerful module that provides comprehensive security features for web applications and RESTful services. It supports various authentication mechanisms, authorization, secure access control, and integration with various authentication providers like LDAP, OAuth, and more.

Spring WebSocket (spring-websocket): The Spring WebSocket module enables real-time, bidirectional communication between web clients and the server. It builds on top of the WebSocket protocol and provides support for handling WebSocket connections, sending and receiving messages, and managing WebSocket sessions.

Spring Web Services Client (spring-ws-support): This module offers support for building client-side components to interact with SOAP-based web services. It provides classes for marshaling and unmarshaling SOAP messages and creating client-side proxies for web service endpoints.

These web modules in the Spring Framework offer a wide range of features to build robust and scalable web applications and services. They follow the principles of dependency injection and inversion of control, allowing developers to create highly maintainable and testable web applications.

Spring Modules – Test modules



The Spring Test module, also known as the Spring TestContext Framework, is a crucial component of the Spring Framework. It provides powerful support for testing Spring applications, including integration testing, unit testing. The Spring Test module is included in the spring-test JAR.

Key features and components of the Spring Test module include:

Spring TestContext Framework: The core of the Spring Test module is the TestContext framework. It allows you to load and manage the Spring application context for testing. With this framework, you can create and configure application contexts, set up test data, and perform various assertions on the context and its components.

Spring Test Annotations: The module provides a set of annotations that simplify the configuration and execution of tests:

@RunWith(SpringRunner.class): This annotation is used at the class level to specify the test runner that integrates JUnit with the Spring TestContext Framework.

@ContextConfiguration: This annotation is used to specify the location of the Spring configuration files or Java configuration classes that define the test context.

@Test: Standard JUnit annotation to mark test methods.

@Autowired: This annotation is used to inject Spring beans into test classes, allowing you to use Spring's dependency injection in your tests.

@DirtiesContext: This annotation indicates that the Spring context should be dirtied (i.e., closed and reloaded) after the test, which is useful when the test modifies the context.

Mock Objects Support: Spring Test provides support for creating and working with mock objects using the @MockBean annotation. It allows you to easily mock dependencies during testing.

Transactional Support: With the @Transactional annotation, you can manage transactions during tests. This annotation ensures that the test methods run within a transaction, and any changes made during the test are rolled back at the end.

Testing Support for Web Applications: Spring Test offers support for testing web applications with the MockMvc and TestRestTemplate classes. They allow you to send HTTP requests to controllers and REST endpoints and perform assertions on the responses.

The Spring Test module is widely used to ensure the correctness and reliability of Spring applications. It enables developers to perform comprehensive testing of Spring components, including controllers, services, repositories, and more. By providing support for various testing scenarios and the ability to set up isolated and repeatable test environments, the Spring Test module promotes best practices for testing in Spring-based projects.

3. A. Explain about Inversion of Control technique in Spring Framework.

Usually it is the developer's responsibility to create the dependent application object using the new operator in an application. Hence any change in the application dependency requires code change and this results in more complexity as the application grows bigger.

Inversion of Control (IoC) helps in creating a more loosely coupled application. IoC represents the inversion of the responsibility of the application object's creation, initialization, and destruction from the application to the third party such as the framework. Now the third party takes care of application object management and dependencies thereby making an application easy to maintain, test, and reuse.

There are many approaches to implement IoC, Spring Framework provides IoC implementation using Dependency Injection(DI).

Introduction to Spring Inversion of Control(IoC)

Spring Container managed application objects are called beans in Spring.

We need not create objects in dependency injection instead describe how objects should be created through configuration.

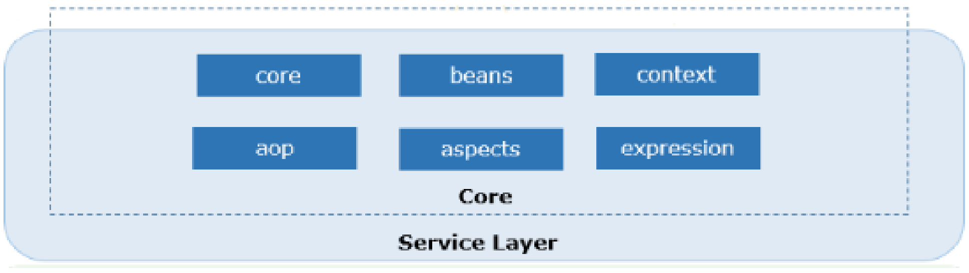
DI is a software design pattern that provides better software design to facilitate loose coupling, reuse, and ease of testing.

Benefits of Dependency Injection(DI):

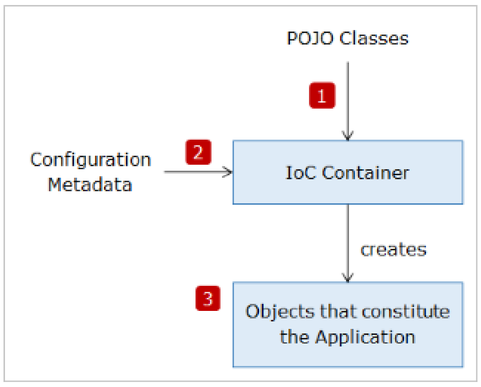
- Helps to create loosely coupled application architecture facilitating re-usability and easy testing.
- Separation of responsibility by keeping code and configuration separately. Hence dependencies can be easily modified using configuration without changing the code.
- Allows to replace actual objects with mock objects for testing, this improves testability by writing simple JUnit tests that use mock objects.

Spring IoC

The core container module of the Spring Framework provides IoC using Dependency Injection.



The Spring container knows which objects to create and when to create through the additional details that we provide in our application called **Configuration Metadata.**



1. Application logic is provided through POJO classes.
2. Configuration metadata consists of bean definitions that the container must manage.
3. IoC container produces objects required by the application using POJO classes and configuration metadata. IoC container is of two types – **BeanFactory and ApplicationContext**.

Spring IoC - Containers

Spring provides two types of containers.

BeanFactory:

- It is the basic Spring container with features to instantiate, configure and manage the beans.
- **org.springframework.beans.factory.BeanFactory** is the main interface representing a BeanFactory container.

ApplicationContext:

- ApplicationContext is another Spring container that is more commonly used in Spring applications.
- **org.springframework.context.ApplicationContext** is the main Interface representing an ApplicationContext container.
- It inherits the BeanFactory features and provides added features to support enterprise services such as internationalization, validation, etc.

ApplicationContext is the preferred container for Spring application development. Let us look at more details on the ApplicationContext container.

BeanFactory Vs ApplicationContext

BeanFactory	ApplicationContext
It does not support annotation based Dependency Injection.	Support annotation based Dependency Injection.
It does not support enterprise services.	Support enterprise services such as validations, internationalization, etc.
By default, it supports Lazy Loading.	By default, it supports Eager Loading. Beans are instantiated during load time.
//Loading BeanFactory BeanFactory factory = new	**// Loading ApplicationContext and instantiating bean**

AnnotationConfigApplicationContext(SpringConfiguration.class);
 	
// Instantiating bean during first access using getBean()
CustomerServiceImpl service = (CustomerServiceImpl) factory.getBean("customerService");
	ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);
	
	// Instantiating bean during first access using getBean()
	CustomerServiceImpl service = (CustomerServiceImpl) context.getBean("customerService");

ApplicationContext and AbstractApplicationContext

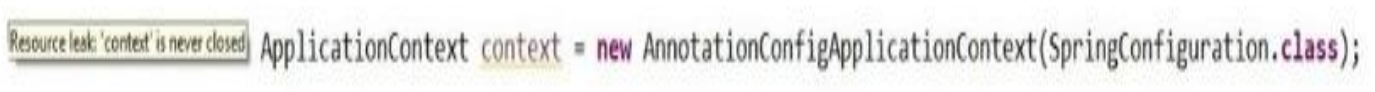
org.springframework.context.annotation.AnnotationConfigApplicationContext is one of the most commonly used implementation of ApplicationContext.

Example: ApplicationContext container instantiation.

```
1.ApplicationContext context = new
  AnnotationConfigApplicationContext(SpringConfiguration.class);
2.Object obj = context.getBean("customerService");
```

1. ApplicationContext container is instantiated by loading the configuration from the SpringConfiguration.class which is available in the utility package of the application.
2. Accessing the bean with id "customerService" from the container.

AbstractApplicationContext



```
Resource leak: 'context' is never closed  ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);
```

You can see a warning message as **Resource leak: 'context' is never closed** while using the ApplicationContext type. This is for the reason that you don't have a close method with BeanFactory or even ApplicationContext. AbstractApplicationContext is an abstract implementation of the ApplicationContext interface and it implements Closeable and AutoCloseable interfaces. To close the application context and destroy all beans in its abstractApplicationContext has a close method that closes this application context.

Access bean in Spring

There are different ways to access bean in Spring

1. The traditional way of accessing bean based on bean id with explicit typecast

```
CustomerServiceImpl service = (CustomerServiceImpl) context.getBean("customerService");
```

2. Accessing bean based on class type to avoid typecast if there is a unique bean of type in the container

```
CustomerServiceImpl service =  context.getBean(CustomerServiceImpl.class);
```

3. Accessing bean through bean id and also type to avoid explicit typecast

```
CustomerServiceImpl service = context.getBean("customerService", CustomerServiceImpl.class);
```

3 B. Explain how to configure the IoC Container using Java based Configuration.

In Spring Framework, configuration metadata is essential for defining how the various components in your application should be wired together. Spring provides multiple ways to configure the application, and each method uses configuration metadata in different formats.

Here are some common ways to provide configuration metadata in Spring:

1. **XML Configuration:** XML-based configuration is one of the earliest and most widely used ways to configure a Spring application. In this approach, you define the beans, their dependencies, and other configuration details in XML files.

Example of XML configuration:

```
<!-- beans.xml -->
<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.xsd">
    <bean id="myService" class="com.example.MyService"/>
    <bean id="myRepository" class="com.example.MyRepository">
        <property name="dataSource" ref="myDataSource"/>
    </bean>
    <bean id="myDataSource" class="com.example.MyDataSource"/>
</beans>
```

2. **Java-based Configuration (Annotations):** Instead of using XML, Spring also allows you to use Java-based configuration with annotations. Here, you use special annotations to define beans and their dependencies directly in Java classes.

Example of Java-based configuration:

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
    @Bean
    public MyRepository myRepository() {
        return new MyRepository(myDataSource());
    }
    @Bean
    public MyDataSource myDataSource() {
        return new MyDataSource();
    }
}
```

3. **Java-based Configuration (Java DSL):** In addition to annotations, Spring provides a Java DSL (Domain-Specific Language) to configure the application programmatically. This gives you more flexibility and control over the configuration process.

Example of Java-based configuration using Java DSL:

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
    @Bean
    public MyRepository myRepository() {
        return new MyRepository(myDataSource());
    }
    @Bean
    public MyDataSource myDataSource() {
        return new MyDataSource();
    }
}
```

4. **Property File Configuration:** Spring allows you to externalize configuration properties in property files. These properties can be loaded into the application context and used to configure beans or other parts of the application.

Example of property file configuration:

```
# application.properties
db.username=myuser
db.password=mypassword
```

AppConfig.java file:

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {
    @Value("${db.username}")
    private String dbUsername;
    @Value("${db.password}")
    private String dbPassword;
    // ... other bean definitions using the dbUsername and dbPassword
}
```

These are some of the common ways to provide configuration metadata in Spring. You can choose the approach that best fits your project's requirements and coding preferences. Additionally, Spring Boot offers even more streamlined and convenient ways to configure applications, building on top of these Spring configuration mechanisms.

4. A. Explain the process of injecting primitive values using Constructor Injection with an example.

Let us consider the CustomerService class of InfyTel Customer application to understand constructor injection.
CustomerService class has a count property, let us now modify this class to initialize count property during bean instantiation using the constructor injection approach.

```
package com.infy.service;
public class CustomerServiceImpl implements CustomerService {
private int count;
public CustomerServiceImpl(int count) {
this.count = count;
}
}
```

How do we define bean in the configuration to initialize values?

```
@Configuration
public class SpringConfiguration {
@Bean // CustomerService bean definition with bean dependencies through constructor
injection
public CustomerServiceImpl customerService() {
return new CustomerServiceImpl(20);
}
```

What is mandatory for constructor injection?
A parameterized constructor is required in the CustomerService class as we are injecting the values through the constructor argument.
Can we use constructor injection to initialize multiple properties?
Yes, we can initialize more than one property.

4 B. Explain the process of injecting non-primitive values using Constructor Injection with an example.

So far, we learned how to inject primitive values using constructor injection in Spring. Now we will look into inject object dependencies using Constructor injection. Consider our InfyTel Customer application. **CustomerServiceImpl.java** class which is dependent on CustomerRepository(class used to in persistence layer to perform CRUD operations) object type to call fetchCustomer() method.

```
package com.infy.service;
public class CustomerServiceImpl implements CustomerService {
```

```
// CustomerServiceImpl needs to contact CustomerRepository, hence injecting the
customerRepository dependency
private CustomerRepository customerRepository;
private int count;
public CustomerServiceImpl() {
}
public CustomerServiceImpl(CustomerRepository customerRepository, int count) {
this.customerRepository = customerRepository;
this.count=count;
}
public String fetchCustomer() {
return customerRepository.fetchCustomer(count);
}
public String createCustomer() {
return customerRepository.createCustomer();
}
}
```

Observe in the above code, CustomerRepository property of CustomerSevice class has not been initialized with any value in the code. This is because Spring dependency is going to be taken care of in the configuration.

How do we inject object dependencies through configuration using constructor injection?

```
package com.infy.util;
@Configuration
public class SpringConfiguration {
@Bean// customerRepository bean definition
public CustomerRepository customerRepository() {
return new CustomerRepository();
}
@Bean // CustomerServic bean definition with bean dependencies through constructor
injection
public CustomerServiceImpl customerService() {
return new CustomerServiceImpl(customerRepository(),20);
}
}
```

5. A. Explain the process of injecting primitive values using Setter Injection in Spring.

Let us now understand Setter Injection in Spring.

In Setter Injection, Spring invokes setter methods of a class to initialize the properties after invoking a default constructor.

How can we use setter injection to inject values for the primitive type of properties?

Consider the below example to understand setter injection for primitive types.

Following the CustomerServiceImpl class has a count property, let us see how to initialize this property during bean instantiation using the setter injection approach.

```
package com.infy.service;
public class CustomerServiceImpl implements CustomerService {
private int count;
public int getCount() {
return count;
}
public void setCount(int count) {
this.count = count;
}
public CustomerServiceImpl(){
}
}
```

How do we define bean in the configuration to initialize values?

```
package com.infy.util;
@Configuration
public class SpringConfiguration {
@Bean // CustomerService bean definition using Setter Injection
public CustomerServiceImpl customerService() {
CustomerServiceImpl customerService = new CustomerServiceImpl();
customerService.setCount(10);
return customerService;
}
}
```

What is mandatory to implement setter injection?

Default constructor and setter methods of respective dependent properties are required in the CustomerServiceImpl class. For setter injection, Spring internally uses the default constructor to create a bean and then invokes a setter method of the respective property based on the name attribute in order to initialize the values.

5 B. Explain the process of injecting non-primitive values using Setter Injection in Spring.

So far, we learned how to inject primitive values using setter injection in Spring.

How do we inject object dependencies using setter injection?

Consider the CustomerServiceImpl class of InfyTel Customer application.

```
package com.infy.service;
public class CustomerServiceImpl implements CustomerService {
private CustomerRepository customerRepository;
private int count;
public CustomerRepository getCustomerRepository() {
return customerRepository;
}
public void setCustomerRepository(CustomerRepository customerRepository) {
this.customerRepository = customerRepository;
}
public int getCount() {
return count;
}
public void setCount(int count) {
this.count = count;
}
}
```

How do we inject object dependencies through configuration using setter injection?

```
package com.infy.util;
@Configuration
public class SpringConfiguration {
@Bean
public CustomerRepository customerRepository() {
return new CustomerRepository();
}
@Bean // Setter Injection
public CustomerServiceImpl customerService() {
CustomerServiceImpl customerService = new CustomerServiceImpl();
customerService.setCount(10);
customerService.setCustomerRepository(customerRepository());
return customerService;
}
}
```

6. Explain about Auto Scanning with an example.

As discussed in the previous example as a developer you have to declare all the bean definition in SpringConfiguration class so that Spring container can detect and register your beans as below :

```
@Configuration
public class SpringConfiguration {
@Bean
public CustomerRepository customerRepository() {
return new CustomerRepository();
}
@Bean
public CustomerServiceImpl customerService() {
return new CustomerServiceImpl();
}
}
```

Is any other way to eliminate this tedious beans declaration?

Yes, Spring provides a way to automatically detect the beans to be injected and avoid even the bean definitions within the Spring configuration file through Auto Scanning. In Auto Scanning, Spring Framework automatically scans, detects, and instantiates the beans from the specified base package, if there is no declaration for the beans in the SpringConfiguration class.

So your SpringConfiguration class will be looking as below:

```
@Configuration
@ComponentScan(basePackages="com.infy")
public class SpringConfiguration {
}
```

Component scanning

Component scanning isn't turned on by default, however. You have to annotate the configuration class with **@ComponentScan** annotation to enable component scanning as follows:

```
@Configuration
@ComponentScan
public class SpringConfiguration  {
}
```

In the above code, Spring will scan the package that contains SpringConfig class and it subpackages for beans. But if you want to scan a different package or multiple packages then you can specify this with the basePackages attribute as follows:

```
@Configuration
@ComponentScan(basePackages = "com.infy.service,com.infy.repository")
public class SpringConfiguration {
}
```

Spring uses @ComponentScan annotation for the auto scan feature. It looks for classes with the stereotype annotations and creates beans for such classes automatically.

Spring bean stereotype annotations

- Stereotype annotations denote the roles of types or methods at the conceptual level.
- Stereotype annotations are @Component, @Service, @Repository, and @Controller annotations.
- These annotations are used for auto-detection of beans using @ComponentScan.
- The Spring stereotype @Component is the parent stereotype.
- The other stereotypes are the specialization of @Component annotation.

Annotation	Usage
@Component	It indicates the class(POJO class) as a Spring component.
@Service	It indicates the Service class(POJO class) in the business layer.
@Repository	It indicates the Repository class(POJO class in Spring DATA) in the persistence layer.
@Controller	It indicates the Controller class(POJO class in Spring MVC) in the presentation layer.

- @Component is a generic stereotype for any Spring-managed component.
- @Repository, @Service, and @Controller are specializations of @Component for more specific use cases.
- @Component should be used when your class does not fall into either of three categories i.e. Controllers, Services, and DAOs.

Configuring IoC container using Java Annotation-based configuration

@Component: It is a general purpose annotation to mark a class as a Spring-managed bean.

```
@Component
public class CustomerLogging{
//rest of the code
}
```

@Service - It is used to define a service layer Spring bean. It is a specialization of the @Component annotation for the service layer.

```
@Service
```

```
public class CustomerSeviceImpl implements CustomerService {
//rest of the code
}
```

@Repository - It is used to define a persistence layer Spring bean. It is a specialization of the @Component annotation for the persistence layer.

```
@Repository
public class CustomerRepositoryImpl implements CustomerRepository {
//rest of the code
}
```

@Controller - It is used to define a web component. It is a specialization of the @Component annotation for the presentation layer.

```
@Controller
public class CustomerController {
//rest of the code
}
```

By default, the bean names are derived from class names with a lowercase initial character. Therefore, your above defined beans have the names customerController, customerServiceImpl, and customerRepositoryImpl. It is also possible to give a specific name with a value attribute in those annotations as follows:

```
@Repository(value="customerRepository")
public class CustomerRepositoryImpl implements CustomerRepository {
//rest of the code
}
```

Note: As a best practice, use @Service for the service layer, @Controller for the Presentation layer, and @Repository for the Persistence layer.

UNIT-II

1. A. Explain the steps involved in creating the Spring Boot Application.

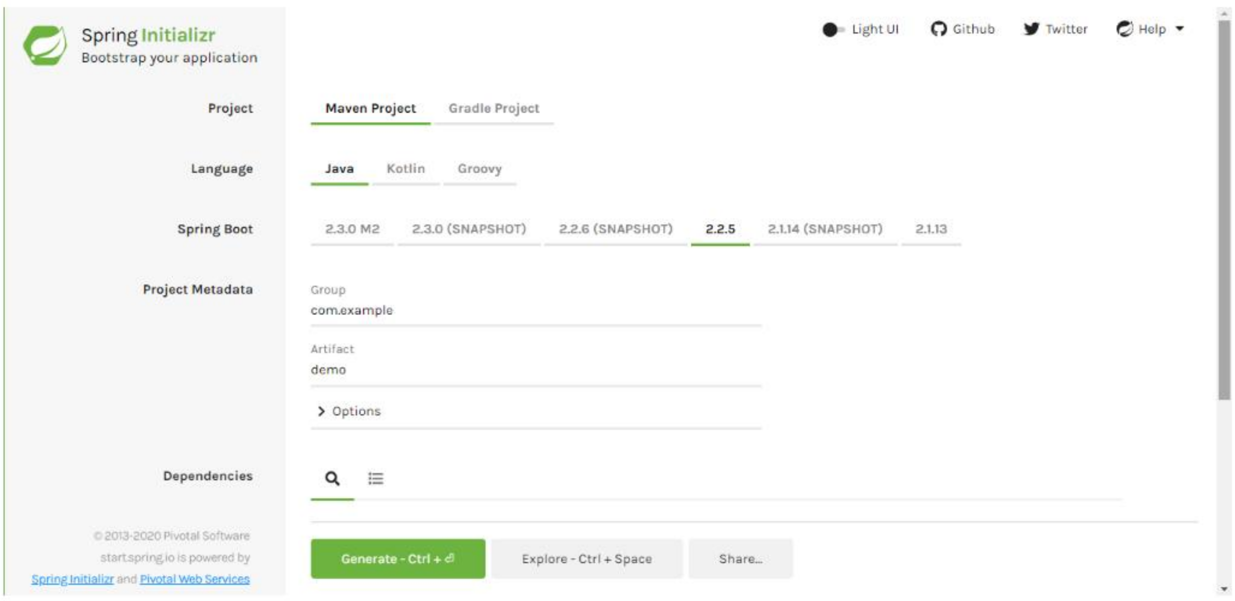
There are multiple approaches to create a Spring Boot application. You can use any of the following approaches to create the application:

- **Using Spring Boot CLI**
- **Using Spring Initializr**
- **Using the Spring Tool Suite (STS)**

Creating Spring Boot Application using Spring Initializr

It is an online tool provided by Spring for generating Spring Boot applications which is accessible at **http://start.spring.io/.** You can use it for creating a Spring Boot project using the following steps:

Step 1: Create your Spring Boot application launch Spring Initializr. You will get the following screen:


Spring Initializr
Bootstrap your application

Light UI · Github · Twitter · Help

Project: Maven Project | Gradle Project

Language: Java | Kotlin | Groovy

Spring Boot: 2.3.0 M2 | 2.3.0 (SNAPSHOT) | 2.2.6 (SNAPSHOT) | 2.2.5 | 2.1.14 (SNAPSHOT) | 2.1.13

Project Metadata
Group: com.example
Artifact: demo
> Options

Dependencies

Generate - Ctrl + ⏎ | Explore - Ctrl + Space | Share...

© 2013-2020 Pivotal Software
start.spring.io is powered by
Spring Initializr and Pivotal Web Services

Note: This screen keeps changing depending on updates from Pivotal and changes in the Spring Boot version.

Step 2: Select Project as Maven, Language as Java, and Spring Boot as 2.1.13 and enter the project details as follows:

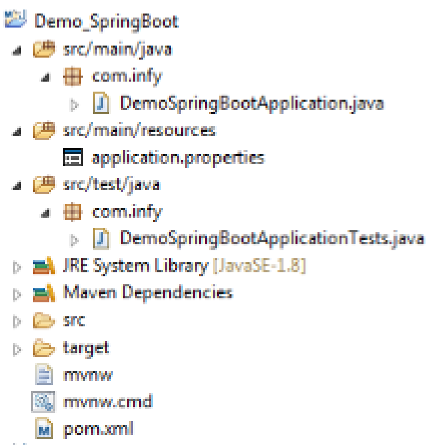
- Choose com.infy as Group
- Choose Demo_SpringBoot as Artifact

Click on More options and choose com.infy as Package Name

Step 3: Click on Generate Project. This would download a zip file to your local machine.

Step 4: Unzip the zip file and extract it to a folder.

Step 5: In Eclipse, Click File → Import → Existing Maven Project. Navigate or type in the path of the folder where you extracted the zip file to the next screen. After finishing, our Spring Boot project should look like as follows:


```
Demo_SpringBoot
  src/main/java
    com.infy
      DemoSpringBootApplication.java
  src/main/resources
    application.properties
  src/test/java
    com.infy
      DemoSpringBootApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

You have created a Spring Boot Maven-based project. Now let us explore what is contained in the generated project.

Understanding Spring Boot project structure

The generated project contains the following files:

1. pom.xml

This file contains information about the project and configuration details used by Maven to build the project.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.1.13.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.infy</groupId>
<artifactId>Demo_SpringBoot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>Demo_SpringBoot</name>
<description>Demo project for Spring Boot</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

2. application.properties

This file contains application-wide properties. To configure your application Spring reads the properties defined in this file. In this file, you can define a server's default port, the server's context path, database URLs, etc.

3. DemoSpringBootApplication.java

```
@SpringBootApplication
public class DemoSpringBootApplication {
public static void main(String[] args) {
SpringApplication.run(DemoSpringBootApplication.class, args);
}
}
```

It is annotated with @SpringBootApplication annotation which triggers auto-configuration and component scanning and can be used to declare one or more @Bean methods also. It contains the main method which bootstraps the application by calling the run() method on the SpringApplication class. The run method accepts DemoSpringBootApplication.class as a parameter to tell Spring Boot that this is the primary component.

4. DemoSpringBootApplicationTest.java
In this file test cases are written. This class is by default generated by Spring Boot to bootstrap Spring application.

1 B. Explain about Spring Boot Runners

So far you have learned how to create and start the Spring Boot application. Now suppose you want to perform some action immediately after the application has started then for this Spring Boot provides the following two interfaces:

- CommandLineRunner
- ApplicationRunner

CommandLineRunner is the Spring Boot interface with a run() method. Spring Boot automatically calls this method of all beans implementing this interface after the application context has been loaded. To use this interface, you can modify the DemoSpringBootApplication class as follows:

```
@SpringBootApplication
public class DemoSpringBootApplication implements CommandLineRunner {
public static void main(String[] args) {
SpringApplication.run(DemoSpringBootApplication.class, args);
}
@Override
public void run(String... args) throws Exception {
System.out.println("Welcome to CommandLineRunner");
}
}
```

2. A. Explain about i) Spring Boot Starters ii) @PropertySource annotation.

i. **Spring Boot Starters**
Spring Boot starters are pre-configured dependency descriptors with the most commonly used libraries that you can add to your application. So you don't need to search for compatible libraries and configure them manually. Spring Boot will ensure that the necessary libraries are added to the build. To use these starters, you have to add them to the pom.xml file. For example, to use spring-boot-starter following dependency needs to be added in pom.xml:

```
<dependency>
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter</artifactId>
</dependency>
```

Spring Boot comes with many starters. Some popular starters which we are going to use in this course are as follows:

- spring-boot-starter - This is the core starter that includes support for auto-configuration, logging, and YAML.
- spring-boot-starter-aop - This starter is used for aspect-oriented programming with Spring AOP and AspectJ.
- spring-boot-starter-data-jdbc - This starter is used for Spring Data JDBC.
- spring-boot-starter-data-jpa - This starter is used for Spring Data JPA with Hibernate.
- spring-boot-starter-web - This starter is used for building a web application using Spring MVC and Spring REST. It also provides Tomcat as the default embedded container.
- spring-boot-starter-test - This starter provides support for testing Spring Boot applications using libraries such as JUnit, Hamcrest, and Mockito.

ii. **@PropertySource annotation**

You can use other files to keep the properties. For example, InfyTelmessage.properties.

InfyTelmessage.properties

```
message=Welcome To InfyTel
```

But by default Spring Boot will load only the application.properties file. So how you will load the **InfyTelmessage.properties** file?

In Spring @PropertySource annotation is used to read from properties file using Spring's Environment interface. The location of the properties file is mentioned in the Spring configuration file using @PropertySource annotation.

So InfyTelmessage.properties which are present in classpath can be loaded using @PropertySource as follows:

```
import org.springframework.context.annotation.PropertySource;
@SpringBootApplication
@PropertySource("classpath:InfyTelmessage.properties")
public class DemoSpringBootApplication {
public static void main(String[] args) throws Exception {
//code
}
}
```

To read the properties you need to autowire the Environment class into a class where the property is required

```
@Autowired
Environment env;
```

You can read the property from Environment using the getProperty() method.

```
env.getProperty("message")
```

2 B. Explain about @Spring Boot Application Annotation with an example.

@SpringBootApplication

We have already learned that the class which is used to bootstrap the Spring Boot application is annotated with @SpringBootApplication annotation as follows:

```
@SpringBootApplication
public class DemoSpringBootApplication {
public static void main(String[] args) {
SpringApplication.run(DemoSpringBootApplication.class, args);
}
}
```

Now let us understand this annotation in detail.

The **@SpringBootApplication** annotation indicates that it is a configuration class and also triggers auto-configuration and component scanning. It is a combination of the following annotations with their default attributes:

@EnableAutoConfiguration – This annotation enables auto-configuration for the Spring boot application which automatically configures our application based on the dependencies that you have added.

@ComponentScan – This enables the Spring bean dependency injection feature by using @Autowired annotation. All application components which are annotated with @Component, @Service, @Repository, or @Controller are automatically registered as Spring Beans. These beans can be injected by using @Autowired annotation.

@Configuration – This enables Java based configurations for Spring boot application.

The class that is annotated with @SpringBootApplication will be considered as the main class, is also a bootstrap class. It kicks starts the application by invoking the SpringApplication.run() method. You need to pass the .class file name of your main class to the run() method.

SpringBootApplication- scanBasePackages

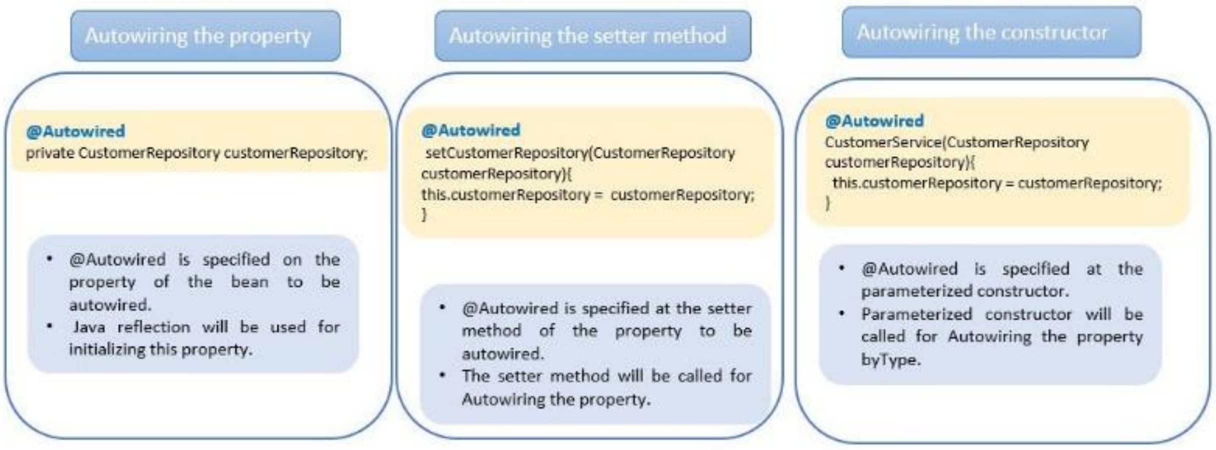
By default, SpringApplication scans the configuration class package and all it's sub-packages. So if our SpringBootApplication class is in "com.eta" package, then it won't scan com.infy.service or com.infy.repository package. We can fix this situation using the SpringBootApplication scanBasePackages property.

```
package com.eta;
@SpringBootApplication(scanBasePackages={"com.infy.service","com.infy.repository"})
public class DemoSpringBootApplication {
public static void main(String[] args) {
SpringApplication.run(DemoSpringBootApplication.class, args);
}
}
```

3. A. Briefly explain about autowiring with an example in Spring Framework.

In Spring if one bean class is dependent on another bean class then the bean dependencies need to be explicitly defined in your configuration class. But you can let the Spring IoC container to inject the dependencies into dependent bean classes without been defined in your configuration class. This is called as autowiring.

To do autowiring, you can use @Autowired annotation. This annotation allows the Spring IoC container to resolve and inject dependencies into your bean.@Autowired annotation performs **byType Autowiring** i.e. dependency is injected based on bean type. It can be applied to attributes, constructors, setter methods of a bean class.
Autowiring is done only for dependencies to other beans. It doesn't work for properties such as primitive data types, String, Enum, etc. For such properties, you can use the **@Value** annotation.



```
@Autowired
private CustomerRepository customerRepository;
```

- @Autowired is specified on the property of the bean to be autowired.
- Java reflection will be used for initializing this property.

```
@Autowired
setCustomerRepository(CustomerRepository
customerRepository){
this.customerRepository = customerRepository;
}
```

- @Autowired is specified at the setter method of the property to be autowired.
- The setter method will be called for Autowiring the property.

```
@Autowired
CustomerService(CustomerRepository
customerRepository){
 this.customerRepository = customerRepository;
}
```

- @Autowired is specified at the parameterized constructor.
- Parameterized constructor will be called for Autowiring the property byType.

Now let us see how to use @Autowired annotation.

@Autowired on Setter methods
The @Autowired annotation can be used on setter methods. This is called a Setter Injection.

```
package com.infy.service;
public class CustomerServiceImpl implements CustomerService {
private CustomerRepository customerRepository;
@Autowired
public void setCustomerRepository(CustomerRepository customerRepository) {
this.customerRepository = customerRepository;
}
--------
}
```

In the above code snippet, the Spring IoC container will call the setter method for injecting the dependency of CustomerRepository.

@Autowired on Constructor
The @Autowired annotation can also be used on the constructor. This is called a Constructor Injection.

```
package com.infy.service;
public class CustomerServiceImpl implements CustomerService {
private CustomerRepository customerRepository;
@Autowired
public CustomerServiceImpl(CustomerRepository customerRepository) {
this.customerRepository = customerRepository;
}
--------------
}
```

@Autowired on Properties
Let us now understand the usage of @Autowired on a property in Spring.

We will use @Autowired in the below code to wire the dependency of CustomerService class for CustomerRepository bean dependency.

```
package com.infy.service;
public class CustomerServiceImpl {
// CustomerService needs to contact CustomerRepository, hence injecting the customerRepository dependency
@Autowired
private CustomerRepository customerRepository;
------------
}
```

@Autowired is by default wire the dependency based on the type of bean.

In the above code, the Spring container will perform dependency injection using the Java Reflection API. It will search for the class which implements CustomerRepository and injects its object. The dependencies which are injected using @Autowired should be available to the Spring container when the dependent bean object is created. If the container does not find a bean for autowiring, it will throw the NoSuchBeanDefinitionException exception.

If more than one beans of the same type are available in the container, then the framework throws an exception indicating that more than one bean is available for autowiring. To handle this **@Qualifier** annotation is used as follows:

```
package com.infy.service;
public class CustomerServiceImpl {
@Autowired
@Qualifier("custRepo")
private CustomerRepository customerRepository;
------------
}
```

@Value Annotation

In Spring @Value annotation is used to insert values into variables and method arguments. Using @Value we can either read spring environment variables or system variables.

We can assign a default value to a class property with @Value annotation:

```
public class CustomerDTO {
@Value("1234567891")
long phoneNo;
@Value("Jack")
String name;
@Value("Jack@xyz.com")
String email;
@Value("ANZ")
String address;
}
```

Note that it accepts only a String argument but the passed-in value gets converted to an appropriate type during value-injection.

To read a Spring environment variable we can use @Value annotation:

```
public class CustomerDTO {
@Value("${value.phone}")
long phoneNo;
@Value("${value.name}")
String name;
@Value("${value.email}")
String email;
@Value("${value.address}")
String address;
}
```

3 B. Explain the Scope of a Bean in Spring Boot Application.

The lifetime of a bean depends on its scope. Bean's scope can be defined while declaring it in the configuration metadata file.
A bean can be in singleton or prototype scope. A bean with the "**singleton"** scope is initialized during the container starts up and the same bean instance is provided for every bean request from the application. However, in the case of the **"prototype"** scope, a new bean instance is created for every bean request from the application.
singleton scope There will be a single instance of "singleton" scope bean in the container and the same bean is given for every request from the application.

The bean scope can be defined for a bean using @Scope annotation in Java class. By default, the scope of a bean is the singleton

```
package com.infy.service;
@Service("customerService")
@Scope("singleton")
public class CustomerServiceImpl implements CustomerService {
@Value("10")
private int count;
public int getCount() {
return count;
}
public void setCount(int count) {
this.count = count;
}
public String fetchCustomer() {
return " The number of customers fetched are : " + count;
}
}
```

prototype scope

For the "prototype" bean, there will be a new bean created for every request from the application.
In the below example, customerService bean is defined with prototype scope. There will be a new customerService bean created for every bean request from the application.

```
package com.infy.service;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;
@Service("customerService")
```

```
@Scope("prototype")
public class CustomerServiceImpl implements CustomerService {
@Value("10")
private int count;
public int getCount() {
return count;
}
public void setCount(int count) {
this.count = count;
}
public String fetchCustomer() {
return " The number of customers fetched are : " + count;
}
}
```

4. Explain about various Logging Methods in Spring Framework environment.

Logging is the process of writing log messages to a central location during the execution of the program. That means Logging is the process of tracking the execution of a program, where

- Any event can be logged based on the interest to the
- When exception and error occurs we can record those relevant messages and those logs can be analyzed by the programmer later

There are multiple reasons why we may need to capture the application activity.

- Recording unusual circumstances or errors that may be happening in the program
- Getting the info about what's going in the application

There are several logging APIs to make logging easier. Some of the popular ones are:

- JDK Logging API
- Apache Log4j
- Commons Logging API

The Logger is the object which performs the logging in applications.

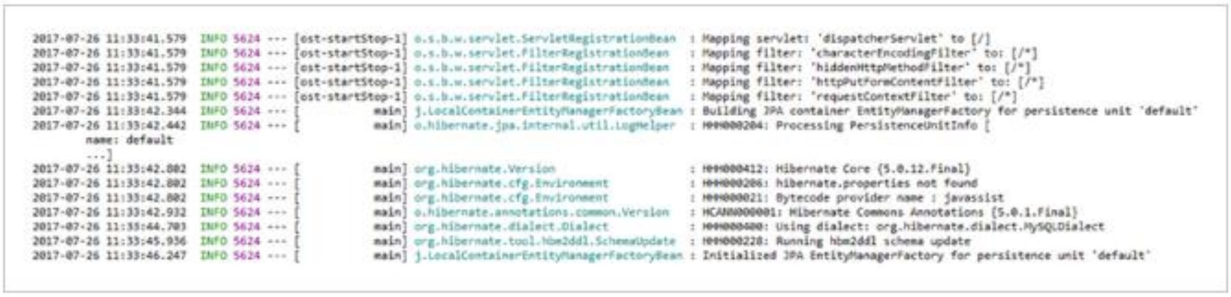
Levels of Logging

Levels in the logger specify the severity of an event to be logged. The logging level is decided based on necessity. For example, TRACE can be used during development and ERROR during deployment. The following table shows the different levels of logging.

Level	Description
ALL	For all the levels (including user defined levels)
TRACE	Informational events
DEBUG	Information that would be useful for debugging the application
INFO	Information that highlights the progress of an application
WARN	Potentially harmful situations
ERROR	Errors that would permit the application to continue running
FATAL	Severe errors that may abort the application
OFF	To disable all the levels

You know that logging is one of the important activities in any application. It helps in quick problem diagnosis, debugging, and maintenance. Let us learn the logging configuration in Spring Boot.

While executing your Spring Boot application, have you seen things like the below getting printed on your console?



```
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2017-07-26 11:33:42.344  INFO 5624 --- [           main] j.LocalContainerEntityManagerFactoryBean : Building JPA container EntityManagerFactory for persistence unit 'default'
2017-07-26 11:33:42.442  INFO 5624 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [
        name: default
        ...]
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate Core {5.0.12.Final}
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.cfg.Environment            : HHH000206: hibernate.properties not found
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.cfg.Environment            : HHH000021: Bytecode provider name : javassist
2017-07-26 11:33:42.932  INFO 5624 --- [           main] o.hibernate.annotations.common.Version   : HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
2017-07-26 11:33:44.703  INFO 5624 --- [           main] org.hibernate.dialect.Dialect            : HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
2017-07-26 11:33:45.936  INFO 5624 --- [           main] org.hibernate.tool.hbm2ddl.SchemaUpdate  : HHH000228: Running hbm2ddl schema update
2017-07-26 11:33:46.247  INFO 5624 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
```

Do you have any guess what are these?

Yes, you are right. These are logging messaged logged on the **INFO level.** However, you haven't written any code for logging in to your application. Then who does this?

By default, Spring Boot configures logging via **Logback** to log the activities of libraries that your application uses.

As a developer, you may want to log the information that helps in quick problem diagnosis, debugging, and maintenance. So, let us see how to customize the default logging configuration of Spring Boot so that your application can log the information that you are interested in and in your own format.

Log the error messages

Have you realized that you have not done any of the below activities for logging which you typically do in any Spring application?

- Adding dependent jars for logging
- Configuring logging through Java configuration or XML configuration

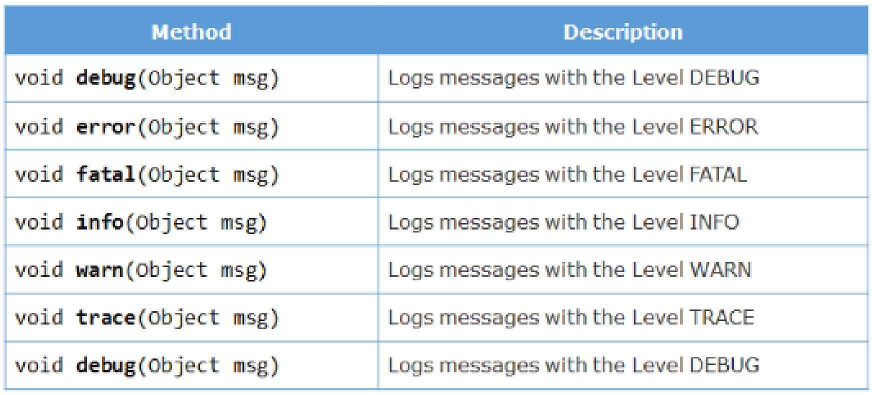
Still, you are able to log your messages. The reason is Spring Boot's default support for logging. The spring-boot-starter dependency includes spring-boot-starter-logging dependency, which configures logging via Logback to log to the console at the INFO level.

Spring Boot uses Commons Logging API with default configurations for Java Util Logging, Log4j 2, and Logback implementation. Among these implementations, Logback configuration will be enabled by default.

You, as a developer, have just created an object for Logger and raise a request to log with your own message in LoggingAspect.java as shown below.

```
public class CustomerServiceImpl implements CustomerService
{
private static Logger logger = LoggerFactory.getLogger(CustomerServiceImpl.class);
public void deleteCustomer(long phoneNumber) {
public void deleteCustomer(long phoneNumber) {
try {
customerRepository.deleteCustomer(phoneNumber);
} catch (Exception e) {
logger.info("In log Exception ");
logger.error(e.getMessage(),e);
}
}
}
```

Apart from info(), the Logger class provides other methods for logging information:

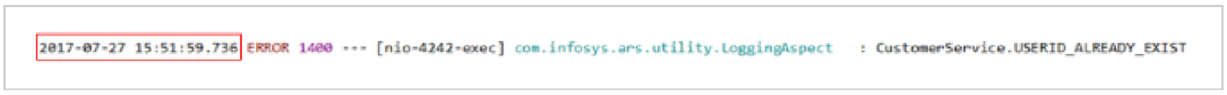


Method	Description
void **debug**(Object msg)	Logs messages with the Level DEBUG
void **error**(Object msg)	Logs messages with the Level ERROR
void **fatal**(Object msg)	Logs messages with the Level FATAL
void **info**(Object msg)	Logs messages with the Level INFO
void **warn**(Object msg)	Logs messages with the Level WARN
void **trace**(Object msg)	Logs messages with the Level TRACE
void **debug**(Object msg)	Logs messages with the Level DEBUG

Log the error messages using Logback

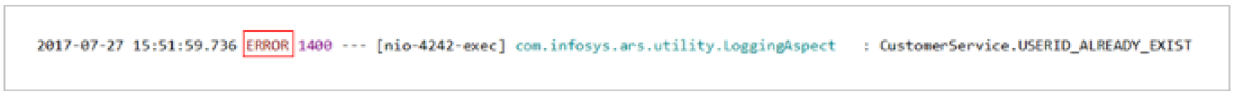
The default log output contains the following information.

Date and Time



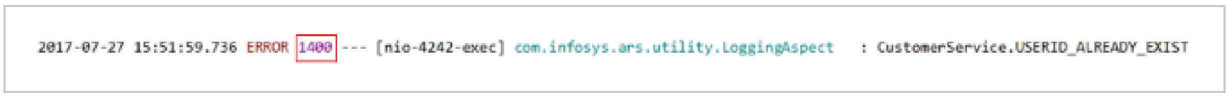
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Log level



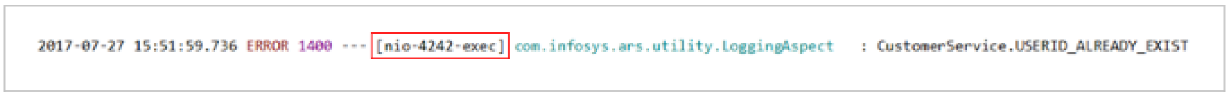
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Process id



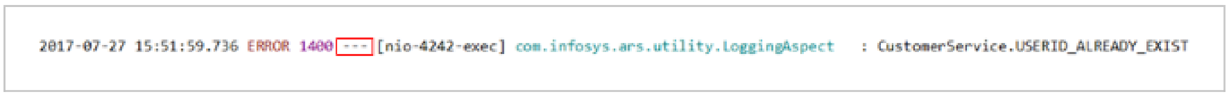
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Thread name



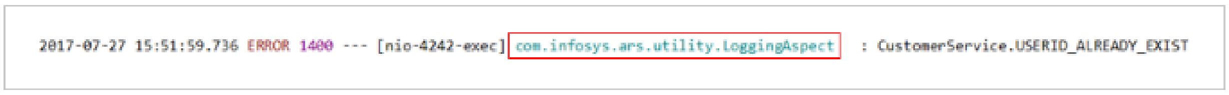
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Separator



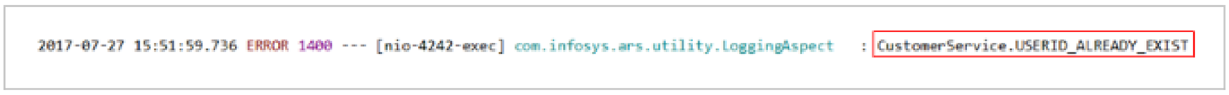
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Logger name



```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Log message



```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

But, how to change this default configuration if you want to,

- log the message in a file rather than console
- log the message in your own pattern
- log the messages of a specific level
- use Log4j instead of Logback

Log into file

By default Spring Boot logs the message on the console. To log into a file, you have to include either logging.file or logging. path property in your application.properties file.

Note: Please note that from Spring boot 2.3.X version onwards logging.file and logging.path has been deprecated we should use "logging.file.name" and " logging.file.path" for the same.

Custom log pattern

Include logging.pattern.* property in application.properties file to write the log message in your own format.

Logging property	**Sample value**	**Description**
logging.pattern.console	%d{yyyy-MM-dd HH:mm:ss,SSS}	Specifies the log pattern to use on the console
logging.pattern.file	%5p [%t] %c [%M] - %m%n	Specifies the log pattern to use in a file

Custom log level

By default, the logger is configured at the INFO level. You can change this by configuring the logging.level.* property in application.properties file as shown below.

```
logging.level.root=WARN
logging.level.com.infosys.ars=ERROR
```

Log4j instead of Logback

Since Spring Boot chooses Logback implementation by default, you need to exclude it and then include log4j 2 instead of in your pom.xml.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
<exclusions>
<exclusion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-logging</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-log4j2</artifactId>
```

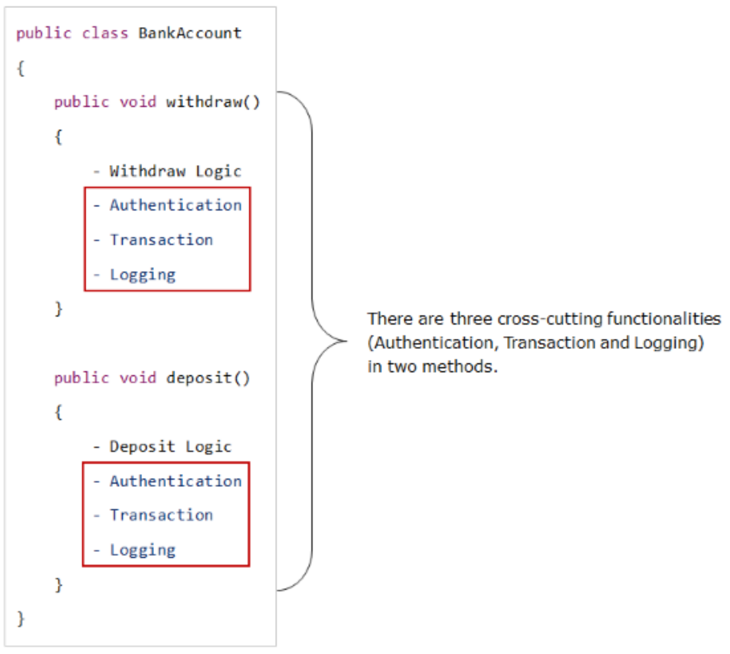
```
14. </dependency>
```

5. A. Explain about Aspect Oriented Programming (AOP) in Spring Boot.

AOP (Aspect Oriented Programming) is used for applying common behaviors like transactions, security, logging, etc. to the application.
These common behaviors generally need to be called from multiple locations in an application. Hence, they are also called as cross cutting concerns in AOP.
Spring AOP provides the solution to cross cutting concerns in a modularized and loosely coupled way.
Advantages

- AOP ensures that cross cutting concerns are kept separate from the core business logic.
- Based on the configurations provided, the Spring applies cross cutting concerns appropriately during the program execution.
- This allows creating a more loosely coupled application wherein you can change the cross cutting concerns code without affecting the business code.
- In Object Oriented Programming(OOP), the key unit of modularity is class. But in AOP the key unit of modularity is an Aspect.

What is an Aspect?
Aspects are the cross-cutting concerns that cut across multiple classes.
Examples: Transaction Management, Logging, Security, etc.
For a better understanding of Aspect Oriented Programming(AOP) concepts let us consider a Banking scenario comprising of BankAccount class with Withdraw and Deposit functionalities as shown below.

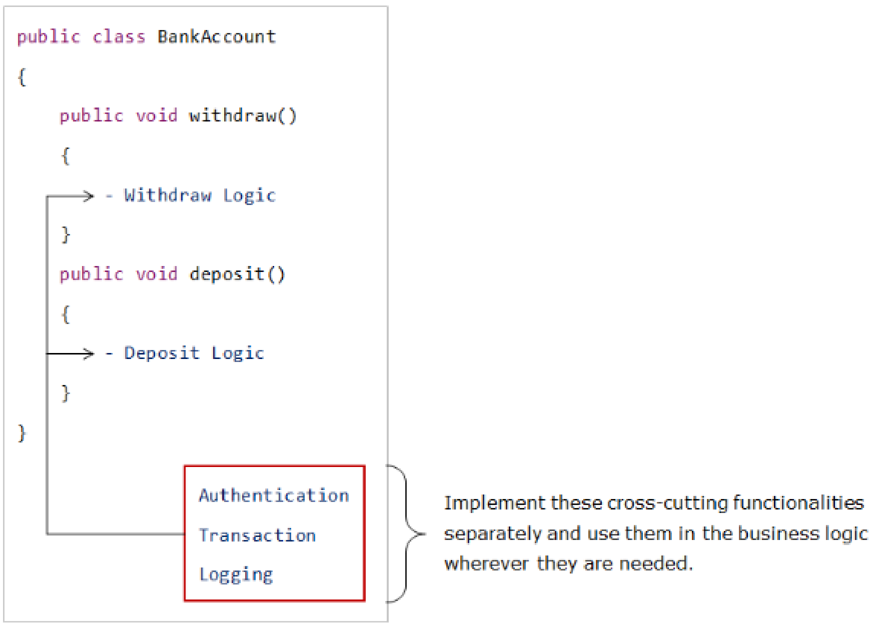


```
public class BankAccount
{
    public void withdraw()
    {
        - Withdraw Logic
        - Authentication
        - Transaction
        - Logging
    }

    public void deposit()
    {
        - Deposit Logic
        - Authentication
        - Transaction
        - Logging
    }
}
```

There are three cross-cutting functionalities (Authentication, Transaction and Logging) in two methods.

In a single class, cross-cutting functionalities are repeated twice. Think of a bigger scenario with many classes. You might need to repeat the cross-cutting concerns many times.

In Spring AOP, we can add these cross-cutting concerns at run time by separating the cross-cutting concerns from the client logic as shown below.


Implement these cross-cutting functionalities separately and use them in the business logic wherever they are needed.

```
public class BankAccount
{
    public void withdraw()
    {
      - Withdraw Logic
    }
    public void deposit()
    {
      - Deposit Logic
    }
}
            Authentication
            Transaction
            Logging
```

In Spring AOP, we can add the cross-cutting functionalities at run time by separating the system services (cross-cutting functionalities) from the client logic.

- Aspect is a class that implements cross-cutting concerns. To declare a class as an Aspect it should be annotated with @Aspect annotation. It should be applied to the class which is annotated with @Component annotation or with derivatives of it.
- Joinpoint is the specific point in the application such as method execution, exception handling, changing object variable values, etc. In Spring AOP a join point is always the execution of a method.
- Advice is a method of the aspect class that provides the implementation for the cross-cutting concern. It gets executed at the selected join point(s). The following table shows the different types of advice along with the execution point they have

Type Of Execution	**Execution Point**
Before	Before advice is executed before the Joinpoint execution.
After	After advice will be executed after the execution of Joinpoint whether it returns with or without exception. Similar to finally block in exception handling.
AfterReturning	AfterReturning advice is executed after a Joinpoint executes and returns successfully without exceptions
AfterThrowing	AfterThrowing advice is executed only when a Joinpoint exits by throwing an exception
Around	Around advice is executed around the Joinpoints which means Around advice has some logic which gets executed before Joinpoint invocation and some logic which gets executed after the Joinpoint returns successfully

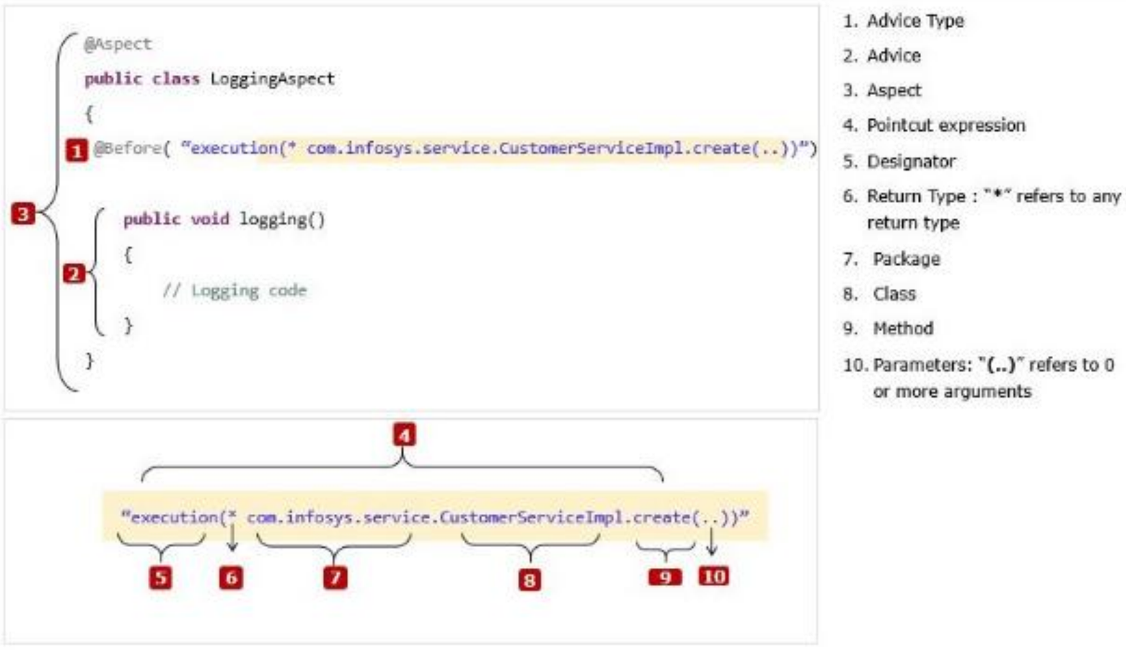
- Pointcut represents an expression that evaluates the method name before or after which the advice needs to be executed.
- In Spring AOP, we need to modularize and define each of the cross cutting concerns in a single class called Aspect.
- Each method of the Aspect which provides the implementation for the cross cutting concern is called Advice.
- The business methods of the program before or after which the advice can be called is known as a Joinpoint.
- The advice does not get inserted at every Joinpoint in the program.
- An Advice gets applied only to the Joinpoints that satisfy the Pointcut defined for the advice.

- Pointcut represents an expression that evaluates the business method name before or after which the advice needs to be called.

5 B. Explain pointcut declaration in Spring AOP.

Spring AOP - Pointcut declaration

Consider an Aspect "LoggingAspect" as shown below to understand different Spring AOP terminologies. LoggingAspect is defined as Spring AOP Aspect by using @Aspect annotation.



```
@Aspect
public class LoggingAspect
{
@Before( "execution(* com.infosys.service.CustomerServiceImpl.create(..))")

    public void logging()
    {
        // Logging code
    }
}
```

```
"execution(* com.infosys.service.CustomerServiceImpl.create(..))"
```

Pointcut declaration

A pointcut is an important part of AOP. So let us look at pointcut in detail.
Pointcut expressions have the following syntax:

```
execution(<modifiers> <return-type> <fully qualified class name>.<method-name>(parameters))
```

where,

- execution is called a pointcut designator. It tells Spring that joinpoint is the execution of the matching method.
- <modifiers> determines the access specifier of the matching method. It is not mandatory and if not specified defaults to the public.
- <return-type> determines the return type of the method in order for a join point to be matched. It is mandatory. If the return type doesn't matter wildcard * is used.
- <fully qualified class name> specifies the fully qualified name of the class which has methods on the execution of which advice gets executed. It is optional. You can also use * wildcard as a name or part of a name.
- <method-name> specifies the name of the method on the execution of which advice gets executed. It is mandatory. You can also use * wildcard as a name or part of a name.
- parameters are used for matching parameters. To skip parameter filtering, two dots(..) are used in place of parameters.

Pointcut	**Description**
execution(public * *(..))	execution of any public methods
execution(* service *(..))	execution of any method with a name

	beginning with "service"
execution(*com.infy.service.CustomerServiceImpl.*(..))	execution of any method defined in CustomerServiceImpl of com.infy.service package
execution(* com.infy.service.*.*(..))	execution of any method defined in the com.infy.service package
execution(public * com.infy.service.CustomerServiceImpl.*(..))	execution of any public method of CustomerServiceImpl of com.infy.service package
execution(public String com.infy.service.CustomerserviceImpl.*(..))	execution of all public method of CustomerServiceImpl of com.infy.service package that returns a String

6. A. Implement various Spring AOP advices using Spring Boot.

Configuring AOP in Spring Boot

To use Spring AOP and AspectJ in the Spring Boot project you have to add the spring-boot-starter-aop starter in the pom.xml file as follows:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Implementing AOP advices:

Before Advice:
This advice is declared using @Before annotation. It is invoked before the actual method call. ie. This advice is executed before the execution of fetchCustomer()methods of classes present in com.infy.service package. The following is an example of this advice:

```
@Before("execution(* com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
public void logBeforeAdvice(JoinPoint joinPoint) {
logger.info("In Before Advice, Joinpoint signature :{}", joinPoint.getSignature());
long time = System.currentTimeMillis();
String date = DateFormat.getDateTimeInstance().format(time);
logger.info("Report generated at time:{}", date);
}
```

After Advice:
This advice is declared using @After annotation. It is executed after the execution of the actual method(fetchCustomer), even if it throws an exception during execution. It is commonly used for resource cleanup such as temporary files or closing database connections. The following is an example of this advice :

```
@After("execution(* com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
public void logAfterAdvice(JoinPoint joinPoint) {
logger.info("In After Advice, Joinpoint signature :{}", joinPoint.getSignature());
long time = System.currentTimeMillis();
String date = DateFormat.getDateTimeInstance().format(time);
logger.info("Report generated at time {}", date);
}
```

After Returning Advice

This advice is declared using @AfterReturning annotation. It gets executed after joinpoint finishes its execution. If the target method throws an exception the advice is not executed. The following is an example of this advice that is executed after the method execution of fetchCustomer()method of classes present in com.infy.service package.

```
@AfterReturning(pointcut = "execution(*
com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
public void logDetails(JoinPoint joinPoint) {
logger.info("In AfterReturning Advice, Joinpoint signature :{}", joinPoint.getSignature());
}
```

You can also access the value returned by the joinpoint by defining the returning attribute of @AfterReturning annotation as follows:

```
@AfterReturning(pointcut = "execution(*
com.infy.service.CustomerServiceImpl.fetchCustomer(..))", returning = "result")
public void logDetails(JoinPoint joinPoint, String result) {
logger.info("In AfterReturning Advice with return value, Joinpoint signature :{}",
joinPoint.getSignature());
logger.info(result.toString());
}
```

In the above code snippet, the value of the returning attribute is returnvalue which matches the name of the advice method argument.

AfterThrowing Advice :

This advice is defined using @AfterThrowing annotation. It gets executed after an exception is thrown from the target method. The following is an example of this advice that gets executed when exceptions are thrown from the fetchCustomer() method of classes present in com.infy.service package. So it is marked with @AfterThrowing annotation as follows:

```
@AfterThrowing("execution(* com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
public void logAfterThrowingAdvice(JoinPoint joinPoint) {
logger.info("In After throwing Advice, Joinpoint signature :{}", joinPoint.getSignature());
}
```

You can also access the exception thrown from the target method inside the advice method as follows:

```
@AfterThrowing(pointcut ="execution(*
com.infy.service.CustomerServiceImpl.fetchCustomer(..))",throwing = "exception")
public void logAfterThrowingAdvice(JoinPoint joinPoint,Exception exception) {
logger.info("In After throwing Advice, Joinpoint signature :{}", joinPoint.getSignature());
logger.info(exception.getMessage());
}
```

Around advice:

This advice gets executed around the joinpoint i.e. before and after the execution of the target method. It is declared using @Around annotation. The following is an example of this advice:

```
@Around("execution(* com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {
System.out.println("Before proceeding part of the Around advice.");
Object cust =  joinPoint.proceed();
System.out.println("After proceeding part of the Around advice.");
return cust;
}
```

In the above code snippet, aroundAdvice method accepts an instance of ProceedingJoinPoint as a parameter. It extends the JoinPoint interface, and it can only be used in the Around advice. The proceed() method invokes the joinpoint.

6 B. Explain Briefly which best practices you should follow when designing the Spring Boot application.

Let us discuss the best practices which need to be followed as part of the Quality and Security for the Spring application and Spring with Spring Boot applications. These practices, when applied during designing and developing a Spring/Spring Boot application, yields better performance.

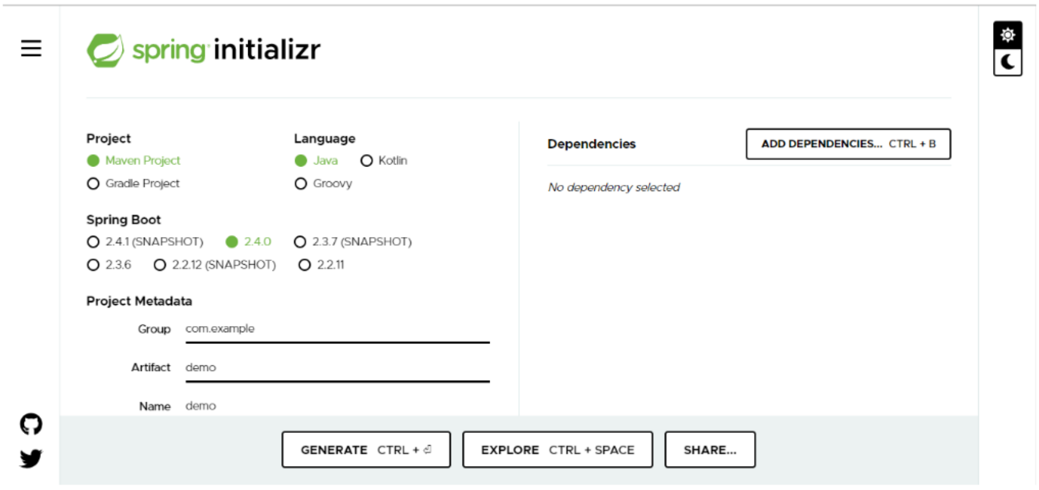
Best Practices:
1. To create a new spring boot project prefer to use Spring Initializr
2. While creating the Spring boot projects must follow the Standard Project Structure
3. Use @Autowired annotation before a constructor.
4. Use constructor injection for mandatory dependencies and Setter injection for optional dependencies in Spring /Spring boot applications
5. Inside the domain class avoid using the stereotype annotations for the automatic creation of Spring bean.
6. To create a stateless bean use the singleton scope and for a stateful bean choose the prototype scope.

Let us understand the reason behind these recommendations and their implications.

Use Spring Initializr for starting new Spring Boot projects

There are three different ways to create a Spring Boot project. They are:
- Using Spring Initializr
- Using the Spring Tool Suite (STS)
- Using Spring Boot CLI

But the recommended and simplest way to create a Spring Boot application is the Spring Boot Initializr as it has a very good UI to download a production-ready project. And the same project can be directly imported into the STS/Eclipse.



Project: [✓ Maven Project] Gradle Project

Language: [✓ Java] Kotlin Groovy

Spring Boot: 2.4.1 (SNAPSHOT) [✓ 2.4.0] 2.3.7 (SNAPSHOT) 2.3.6 2.2.12 (SNAPSHOT) 2.2.11

Project Metadata
Group com.example
Artifact demo
Name demo

Dependencies
No dependency selected

Note: The above screen keeps changing depending on updates from Pivotal and changes in the Spring Boot version.

Standard Project Structure for Spring Boot Projects

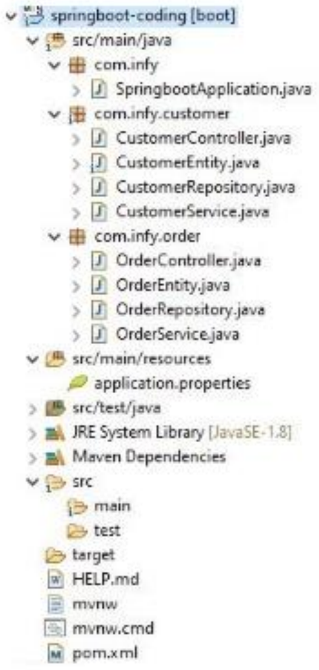
There are two recommended ways to create a spring boot project structure for Spring boot applications, which will help developers to maintain a standard coding structure.

Don't use the "default" Package:

When a developer doesn't include a package declaration for a class, it will be in the "default package". So the usage of the "default package" should be avoided as it may cause problems for Spring Boot applications that generally use the annotations like @ComponentScan, or @SpringBootApplication. So, during the creation of the classes, the developer should follow Java's package naming conventions. For example, com.infy.client, com.infy.service, com.infy.controller etc.

There are 2 approaches that can be followed to create a standard project structure in Spring boot applications.

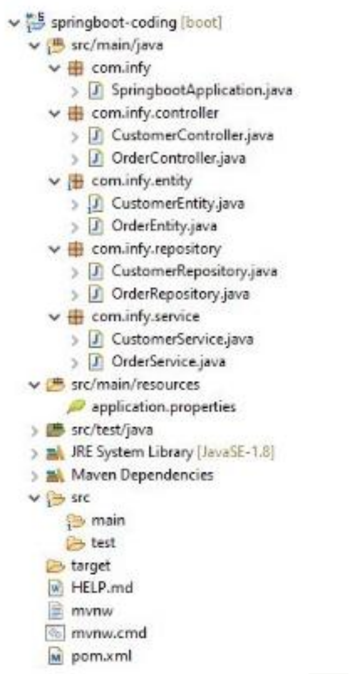
First approach: The first approach shows a layout which generally recommended by the Spring Boot team. In this approach we can see that all the related classes for the customer have grouped in the **"com.infy.cutomer"** package and all the related classes for the order have grouped in the **"com.infy.order"** package



```
springboot-coding [boot]
  src/main/java
    com.infy
      SpringbootApplication.java
    com.infy.customer
      CustomerController.java
      CustomerEntity.java
      CustomerRepository.java
      CustomerService.java
    com.infy.order
      OrderController.java
      OrderEntity.java
      OrderRepository.java
      OrderService.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
    main
    test
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Second approach: However the above structure works well but developers prefer to use the following structure.

In this approach, we can see that all the service classes related to customer and Order are grouped in the **"com.infy.service"** package. Similar to that we can see we grouped the repository, controller, and entity classes.



```
springboot-coding [boot]
  src/main/java
    com.infy
      SpringbootApplication.java
    com.infy.controller
      CustomerController.java
      OrderController.java
    com.infy.entity
      CustomerEntity.java
      OrderEntity.java
    com.infy.repository
      CustomerRepository.java
      OrderRepository.java
    com.infy.service
      CustomerService.java
      OrderService.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
    main
    test
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Best Practices: Spring Application

- **Inside the domain class, try to avoid using the stereotype annotations for the automatic creation of Spring bean.**

```
@Component
public class Employee {
// Methods and variables
}
```

Avoid creating beans for the domain class like the above.

- **Avoid scanning unnecessary classes to discover Beans. Specify only the required class for scanning.**

Suppose that if we need to discover beans declared inside the service package. Let us write the code for that.

```
@Configuration
@ComponentScan("com.infosys")
public class AppConfig {
}
```

In the above configuration class, we mentioned scanning the entire package com.infosys. But our actual requirement is only needed to scan the service packages. We can avoid this unnecessary scanning by replacing the code as below.

```
@Configuration
@ComponentScan("com.infosys.service")
public class AppConfig {
}
```

- **Java doesn't allow annotations placed on interfaces to be inherited by the implemented class so make sure that we place the spring annotations only on class, fields, or methods.**
- **As a good practice place @Autowired annotation before a constructor.**

We know that there are three places we can place @Autowired annotation in Spring on fields, on setter method, and on a constructor. The classes using field injection are difficult to maintain and it is very difficult to test. The classes using setter injection will make testing easy, but this has some disadvantages like it will violate encapsulation. Spring recommends that use @Autowired on constructors that are easiest to test, and the dependencies can be immutable.

- **In the case of AOP as a best practice store all the Pointcuts in a common class which will help in maintaining the pointcuts in one place.**

```
public class CommonPointConfig {
@Pointcut("execution(* com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
public void logAfterAdvice(JoinPoint joinPoint){}
@Pointcut("execution(* com.infy.service.CustomerServiceImpl.fetchCustomer(..))")
public void logBeforeAdvice(){}
}
```

The above common definition can be used when defining pointcuts in other aspects.

```
@Around("com.infy.service.CustomerServiceImpl.fetchCustomer.aspect.CommonPointConfig
.logBeforeAdvice()")
```

- **While defining the scope of the beans choose wisely.**

If we want to create a stateless bean then singleton scope is the best choice. In case if you need a stateful bean then choose prototype scope.

- **When to choose Constructor-based DI and setter-based DI in Spring /Spring boot applications**

A Spring application developer can mix the Constructor injection and Setter injection in the same application but it's a good practice to use constructor injection for mandatory dependencies and Setter injection for optional dependencies.

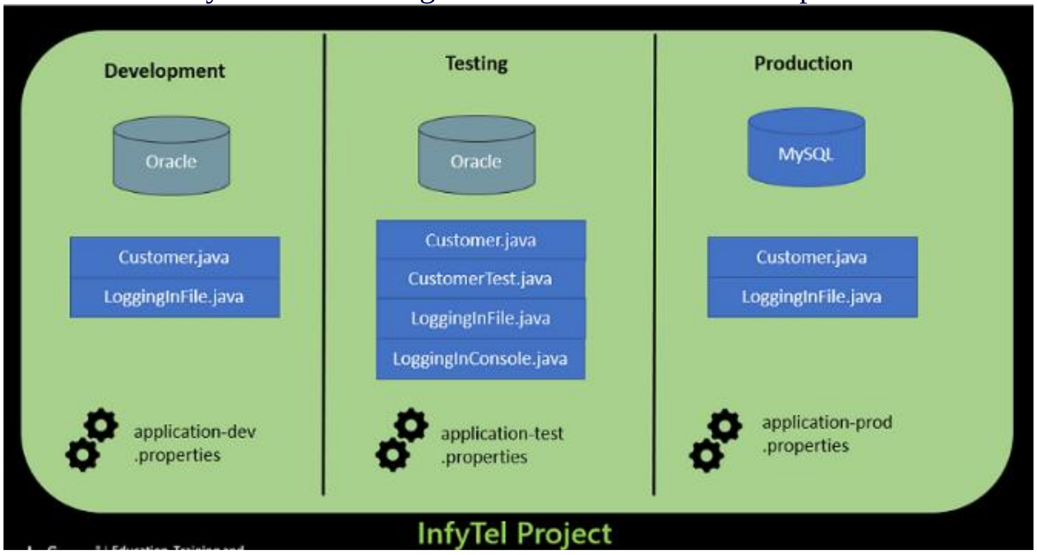
UNIT-III (PART -A)

1. A. Explain the concept of Spring Profiles in Spring Data JPA.

Analyze the below requirements:
Requirement1: InfyTel a telecom application has to use Oracle database in development and MySQL database in production environment. So, developer has two different application.properties file according to the configuration needed in development and production.

How will spring read the appropriate application.properties files depending on the environment?

Requirement2: The project requires logging levels and logging properties (i.e., files/console output) to be set differently in testing environment and production environment.



The above two requirements can be addressed easily using **Spring Profiles.**

What is Spring Profile?

Spring Profiles helps to classify the classes and properties file for the environment. You can create multiple profiles and set one or more profiles as the active profile. Based on the active profile spring framework chooses beans and properties file to run.

Let us see how to configure different profiles using spring profiles in our projects.

Steps to be followed:

1) Identify the beans which has to be part of a particular profile [Not mandatory]
2) Create environment-based properties file
3) Set active profiles.

You will see each step in-detail.

How to declare Spring Profile?

You can identify the Spring beans which have to be part of a profile using the following ways

1)Annotation @Profile

2) Bean declaration in XML

Note: In this course, the annotation based approach is covered.

@Profile

@Profile helps spring to identify the beans that belong to a particular environment.

1. Any class which is annotated with stereotype annotations such as @Component,@Service,@Repository and @Configuration can be annotated with @Profile .
2. @Profile is applied at class level except for the classes annotated with @Configuration where @Profile is applied at the method level.

@Profile- Class level:

```
@Profile("dev")
@Component
@Aspect
public class LoggingAspect { }
```

This LoggingAspect class will run only if “dev” environment is active.

@Profile- Method level:

```
@Configuration
public class SpringConfiguration {
@Bean("customerService")
@Profile("dev")
public CustomerService customerServiceDev() {
CustomerService customerServiceDev= new CustomerService();
customerServiceDev.setName("Developement-Customer");
return customerServiceDev;
}
@Bean("customerService")
@Profile("prod")
public CustomerService customerServiceProd() {
CustomerService customerServiceProd=new CustomerService();
customerServiceProd.setName("Production-Customer");
return customerServiceProd;
}
}
```

In the above code snippet CustomerService bean is configured differently in the "dev" and "prod" environments. Depending on the currently active profile Spring fetches CustomerService accordingly.

Note: @Profile annotation in class level should not be overridden in the method level. It will cause "NoSuchBeanDefinitionException".

@Profile value can be prefixed with !(Not) operator.

```
@Profile("!test")
@Configuration
@ComponentScan(basePackages="com.infy.service")
public class SpringConfiguration { }
```

This SpringConfiguration class will run in all environments other than test.

Note:
if you do not apply @profile annotation on a class, means that the particular bean is available in all environment(s).

Creating environment-based properties files for Spring Boot projects

Spring Boot relies on application.properties file for configuration. To write configuration based on environment, you need to create different application.properties files. The file name has a naming convention as application-<user created profile name>.properties

Example:

application-dev.properties

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1522:devDB
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver.class=oracle.jdbc.driver.OracleDriver
# logging
logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%
logging.level.org.springframework.web: ERROR
```

application-test.properties

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1522:testDB
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver.class=oracle.jdbc.driver.OracleDriver
# logging
logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n
```

```
9.  logging.level.org.springframework.web: DEBUG
```

application-prod.properties

```
1.  # mysql settings
2.  spring.datasource.url=jdbc:mysql://localhost:3306/prodDB?useSSL=false
3.  spring.datasource.username=root
4.  spring.datasource.password=root
5.  spring.datasource.driver.class=com.mysql.jdbc.Driver
6.  # logging
7.  logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} -
    %msg%
8.  logging.level.org.springframework.web: INFO
```

Note: The above shows how application.properties file with different configuration is created for 3 different environments(dev,test and prod)

1 B. Explain the procedure of setting Active Profiles in Spring Data JPA.

By now you are aware of how to map a bean or configure properties to a certain profile(eg :dev/prod/test), next we need to set the one which is active..
The different way to do is as follows:

1) application.properties
2) JVM System Parameter
3) Maven Profile

Note :You can make more than 1 profile as active at a time.

Set active profile - application.properties

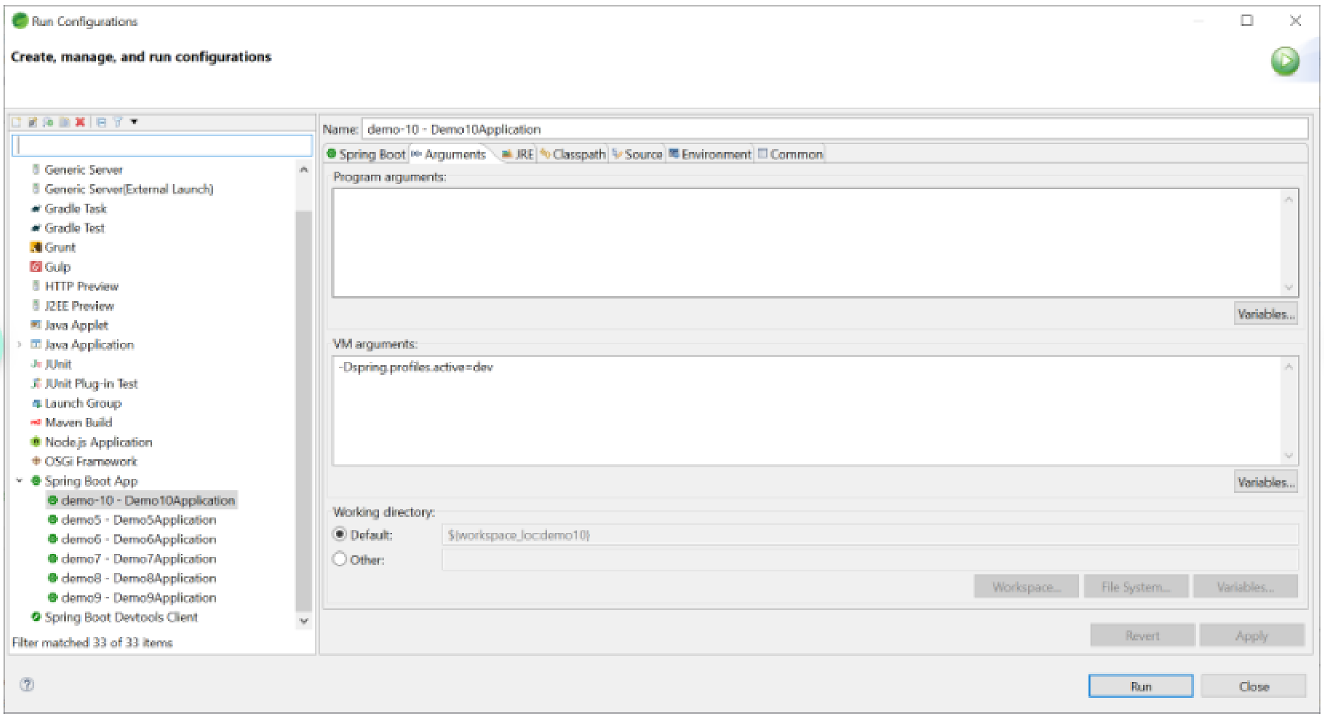
To set an active profile you need to write the below property in the main application.properties file.

```
spring.profiles.active=dev

        OR

spring.profiles.active=dev,prod
```

Set active profile - JVM System Parameter

You can set profile using JVM system arguments in two ways , either set the VM arguments in Run configuration or programmatic configuration.
To set through Run configuration give the below command in VM arguments.

```
-Dspring.profiles.active=dev
```


Run Configurations

Create, manage, and run configurations

Generic Server
Generic Server(External Launch)
Gradle Task
Gradle Test
Grunt
Gulp
HTTP Preview
J2EE Preview
Java Applet
Java Application
JUnit
JUnit Plug-in Test
Launch Group
Maven Build
Node.js Application
OSGi Framework
Spring Boot App
demo-10 - Demo10Application
demo5 - Demo5Application
demo6 - Demo6Application
demo7 - Demo7Application
demo8 - Demo8Application
demo9 - Demo9Application
Spring Boot Devtools Client

Filter matched 33 of 33 items

Name: demo-10 - Demo10Application

Program arguments:

VM arguments:
-Dspring.profiles.active=dev

Working directory:
Default: ${workspace_loc:demo10}
Other:

To set through programmatic approach set system property as follows,

```
public static void main( String[] args )    {
System.setProperty(AbstractEnvironment.ACTIVE_PROFILES_PROPERTY_NAME,
"prod");}
```

Set active profile - Maven

To set a profile using maven, follow the below steps.

Step 1: Add <profiles> tag to pom.xml file as follows.

```
<profiles>
<profile>
<id>dev</id>
<properties>
<spring.profiles.active>dev</spring.profiles.active>
</properties>
</profile>
<profile>
<id>test</id>
<activation>
<activeByDefault>true</activeByDefault>
</activation>
<properties>
<spring.profiles.active>test</spring.profiles.active>
</properties>
</profile>
</profiles>
<build>
<resources>
```

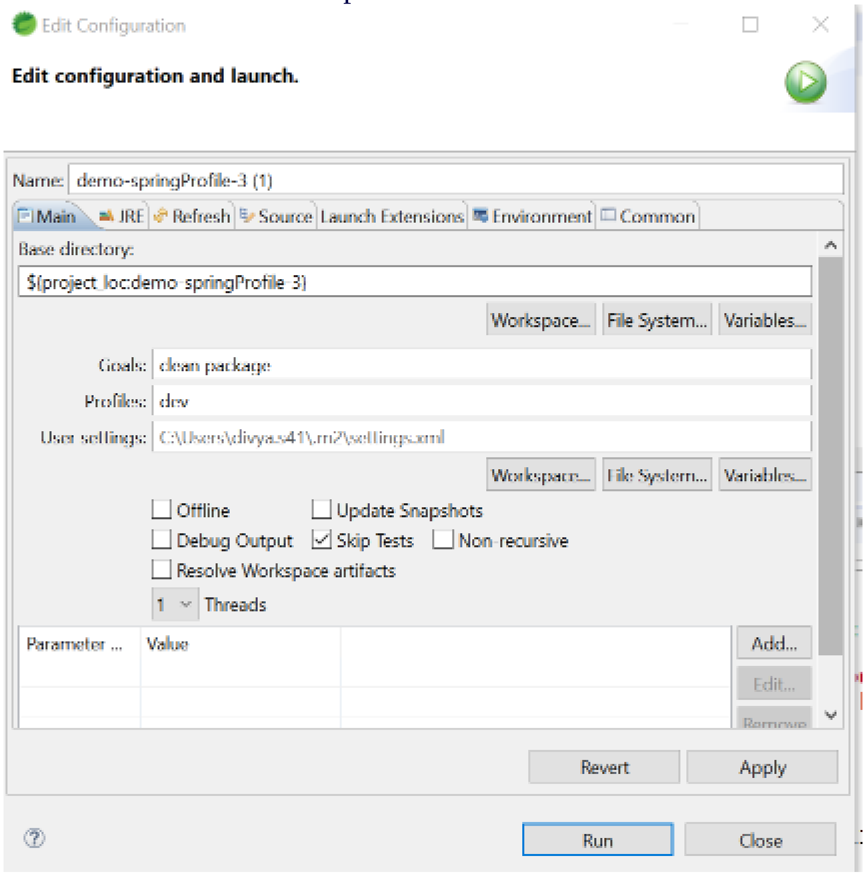
```
<resource>
<directory>src/main/resources</directory>
<filtering>true</filtering>
</resource>
</resources>
</build>
```

Step 2: Add below to the application.properties file.

```
spring.profiles.active=@spring.profiles.active@
```

Step 3: Build the project as guided below,

i. Run As -> Maven build
ii. Goals as clean package
iii. Profiles as dev
iv. Check Skip Tests


Edit Configuration

Edit configuration and launch.

Name: demo-springProfile-3 (1)

Main | JRE | Refresh | Source | Launch Extensions | Environment | Common

Base directory:
${project_loc:demo-springProfile-3}

Goals: clean package
Profiles: dev
User settings: C:\Users\divya.s41\.m2\settings.xml

☐ Offline ☐ Update Snapshots
☐ Debug Output ☑ Skip Tests ☐ Non-recursive
☐ Resolve Workspace artifacts
1 Threads

Parameter ...	Value

2. A. Write about the limitations of JDBC API.

InfyTel Application's data access layer has been implemented using JDBC API. But there are some limitations of using JDBC API.

As seen in the demo when JDBC API is used for database operation, the developer needs to provide a lot of data access code before and after the execution of a query. This increases the code size and it becomes difficult to maintain and make changes.

All the exceptions thrown in JDBC have checked exceptions that require try-catch blocks in the code. This also makes the code difficult to read and maintain.

If the database connection is not closed in an application, it will lead to resource leakage.

Let us understand the limitations of JDBC API

- A developer needs to open and close the connection.
- A developer has to create, prepare, and execute the statement and also maintain the resultset.
- A developer needs to specify the SQL statement(s), prepare, and execute the statement.
- A developer has to set up a loop for iterating through the result (if any).
- A developer has to take care of exceptions and handle transactions.

The above limitations can be solved with the technologies Spring ORM.

So Let's discuss more on ORM and also let's see some classes and interfaces of **Spring ORM** which will help to overcome the above limitations. Later we'll look at some features of Spring ORM which unnecessarily increase our repository implementations and will see how **Spring Data JPA** will help us to completely remove the repository class implementations by using Spring ORM specifications internally.

2 B. Explain about Object Relational Mapping (ORM).

To understand Spring Data JPA we need to know about ORM and Spring ORM

As seen in the previous demo of the InfyTel application where the persistence layer has been implemented using JDBC API, there are several limitations and challenges. Let us now look at them.

JDBC, I/O, Serialization do not solve the problem of data persistence effectively. For a medium to be effective, it needs to take care of the fundamental difference in the way Object-Oriented Programs(OOP) and RDBMS deals with the data.

- In Programming languages like Java, the related information or the data will be persisted in the form of hierarchical and interrelated objects.
- In the relational database, the data is persisted as table format or relations.

The greatest challenge in integrating the concepts of RDBMS and OOP is a mapping of the Java objects to databases. When object and relational paradigms work with each other, a lot of technical and conceptual difficulties arise, as mapping of an object to a table may not be possible in all the contexts. Storing and retrieving Java objects using a Relational database exposes a paradigm mismatch called "Object-Relational Impedance Mismatch". These differences are because of perception, style, and patterns involved in both the paradigms that lead to the following paradigm mismatches:

- **Granularity**: Mismatch between the number of classes in the object model and the number of tables in the relational model.
- **Inheritance or Subtype**: Inheritance is an object-oriented paradigm that is not available in RDBMS.
- **Associations**: In object-oriented programming, the association is represented using reference variables, whereas, in the relational model foreign keys are used for associating two tables.
- **Identity**: In Java, object equality is determined by the "==" operator or "equals()" method, whereas in RDBMS, uses the primary key to uniquely identify the records.
- **Data Navigation**: In Java, the dot(.) operator is used to travel through the object network, whereas, in RDBMS join operation is used to move between related records.

What is ORM?
Object Relational Mapping (ORM) is a technique or design pattern, which maps object models with the relational model. It has the following features:

- It resolves the object-relational impedance mismatch by mapping
    - Java classes to tables in the database
    - Instance variables to columns

    - Objects to rows in the table
- It helps the developer to get rid of SQL queries. They can concentrate on the business logic and work with the object model which leads to faster development of the application.
- It is database independent. All database vendors provide support for ORM. Hence, the application becomes portable without worrying about the underlying database.

Benefits of Object Relational Mapping(ORM)

- ORM provides a programmatic approach to implement database operations.
- ORM maps Java objects to the relational database tables in an easier way based on simple configuration.
- Supports simple query approaches like HQL(Hibernate Query language) and JPQL (Java Persistence Query Language)
- Supports object-oriented concepts such as inheritance, mapping, etc.

To use ORM in Java applications Java Persistence API (JPA) specification is used. There are several implementations of JPA available in the market, such as Hibernate, OpenJPA, DataNucleus, EclipseLink, etc. EclipseLink is the reference implementation for JPA.

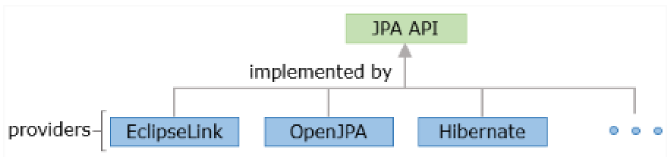
Note: Hibernate is one of the most popular implementations and in this course, our Spring Data JPA applications internally use this implementation.

ORM Providers

The Java Persistence API (JPA) is a Java EE specification that defines how data persistence-related tasks are handled using object-relational mapping (ORM) frameworks in Java applications. It provides the following features:

- Defines an API for mapping the object model with the relational model
- Defines an API for performing CRUD operations
- Standardizes ORM features and functionalities in Java.
- Provides an object query language called Java Persistence Query Language (JPQL) for querying the database.
- Provides Criteria API to fetch data over an object graph.

There are multiple providers available in the market which provides an implementation of JPA specification such as EclipseLink, OpenJPA, Hibernate, etc. as shown below:



Hibernate is the most widely used framework among these.
Once you have understood about ORM and its providers let's look at Spring ORM in brief.
Spring ORM:

Spring Framework is the most popular open-source Java application framework which supports building all types of Java applications like web applications, database-driven applications, batch applications, and many more. Spring framework's features such as Dependency Injection and Aspect-Oriented Programming help in developing a simple, easily testable, reusable, and maintainable application.

Spring is organized in a modular fashion. Developers can pick and choose the modules as per their needs. Spring ORM is a module under the Spring umbrella that covers many persistence technologies, namely JPA, JDO, Hibernate, and iBatis, etc. For each technology, the configuration basically consists of injecting a DataSource bean into SessionFactory or EntityManagerFactory and helps in performing CRUD operations.

Repository implementation using Spring ORM:

Spring ORM provides integration classes to integrate the application with ORM solutions such as JPA, Hibernate, MyBatis to perform database operations and transaction management smoothly. Hibernate specific exceptions need not be declared or caught as @Repository annotation. Spring enables the exception of clean translation.

The repository implementation class requires appropriate persistence resource bean from Spring Framework as below:

- JPA based repository uses EntityManagerFactory bean
- Hibernate based repository uses SessionFactory bean

JPA based repositories are the most convenient way of developing our repository layer of Spring ORM applications as JPA is a specification and Hibernate is an implementation. So with little modification in the future, we can migrate from one ORM implementation to another.

Developing the persistence layer using Spring ORM JPA possess the following advantages:

- Provides a programmatic approach to implement database operations.
- Maps Java objects to the relational database tables with help of entity classes based on simple configuration.
- Supports HQL(Hibernate Query language) and JPQL (Java Persistence Query Language)
- Supports object-oriented concepts such as inheritance, mapping, etc.

So let's see additional configurations required for developing the repository layer of a Spring ORM JPA application with Spring Boot.

3. A. Explain about Spring Data JPA with Spring Boot.

Introduction to Spring Boot

We have learned that Spring is a lightweight framework for developing enterprise applications. But using Spring for application development is challenging for developer because of the following reason which reduces productivity and increases the development time:

1. Configuration

Developing a Spring application requires a lot of configuration. This configuration also needs to be overridden for different environments like production, development, testing, etc. For example, the database used by the testing team may be different from the one used by the development team. So we have to spend a lot of time writing configuration instead of writing application logic for solving business problems.

2. Project Dependency Management

When you develop a Spring application you have to search for all compatible dependencies for the Spring version that you are using and then manually configure them. If the wrong version of dependencies is selected, then it will be an uphill task to solve this problem. Also for every new feature added to the application, the appropriate dependency needs to be identified and added. All this reduces productivity.

So to handle all these kinds of challenges Spring Boot came in the market.

What is Spring Boot?
Spring Boot is a framework built on the top Spring framework that helps developers build Spring-based applications quickly and easily. The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.

But how does it work? It works because of the following reasons,

1. Spring Boot is an opinionated framework
Spring Boot forms opinions. It means that Spring Boot has some sensible defaults which you can use to quickly build your application. For example, Spring Boot uses embedded Tomcat as the default web container.

2. Spring Boot is customizable
Though Spring Boot has its defaults, you can easily customize it at any time during your development based on your needs. For example, if you prefer log4j for logging over Spring Boot built-in logging support then you can easily make dependency change in your pom.xml file to replace the default logger with log4j dependencies.

The main Spring Boot features are as follows:
1. Starter Dependencies
2. Automatic Configuration
3. Spring Boot Actuator
4. Easy-to-use Embedded Servlet Container Support

Creating a Spring Boot Application

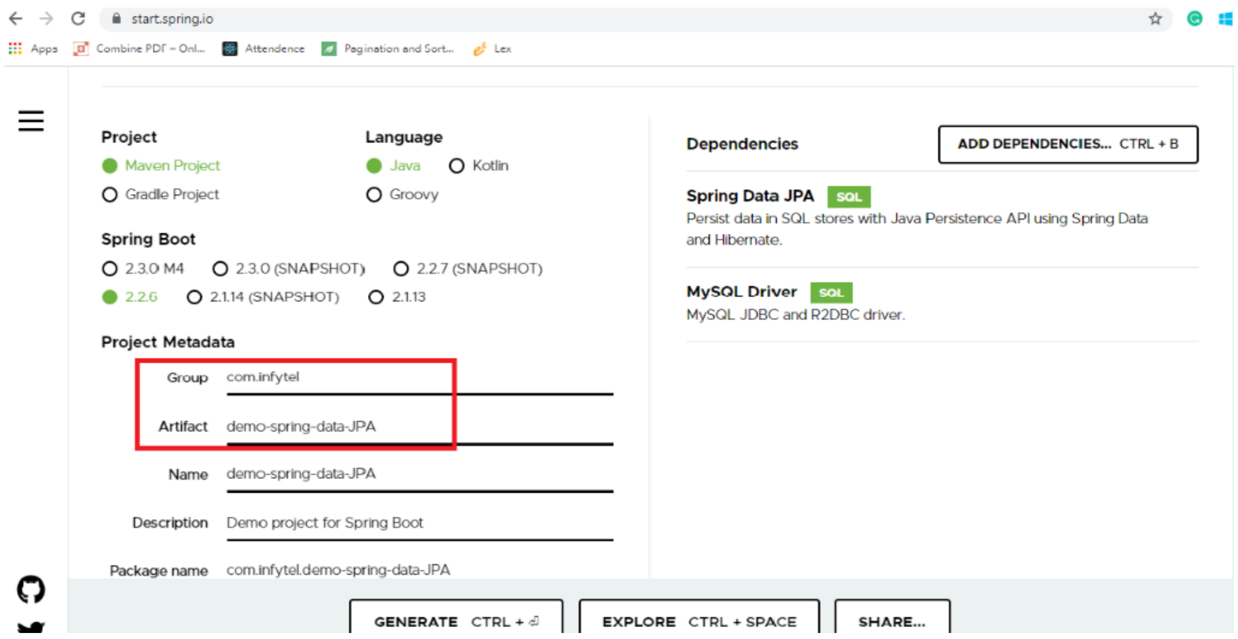
There are multiple approaches to create a Spring Boot application. You can use any of the following approaches to create the application:
- Using Spring Initializr
- Using the Spring Tool Suite (STS)
- Using Spring Boot CLI

In this course, you will learn how to use Spring Initializr for creating Spring Data JPA applications.

Spring Initializr:
It is an online tool provided by Spring for generating Spring Boot applications which is accessible at http://start.spring.io. You can use it for creating a Spring Boot project using the following steps:

Step 1: Launch Spring Initializr to create your Spring Data Boot application and do the below.

Group and Artifact under Project Metadata stand for package name and project name respectively. Give them any valid value according to your project, retain other default values (Project, Language & Spring Boot version). Add dependencies required to run your Spring Boot application. In this particular case, you need to add two dependencies namely Spring Data JPA & MySQL Driver(InfyTel application uses MySQL DB)


start.spring.io

Apps | Combine PDF – Onl... | Attendence | Pagination and Sort... | Lex

Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.3.0 M4 ○ 2.3.0 (SNAPSHOT) ○ 2.2.7 (SNAPSHOT)
● 2.2.6 ○ 2.1.14 (SNAPSHOT) ○ 2.1.13

Project Metadata

Group: com.infytel
Artifact: demo-spring-data-JPA
Name: demo-spring-data-JPA
Description: Demo project for Spring Boot
Package name: com.infytel.demo-spring-data-JPA

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

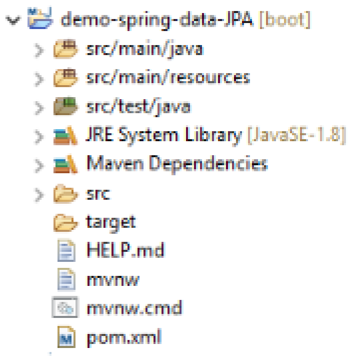
GENERATE CTRL + ⏎ | EXPLORE CTRL + SPACE | SHARE...

Note: This screen keeps changing depending on updates from Pivotal and change in the Spring Boot version.

Step 2: Select Project as Maven, Language as Java, and Spring Boot as 2.2.6 and the necessary dependencies for SpringORM as shown in the image, then enter the project details as follows:
Choose com.infytel as Group
Choose demo-spring-data-JPA as Artifact
Click on More options and choose com.infytel as package Name
Step 3: Click on Generate Project. This would download a zip file to the local machine.

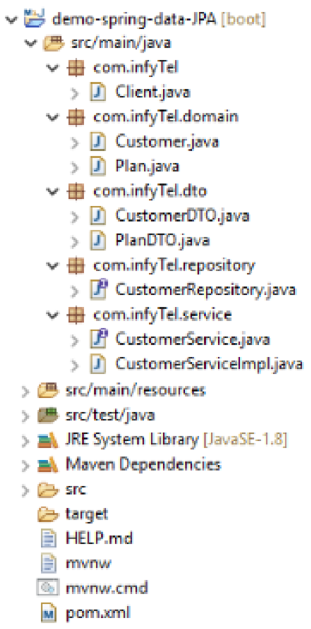
Step 4: Unzip the zip file and extract to a folder.

Step 5: In Eclipse/ STS, Click File → Import → Existing Maven Project. Navigate or type in the path of the folder where you extracted the zip file to the next screen. After finishing, our Spring Boot project should look like as follows:



```
demo-spring-data-JPA [boot]
  src/main/java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Add the respective files in the project as below:



```
demo-spring-data-JPA [boot]
  src/main/java
    com.infyTel
      Client.java
    com.infyTel.domain
      Customer.java
      Plan.java
    com.infyTel.dto
      CustomerDTO.java
      PlanDTO.java
    com.infyTel.repository
      CustomerRepository.java
    com.infyTel.service
      CustomerService.java
      CustomerServiceImpl.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

We have successfully created a Spring Boot Maven-based project with Spring Data JPA dependencies.

Spring Data JPA- Project Components

The generated project contains the following files:

1. **pom.xml:** This file contains information about the project and configuration details used by Maven to build the project.
2. **Spring Boot Starter Parent and Spring Boot starters**: Defines key versions of dependencies and combine all the related dependencies under a single dependency.
3. **application.properties**: This file contains application-wide properties. Spring reads the properties defined in this file to configure a Spring Boot application. A developer can define a server's default port, server's context path, database URLs, etc, in this file.
4. **ClientApplication**: Main application with @SpringBootApplication and code to interact with the end user.

3 B. Explain about the dependency concept with a POM.xml in Spring Boot.

Dependencies

1. pom.xml :

This file contains information about the project and configuration/dependency details used by Maven to build the Spring Data JPA project.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.6.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
```

```xml
<groupId>com.infytel</groupId>
<artifactId>demo-spring-data-JPA</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo-spring-data-JPA</name>
<description>Demo project for Spring Data JPA with Spring Boot</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Spring Boot Starter Parent and Spring Boot starters

Let us see more about the content of pom.xml

2. Spring Boot Starter Parent:

The Spring Boot Starter Parent defines key versions of dependencies and default plugins for quickly building Spring Boot applications. It is present in pom.xml file of application as a parent as follows:

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.6.RELEASE</version>
<relativePath/>
</parent>
```

It allows you to manage the following things for multiple child projects and modules:

- Configuration – The Java version and other properties.
- Dependencies – The version of dependencies.

Default Plugins Configuration – This includes default configuration for Maven plugins such as maven-failsafe-plugin, maven-jar-plugin, maven-surefire-plugin, maven-war-plugin.

Spring Boot starters:
Now let us discuss other starters of Spring Boot.
Spring Boot starters are pre-configured dependency descriptors with the most commonly used libraries that you can add in your application. So you don't need to search for compatible libraries and configure them manually. Spring Boot will ensure that the necessary libraries are added to the build. To use these starters, you have to add them to the pom.xml file. For example, to use spring-boot-starter following dependency needs to be added in pom.xml.

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

The starters combine all the related dependencies under a single dependency. It is a one-stop-shop for all required Spring related technologies without browsing through the net and downloading them individually.

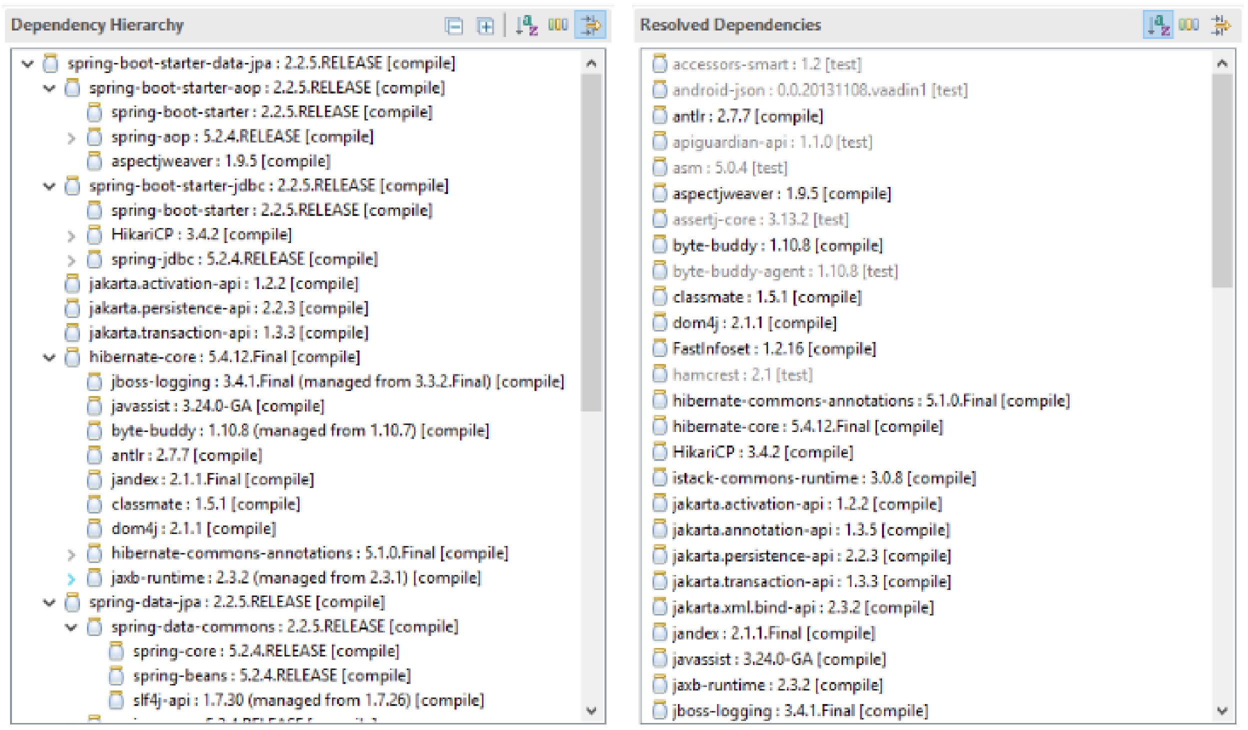
Spring Boot comes with many starters. Some popular starters are as follows:

- spring-boot-starter - This is the core starter that includes support for auto-configuration, logging, and YAML.
- spring-boot-starter-aop - This starter is used for aspect-oriented programming with Spring AOP and AspectJ.
- spring-boot-starter-data-jdbc - This starter is used for Spring Data JDBC.
- spring-boot-starter-data-jpa - This starter is used for Spring Data JPA with Hibernate.
- spring-boot-starter-web - This starter is used for building a web application using Spring MVC and Spring REST. It also provides Tomcat as the default embedded container.
- spring-boot-starter-test - This starter provides support for testing Spring Boot applications using libraries such as JUnit, Hamcrest, and Mockito.

One of the starters of Spring Boot is spring-boot-starter-data-jpa. It informs Spring Boot that it is a Spring Data JPA application.

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

Note that the version number of spring-boot-starter is not defined here as it can be determined by spring-boot-starter-parent. By opening the Dependency Hierarchy tab of your pom.xml, you can see that this single starter POM combines several dependencies in it.



Dependency Hierarchy

```
spring-boot-starter-data-jpa : 2.2.5.RELEASE [compile]
  spring-boot-starter-aop : 2.2.5.RELEASE [compile]
    spring-boot-starter : 2.2.5.RELEASE [compile]
    spring-aop : 5.2.4.RELEASE [compile]
    aspectjweaver : 1.9.5 [compile]
  spring-boot-starter-jdbc : 2.2.5.RELEASE [compile]
    spring-boot-starter : 2.2.5.RELEASE [compile]
    HikariCP : 3.4.2 [compile]
    spring-jdbc : 5.2.4.RELEASE [compile]
  jakarta.activation-api : 1.2.2 [compile]
  jakarta.persistence-api : 2.2.3 [compile]
  jakarta.transaction-api : 1.3.3 [compile]
  hibernate-core : 5.4.12.Final [compile]
    jboss-logging : 3.4.1.Final (managed from 3.3.2.Final) [compile]
    javassist : 3.24.0-GA [compile]
    byte-buddy : 1.10.8 (managed from 1.10.7) [compile]
    antlr : 2.7.7 [compile]
    jandex : 2.1.1.Final [compile]
    classmate : 1.5.1 [compile]
    dom4j : 2.1.1 [compile]
    hibernate-commons-annotations : 5.1.0.Final [compile]
    jaxb-runtime : 2.3.2 (managed from 2.3.1) [compile]
  spring-data-jpa : 2.2.5.RELEASE [compile]
    spring-data-commons : 2.2.5.RELEASE [compile]
      spring-core : 5.2.4.RELEASE [compile]
      spring-beans : 5.2.4.RELEASE [compile]
      slf4j-api : 1.7.30 (managed from 1.7.26) [compile]
```

Resolved Dependencies

```
accessors-smart : 1.2 [test]
android-json : 0.0.20131108.vaadin1 [test]
antlr : 2.7.7 [compile]
apiguardian-api : 1.1.0 [test]
asm : 5.0.4 [test]
aspectjweaver : 1.9.5 [compile]
assertj-core : 3.13.2 [test]
byte-buddy : 1.10.8 [compile]
byte-buddy-agent : 1.10.8 [test]
classmate : 1.5.1 [compile]
dom4j : 2.1.1 [compile]
FastInfoset : 1.2.16 [compile]
hamcrest : 2.1 [test]
hibernate-commons-annotations : 5.1.0.Final [compile]
hibernate-core : 5.4.12.Final [compile]
HikariCP : 3.4.2 [compile]
istack-commons-runtime : 3.0.8 [compile]
jakarta.activation-api : 1.2.2 [compile]
jakarta.annotation-api : 1.3.5 [compile]
jakarta.persistence-api : 2.2.3 [compile]
jakarta.transaction-api : 1.3.3 [compile]
jakarta.xml.bind-api : 2.3.2 [compile]
jandex : 2.1.1.Final [compile]
javassist : 3.24.0-GA [compile]
jaxb-runtime : 2.3.2 [compile]
jboss-logging : 3.4.1.Final [compile]
```

Spring Data JPA dependency:

The following is the Spring Data JPA dependency - spring-boot-starter-data-jpa added in pom.xml. This dependency will add all the necessary libraries for Spring ORM to the project automatically.

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

Database dependency:

The MySQL dependency mysql-connector-java added in pom.xml.

```xml
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
```

Note: Incase if any other DB is used an appropriate dependency can be used.

Configuring database and ClientApplication

3.application.properties: This is for adding the database details to the project (added the necessary details for MySQL DB).

```
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/sample
spring.datasource.username=root
spring.datasource.password=root
```

4.ClientApplication:

The class Client is annotated with @SpringBootApplication annotation which is used to bootstrap Spring Boot application.

```
@SpringBootApplication
public class ClientApplication{
public static void main(String[] args) {
SpringApplication.run(ClientApplication.class, args);
}
}
```

The @SpringBootApplication annotation indicates that it is a configuration class and also triggers auto-configuration and component scanning. It is a combination of following annotations with their default attributes.

- @EnableAutoConfiguration – This annotation enables auto-configuration for Spring boot application which automatically configures our application based on the dependencies that a developer has already added.
- @ComponentScan – This enables the Spring bean dependency injection feature by using @Autowired annotation. All application components which are annotated with @Component, @Service, @Repository or @Controller are automatically registered as Spring Beans. These beans can be injected by using @Autowired annotation.
- @Configuration – This enables Java based configurations for Spring boot application.

The class that is annotated with @SpringBootApplication will be considered as the main class, is also a bootstrap class. It kicks starts the application by invoking the SpringApplication.run() method. The developer needs to pass the .class file name of the main class to the run() method.

Note: A developer can write any piece of code to interact with an end user by implementing the CommandLineRunner interface and overriding the run() method. Let's discuss more on this.

Executing the Spring Boot application:
To execute the Spring Boot application run the application as a standalone Java class which contains the main method.
Spring Boot Runners: So far you have learned how to create and start Spring Boot application. Now suppose you want to perform some action immediately after the application has started then for this Spring Boot provides the following two interfaces:

- CommandLineRunner
- ApplicationRunner

CommandLineRunner is the Spring Boot interface with a run() method. Spring Boot automatically calls this method of all beans implementing this interface after the application context has been loaded. To use this interface, you can modify the ClientApplication.java class as follows:

```
@SpringBootApplication
public class ClientApplication implements CommandLineRunner {
public static void main(String[] args) {
SpringApplication.run(DemoSpringBootApplication.class, args);
}
@Override
public void run(String... args) throws Exception {
System.out.println("Welcome to CommandLineRunner");
}
}
```

We have seen how Spring Boot helps in developing Spring Data JPA applications.

Let us now discuss how to implement the InfyTel application CRUD operation using Spring Data JPA with Spring Boot in the coming modules.

4. A. Explain the procedure to perform various CRUD operations using Spring Data JPA Configuration.

As seen in the **Spring Data JPA Repository hierarchy** when we are extending the **JpaRepository** interface then Spring provides the auto implementation of all the methods from *JpaRepository*, *PagingAndSortingRepository*, *CrudRepository* and will make them available to the application. An appropriate method can be used in the **service layer** of the application depending on the required database operation.

So our CustomerRepository interface will look as below:

```
package com.infyTel.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import com.infyTel.domain.Customer;
public interface CustomerRepository extends JpaRepository<Customer, Long>{
}
```

Spring Data JPA provides a default implementation for each method defined by one of its repository interfaces. That means that you no longer need to implement basic read or write operations by implementing the above interface. So for implementing our CRUD operations we are going to use 4 methods from their respective interfaces, which are listed below:

- saveAndFlush(S entity) of JpaRepository which saves an entity and flushes changes instantly.
- deleteById(ID id) of CrudRepository which deletes the entity based on the given id/primary key.
- findById(ID id) of CrudRepository which returns a given entity based on the given id/primary key.
- save(S entity) of CrudRepository which saves the given entity in the database.

Now let's look at how the service layer is implemented and the service implementation class uses these 4 methods depending on the required database operation.

CRUD Operation with Spring Data JPA - Service Layer

The service interface will act as a bridge between our Client class and Service implementation class. So to perform CRUD operation we can define all the required methods in the CustomerService interface as below:

CustomerService.java:

```
package com.infyTel.service;
import com.infyTel.dto.CustomerDTO;
public interface CustomerService {
public void insertCustomer(CustomerDTO Customer) ;
public void removeCustomer(Long phoneNo);
public CustomerDTO getCustomer(Long phoneNo);
public String updateCustomer(Long phoneNo,Integer newPlanId);
}
```

The service implementation class invokes repository methods through repository bean to perform the required CRUD operation as shown below:

CustomerServiceImpl.java:

```java
package com.infyTel.service;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infyTel.domain.Customer;
import com.infyTel.dto.CustomerDTO;
import com.infyTel.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService {
@Autowired
private CustomerRepository repository;
@Override
public void insertCustomer(CustomerDTO customer) {
repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
}
@Override
public void removeCustomer(Long phoneNo) {
repository.deleteById(phoneNo);
}
@Override
public CustomerDTO getCustomer(Long phoneNo) {
Optional<Customer> optionalCustomer = repository.findById(phoneNo);
Customer customerEntity = optionalCustomer.get();// Converting Optional<Customer> to
Customer
CustomerDTO customerDTO = Customer.prepareCustomerDTO(customerEntity);
return customerDTO;
}
@Override
public String updateCustomer(Long phoneNo, Integer newPlanId) {
Optional<Customer> optionalCustomer = repository.findById(phoneNo);
Customer customerEntity = optionalCustomer.get();
customerEntity.setPlanId(newPlanId);
repository.save(customerEntity);
return "The plan for the customer with phone number :" + phoneNo + " has been updated
successfully.";
}
}
```

In service class implementation, we are autowiring CustomerRepository instance to make use of required repository methods in the service layer.

Even though we are defining CustomerRepository as an interface, we are able to get an instance of CustomerRepository because Spring internally provides a proxy object for this interface with auto-generated methods. Hence we could autowire bean of CustomerRepository.

4 B. Explain the concept of Pagination & Sorting with an example.

Spring Data JPA – Pagination

Let us now understand paging and sorting support from Spring Data.

PagingAndSortingRepository interface of Spring Data Commons provides methods to support paging and sorting functionalities.

```
public interface PagingAndSortingRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {
Iterable<T> findAll(Sort sort);
Page<T> findAll(Pageable pageable);
}
```

A Page object provides the data for the requested page as well as additional information like total result count, page index, and so on.

Pageable instance dynamically adds paging details to statically defined query. Let us see more details in the example.

Pagination

The steps to paginate the query results are:

Step 1: JpaRepository is a sub-interface of the PagingAndSortingRepository interface. The Application standard interface has to extend JpaRepository to get paging and sorting methods along with common CRUD methods.

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

Note: findAll(Pageable page) and findAll(Sort sort) methods are internally provided by Spring.

Step 2: In client code, create a org.springframework.data.domain.Pageable object by instantiating org.springframework.data.domain.PageRequest describing the details of the requested page is shown below:

To ask for the required page by specifying page size, a new PageRequest instance must be created.

```
//First argument '0" indicates first page and second argument 4 represents number of records.
Pageable pageable = PageRequest.of(0, 4);
```

Step 3: In the client code, pass the Pageable object to the repository method as a method parameter.

```
Page<Customer> customers = customerRepository.findAll(pageable);
```

Spring Data JPA – Sorting

Sorting is to order query results based on the property values of the entity class.

For example, to sort employee records based on the employee's first name field can be done using the below steps:

Step 1: Standard interface has to extend JpaRepository to use sorting method provided by Spring

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

Step 2: In the client code, create org.springframework.data.domain.Sort instance describing the sorting order based on the entity property either as ascending or descending and pass the instance of Sort to the repository method.

```
customerRepository.findAll(Sort.by(Sort.Direction.ASC,"name"));
```

In the Sort() method used above,

- The first parameter specifies the order of sorting i.e. is ascending order.

- The second parameter specifies the field value for sorting.

UNIT-III (PART -B)

5. A. How does Spring support the Query creation based on the Method Name?

Query method names are derived by combining the property name of an entity with supported keywords such as "findBy", "getBy", "readBy" or "queryBy".

```
//method name where in <Op> is optional, it can be Gt,Lt,Ne,Between,Like etc..
findBy <DataMember><Op>
```

Example: Consider the Customer class as shown below:

```
public class Customer {
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
//getters and setters
}
```

To query a record based on the address using query creation by the method name, the following method has to be added to the CustomerRepository interface.

```
interface CustomerRepository extends JpaRepository<Customer, Long>{
Customer findByAddress(String address); // method declared
}
```

The programmer has to provide only the method declaration in the interface. Spring takes care of auto-generating the query, the mechanism strips the prefix findBy from the method and considers the remaining part of the method name and arguments to construct a query.

Let us understand this concept in detail, through a demo.

Common findBy methods

Consider the Customer and Address classes given below:

Customer.java

```
@Entity
public class Customer{
@Id
int customerId;
boolean active;
int creditPoints;
String firstName;
String lastName;
String contactNumber;
String email;
@OneToOne( cascade = CascadeType.ALL)
@JoinColumn
Address address;
-----------
}
```

Address.java

```
@Entity
public class Address {
@Id
private int addressId;
private String city;
private String pincode;
----------------
}
```

The CustomerRepository interface with some common findBy methods is shown below:

```
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
// Query record based on email
// Equivalent JPQL: select c from Customer c where c.email=?
Customer findByEmail(String email);
// Query records based on LastName is like the provided last name
// select c from Customer c where c.lastName LIKE CONCAT('%',?,'%')
List<Customer> findByLastNameLike(String lastname);
// Query records based on email or contact number
// select c from Customer c where c.email=? or c.contactNumber=?
List<Customer> findByEmailOrContactNumber(String email, String number);
// Query records based on FirstName and city details. Following query creates the property
traversal for city as Customer.address.city
// select c from Customer c  where c.firstName=? and c.address.city=?
List<Customer>  findByFirstNameAndAddress_City(String fname, String city);
// Query records based on last name and order by ascending based on first name
// select c from Customer c where c.lastName=? order by c.firstName
List<Customer> findByLastNameOrderByFirstNameAsc(String lastname);
// Query records based on specified list of cities
//select c from Customer c where c.address.city in ?1
List<Customer> findByAddress_CityIn(Collection<String> cities);
// Query records based if customer is active
//select c from Customer c where c.active = true
List<Customer> findByActiveTrue();
// Query records based on creditPoints >= specified value
//select c from Customer c where c.creditPoints >=?1
List<Customer> findByCreditPointsGreaterThanEqual(int points) ;
// Query records based on creditpoints between specified values
//select c from Customer c where c.creditPoints between ?1 and ?2
List<Customer> findByCreditPointsBetween(int point1, int point2)
}
```

Note:

List<Customer> findByFirstNameAndAddress_City(String fname, String city)
It can also be written as:

List<Customer> findByFirstNameAndAddressCity(String fname, String city);

In case there is a property by name “addressCity” in Customer class then to resolve this ambiguity we can add _ inside findBy method for manual traversal.

Spring Data JPA - Query Precedence

So far, we learned the following Query creation approaches in Spring Data JPA.

- Query creation based on the method name.
- Query creation using @NamedQuery: JPA named queries through a naming convention.
- Query creation using @Query: annotate your query method with @Query.

If a query is provided using more than one approach in an application. What is the default precedence given by the Spring?

Following is the order of default precedence:

1. @Query always takes high precedence over other options
2. @NameQuery
3. findBy methods

Note: If a developer wants to change the query precedence, then he can provide with extra configuration. This is not been discussed in this course.

5 B. How does Spring support the Query creation based on @NamedQuery and @Query?

Spring Data JPA - Query Creation Using JPA NamedQueries

By now you know, how to support a query through a method name.
Though a query created from the method name suits very well but, in certain situations, it is difficult to derive a method name for the required query.
The following options can be used in these scenarios:

1. Using JPA NamedQueries: JPA named queries through a naming convention
2. Using @Query: Use @Query annotation to your query method

Let us now understand JPA NamedQueries:

Define annotation-based configuration for a NamedQuery at entity class with @NamedQuery annotation specifying query name with the actual query.

```
@Entity
//name starts with the entity class name followed by the method name separated by a dot.
@NamedQuery(name = "Customer.findByAddress", query = "select c from Customer c where
c.address = ?1")
public class Customer {
@Id
@Column(name = "phone_no")
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
--------
}
```

Now for executing this named query one needs to specify an interface as given below with method declaration.

```
public interface CustomerRepository extends JpaRepository<Customer, Long>{
Customer findByAddress(String address);
}
```

Spring Data will map a call to findByAddress() method to a NamedQuery whose name starts with entity class followed by a dot with the method name. Hence, in the above code, Spring will use the NamedQuery with the name Customer.findByAddress() method instead of creating it.

NamedQuery approach has advantage as maintenance costs are less because the queries are provided through the class. However, the drawback is that for every new query declaration domain class needs to be recompiled.

Spring Data JPA - Query Creation Using @Query Annotation
The NamedQueries approach is valid for the small number of queries.
@Query annotation can be used to specify query details at repository interface methods instead of specifying at entity class. This will also reduce the entity class from persistence related information.
Queries annotated to the query method has high priority than queries defined using @NamedQuery.
@Query is used to write flexible queries to fetch data.
Example: Declare query at the method level using **@Query**.

```
public interface CustomerRepository extends JpaRepository<Customer, Long>{
//Query string is in JPQL
@Query("select cus from Customer cus where cus.address = ?1")
Customer findByAddress(String address);
}
```

@Query annotation supports both JPQL (Java Persistence Query Language) and native SQL queries.

By default, it supports JPQL. The nativeQuery attribute must be set to true to support native SQL.

Example: Declare query at the method level using @Query with nativeQuery set to true.

```
public interface CustomerRepository extends JpaRepository<Customer, Long>{
//Query string is in SQL
@Query("select cus from Customer cus where cus.address = ?1", nativeQuery = true)
Customer findByAddress(String address);
}
```

The disadvantage of writing queries in native SQL is that they become vendor-specific database and hence portability becomes a challenge. Thus, both @NamedQuery and @Query supports JPQL.

Now, what is JPQL?
JPQL:
JPQL is an object-oriented query language that is used to perform database operations on persistent entities. Instead of a database table, JPQL uses the entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks.

Features:

The features of JPQL are that it can:

- perform join operations
- update and delete data in a bulk
- perform an aggregate function with sorting and grouping clauses
- provide both single and multiple value result types

Creating Queries using JPQL:
JPQL provides two methods that can be used to perform database operations. They are: -

1. **Query createQuery(String name)** - The createQuery() method of EntityManager interface is used to create an instance of the Hibernate Query interface for executing JPQL statement. This method creates dynamic queries that can be defined within business logic.

Some of the examples of JPQL using createQuery method:(assuming the entity class name as - 'CustomerEntity' which is mapped to a relational table 'Customer')

- Fetching all the Customer names:

```
Query query = em.createQuery("Select c.name from CustomerEntity c");
```

- Updating the plan of a customer:

```
Query query = em.createQuery( "update CustomerEntity SET planId=5 where
phoneNumber=7022713766");
query.executeUpdate();
```

- Deleting a customer:

```
Query query = em.createQuery( "delete from CustomerEntity where
phoneNumber=7022713766");
query.executeUpdate();
```

2. **Query createNamedQuery(String name)** - The createNamedQuery() method of EntityManager interface is used to create an instance of the Hibernate Query interface for executing named queries. This method is used to create static queries that can be defined in the entity class.

Let's see a simple example of JPQL using createNamedQuery method to fetch all the details of customers in InfyTel application:(Assume the entity class name is - 'CustomerEntity' which is mapped to a relational table 'Customer')

```
@NamedQuery(name = "getAll" , query = "Select c from CustomerEntity s")
```

6. A. Explain about Spring Declarative Transaction in Spring Data JPA.

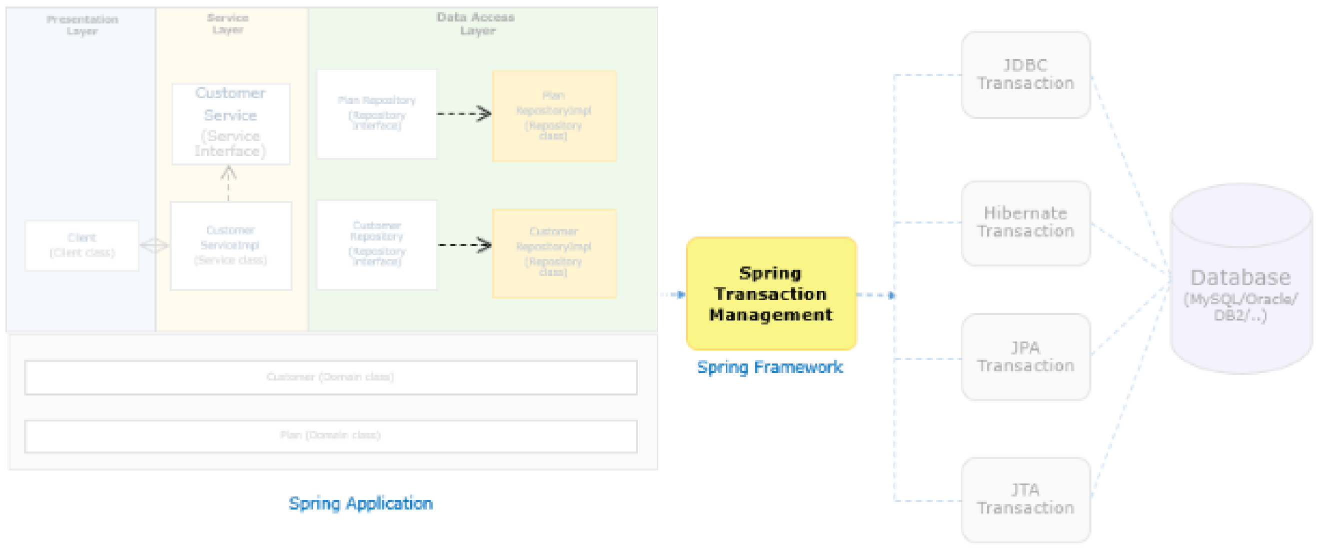
What is Spring Transaction?

The Spring framework provides a common transaction API irrespective of underlying transaction technologies such as JDBC, Hibernate, JPA, and JTA.

You can switch from one technology to another by modifying the application's Spring configuration. Hence you need not modify your business logic anytime.

Spring provides an abstract layer for transaction management by hiding the complexity of underlying transaction APIs from the application.

- Spring supports both local and global transactions using a consistent approach with minimal modifications through configuration.
- Spring supports both declarative and programmatic transaction approaches.


Presentation Layer
Service Layer
Data Access Layer
Customer Service (Service Interface)
Plan Repository (Repository Interface)
Plan RepositoryImpl (Repository class)
Client (Client class)
Customer ServiceImpl (Service class)
Customer Repository (Repository Interface)
Customer RepositoryImpl (Repository class)
Customer (Domain class)
Plan (Domain class)
Spring Application
Spring Transaction Management
Spring Framework
JDBC Transaction
Hibernate Transaction
JPA Transaction
JTA Transaction
Database (MySQL/Oracle/ DB2/..)

Spring Transaction - Approaches

Different ways to achieve Spring transaction:

Type	**Definition**	**Implementation**
Declarative Transaction	Spring applies the required transaction to the application code in a declarative way using a simple configuration. Spring internally uses Spring AOP to provide transaction logic to the application code.	• Using pure XML configuration • Using @Transactional annotation approach
Programmatic Transaction	The Required transaction is achieved by adding transaction-related code in the application code. This approach is used for more fine level control of transaction requirements. This mixes the transaction functionality with the business logic and hence it is recommended to use only based on the need.	• Using the TransactionTemplate (adopts the same approach as JdbcTemplate) • Using a PlatformTransactionManager implementation.

Which is the preferred approach to implement Spring transactions?
Declarative transaction management is the most commonly used approach by Spring Framework users.
This option keeps the application code separate from the transaction serves as the required transaction is provided through the only configuration.

Let us proceed to understand, how Spring declarative transactions can be implemented using the annotation-based approach in the Spring Data JPA application in detail.

Note:
- In this course, we will be covering only Spring declarative transactions using the annotation-based approach.
- Spring Declarative Transaction is treated as an aspect. Spring will apply the required transaction at run time using Spring AOP.

Application Development Using Spring Declarative Transaction

Considering the InfyTel scenario update the Customer's current plan and Plan details. The InfyTel application uses Spring ORM for data access layer implementation. Let us now apply the Spring transaction to this application.

This requirement provides an update on the following tables:

1. Customer table: To update the current plan.
2. Plan table: To update the plan details.

Declarative Transaction configurations

The important steps to implement Spring declarative transaction in Spring ORM application is :

- Add @Transactional annotation on methods that are required to be executed in the transaction scope as all other things like dependencies management, etc are already taken care of by spring-boot-starter-data-jpa jar

Spring Data JPA Dependency:

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

MySQL Driver dependency:

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
```

We can implement Spring declarative transactions using the annotation-based approach.
Let us understand more on @Transactional annotation.
@Transactional annotation offers ease-of-use to define the transaction requirement at method, interface, or class level.
Through declarative transaction management, update (Customer customer) method can be executed in transaction scope using Spring.

```
public class CustomerServiceImpl {
------------------------
@Transactional
public void update(Customer customer) {
//Method to update the current plan in Customer table
customerDAO.update(customer);
//Method to update the new plan details in Plan table
planDAO.updatePlan(customer.getPlan());
}
}
```

This annotation will be identified automatically by Spring Boot if we have already included the spring-boot-starter-data-jpa jar.

Should the developer use @Transactional annotation before every method?

No, @Transactional annotation can be placed at the class level which makes all its methods execute in a transaction scope. For e.g. insertCustomer() and updateCustomerDetails() methods executes in transaction scope.

```
@Transactional    // This annotation makes all the methods of this class to execute in
transaction scope
public class CustomerServiceImpl {
-----------
public void insertCustomer(Customer customer) {
```

```
------------------
}
public void updateCustomerDetails(Customer customer) {
------------------
}
}
```

Spring Declarative Transaction Management - Transaction Managers

The key interface to implement the Spring transaction is org.springframework.transaction.PlatformTransactionManager Interface.

There are different implementations available for PlatformTransactionManager to support different data access technologies such as JDBC, JTA, Hibernate, JPA and so on.

The PlatformTransactionManager implementations are responsible for managing transactions. Hence, they are transaction managers.

The following table describes some of the commonly used transaction managers:

TransactionManager	Data Access Technique
DataSourceTransactionManager	JDBC
HibernateTransactionManager	Hibernate
JpaTransactionManager	JPA
JtaTransactionManager	Distributed transaction

Transaction managers can be defined as any other beans in the Spring configuration only if we are not using Spring Boot.

Since the InfyTel application is developed using Spring Boot and Spring Data JPA, So no bean needed to be defined explicitly.

6 B. Explain about the update operation in Spring Data JPA.

Now that you know, how to use @Query annotation for query operations.

Can @Query annotation be used for performing modification operations?
Yes, it can execute modifying queries such as update, delete or insert operations using @Query annotation along with @Transactional and @Modifying annotation at query method.

Example: Interface with a method to update the name of a customer based on the customer's address.

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
@Transactional
@Modifying(clearAutomatically = true)
@Query("update Customer c set c.name = ?1 where c.address = ?2")
void update(String name, String address);
}
```

Let us now understand in detail, why there is a need for the following:

@Modifying: This annotation will trigger @Query annotation to be used for an update operation instead of a query operation.

@Modifying(clearAutomatically = true): After executing modifying query, EntityManager might contain unrequired entities. It is a good practice to clear the EntityManager cache automatically by setting @Modifying annotation’s clearAutomatically attribute to true.

@Transactional: Spring Data provided CRUD methods on repository instances that support transactional by default with read operation and by setting readOnly flag to true. Here, @Query is used for an update operation, and hence we need to override default readOnly behavior to read/write by explicitly annotating a method with @Transactional.

More on @Modifying

@Modifying: This annotation triggers the query annotated to a particular method as an updating query instead of a selecting query. As the EntityManager might contain outdated entities after the execution of the modifying query, we should clear it. This effectively drops all non-flushed changes still pending in the EntityManager. If we don't wish the EnetiyManager to be cleared automatically we can set @Modifying annotation's clearAutomatically attribute to false.

Fortunately, starting from Spring Boot 2.0.4.RELEASE, Spring Data added **flushAutomatically** flag to auto flush any managed entities on the persistence context before executing the modifying query.

Thus, the safest way to use **Modifying** is:

```
@Modifying(clearAutomatically=true, flushAutomatically=true)
```

Now let's take a small scenario where we don't use these two attributes with **@Modifying** annotation:

Assume a Customer table with two columns name and active, exists in the database with only one row as Tom, true.

Repository:

```
public interface CustomerRepository extends JPARepository<Customer, Intger> {
@Modifying
@Query("delete Customer c where c.active=0")
public void deleteInActiveCustomers();
}
```

In the above repository, @Modifying annotation is used without clearAutomatically and flushAutomatically attributes. So let's visualize what happens when the Service call happens.

Visualization -1: If flushAutomatically attribute is not used in the Repository:

```
public class CustomerServiceImpl implements CustomerService {
Customer customerTom = customerRepository.findById(1); // Stored in the First Level Cache
customerTom.setActive(false);
customerRepository.save(customerTom);
customerRepository.deleteInActiveUsers();// By all means it won't delete the customerTom
/*customerTom still exist since customerTom with 'active' attribute being set to false was not flushed into the database when @Modifying kicks in*/
}
```

Visualization -2: If clearAutomatically attribute is not used in the Repository:

```
public class CustomerServiceImpl implements CustomerService {
Customer customerTom = customerRepository.findById(1); // Stored in the First Level Cache
customerRepository.deleteInActiveCustomers(); // We think that customerTom is deleted now
System.out.println(customerRepository.findById(1).isPresent()) // Will return TRUE
System.out.println(customerRepository.count()) // Will return 1
// TOM still exist in this transaction persistence context
// TOM's object was not cleared upon @Modifying query execution
// TOM's object will still be fetched from First Level Cache
/* clearAutomatically attribute takes care of doing the clear part on the objects being modified
for current transaction persistence context*/
}
```

7. A. Explain Custom Repository Implementation using Spring Data JPA.

So far, we have seen different approaches to create queries.
Sometimes, customization of a few complex methods is required. Spring easily support this, you can use custom repository code and integrate it with Spring Data abstraction.
Example: Let us consider a customer search scenario wherein customer details needs to be fetched based on name and address or gender or age.

Entity class Customer is shown below:

```
@Entity
public class Customer {
@Id
private Long phoneNumber;
private String name;
private Integer age;
private Character gender;
private String address;
private Integer planId;
-------
}
```

The steps to implement the custom repository with a method to Retrieve customer records based on search criteria are as follows:

Step 1: Define an interface and an implementation for the custom functionality.

```
public interface ICustomerRepository {
public List<Customer> searchCustomer(String name, String addr, Character gender, Integer
age);
}
```

Step 2: Implement this interface using repository class as shown below:

```
public class CustomerRepositoryImpl implements ICustomerRepository{
private EntityManagerFactory emf;
@Autowired
public void setEntityManagerFactory(EntityManagerFactory emf) {
this.emf = emf;
}
@Override
public List<Customer> searchCustomer(String name, String address, Character gender, Integer
age) {
```

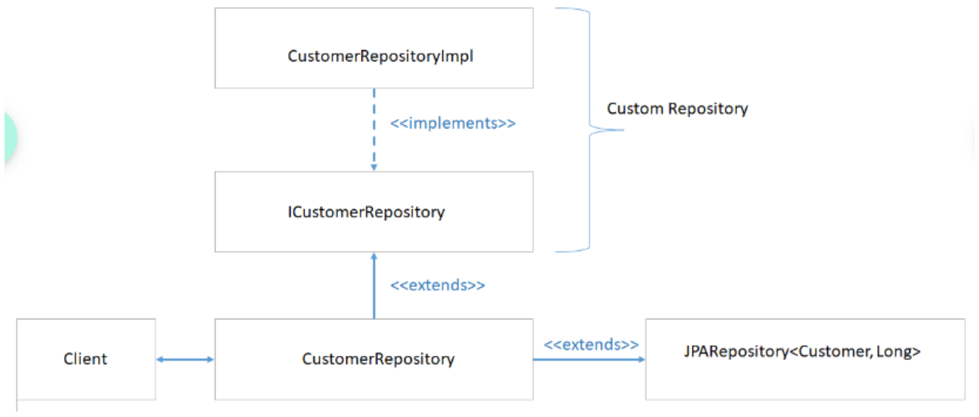
```
EntityManager em = emf.createEntityManager();
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> query = builder.createQuery(Customer.class);
Root<Customer> root = query.from(Customer.class);
Predicate cName = builder.equal(root.get("name"), name);
Predicate cAddress = builder.equal(root.get("address"), address);
Predicate exp1 = builder.and(cName, cAddress);
Predicate cGender = builder.equal(root.get("gender"), gender);
Predicate cAge = builder.equal(root.get("age"), age);
Predicate exp2 = builder.or(cGender,cAge);
query.where(builder.or(exp1, exp2));
return em.createQuery(query.select(root)).getResultList();
}
```

Step 3: Define repository interface extending the custom repository interface as shown below:

```
public interface CustomerRepository extends JpaRepository<Customer, Long>,ICustomerRepository{
}
```

Now, standard repository interface CustomerRepository extends both JpaRepository and the custom interface(ICustomerRepository). Hence, all the Spring data provided default methods, as well as the custom defined method(searchCustomer), will be accessible to the clients.

Data access layer of an application has the following files with dependencies is as shown below:



7 B. Explain Briefly which best practices you should follow when designing the application using Spring Data JPA.

Some of the best practices that need to be followed as part of the Quality and Security for Spring Boot applications. These practices, when applied during designing and developing a Spring Boot application, yields better performance.

Best Practices:

1. Use Spring Initializr for creating Spring Boot projects
2. Use the correct project Structure while creating Spring Boot projects
3. Choose Java-based configuration over XML based configuration
4. Use Setter injection for optional dependencies and Constructor injection for mandatory dependencies
5. Use @Service for Business Layer classes
6. Follow the Spring bean naming conventions while creating beans

Let us understand the reason behind these recommendations and their implications.

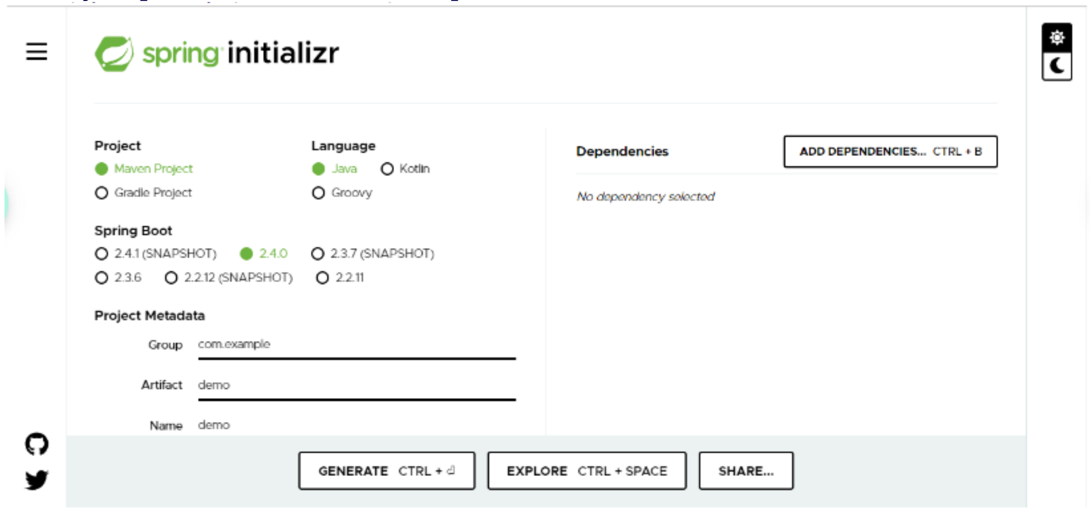
Note: For the demos already covered in the course, we would be applying these best practices.

Use Spring Initializr for starting new Spring Boot projects

There are three different ways to generate a Spring Boot project. They are:

- Using Spring Initializr
- Using the Spring Tool Suite (STS)
- Using Spring Boot CLI

But the recommended and simplest way to create a Spring Boot application is the Spring Boot Initializr as it has good UI to download a production-ready project. And the same project can be directly imported into the STS/Eclipse.



spring initializr

Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.4.1 (SNAPSHOT) ● 2.4.0 ○ 2.3.7 (SNAPSHOT)
○ 2.3.6 ○ 2.2.12 (SNAPSHOT) ○ 2.2.11

Project Metadata
Group: com.example
Artifact: demo
Name: demo

Dependencies ADD DEPENDENCIES... CTRL + B
No dependency selected

Note: The above screen keeps changing depending on updates from Pivotal and changes in the Spring Boot version.

Standard Project Structure for Spring Boot Projects

To ease of use the Developers can follow two ways of creating the package structure in a Spring Boot application, which will also help them to maintain the application in the future.

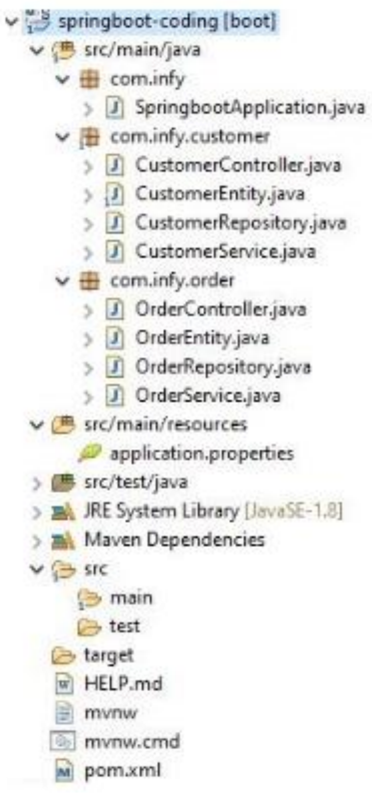
Before delving deeper let us discuss if a Developer is not providing any package structure. i.e. "default" package for all the classes and its disadvantages.

"default" Package:

When the Developer is not providing the package declaration at the class level, then the class will be in the "default" package. This approach should be avoided because it can cause problems for the Boot applications that frequently use the annotations like @EntityScan, @ComponentScan, or @SpringBootApplication and the Spring Boot application has to mandatorily scan all the classes in the "default" package instead of the required classes which will decrease the performance of the application. So it is recommended to create the appropriate classes inside appropriate packages with proper Java's package naming conventions. For example, com.infosys.utilities, com.infosys.client, com.infosys.service, etc.

So the developer can follow either one of the two recommended approaches.

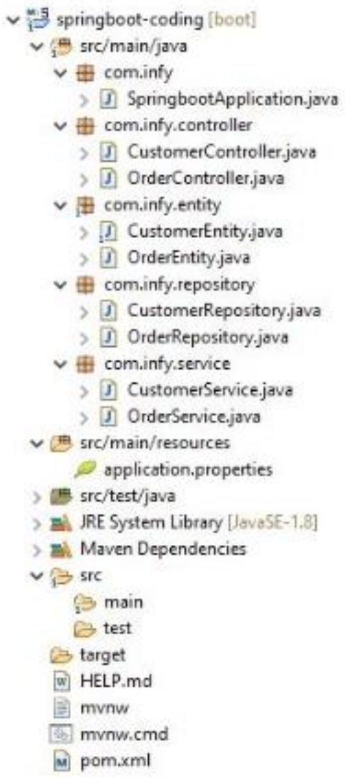
First approach: The first approach shows a layout which generally recommended by the Spring Boot team. In this approach, all the related classes for the Customer have grouped in the "**com.infy.customer**" package and all the related classes for the Order have grouped in the "**com.infy.order**" package



```
springboot-coding [boot]
  src/main/java
    com.infy
      SpringbootApplication.java
    com.infy.customer
      CustomerController.java
      CustomerEntity.java
      CustomerRepository.java
      CustomerService.java
    com.infy.order
      OrderController.java
      OrderEntity.java
      OrderRepository.java
      OrderService.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
    main
    test
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Second approach: The above-shown approach works fine but the developers can follow the second approach as that's better readable and maintainable.

In this approach, all the service classes related to Customer and Order are grouped in the **"com.infy.service"** package. Similar to that we can see we grouped the **dao**, **controller**, and **model** classes.



```
springboot-coding [boot]
  src/main/java
    com.infy
      SpringbootApplication.java
    com.infy.controller
      CustomerController.java
      OrderController.java
    com.infy.entity
      CustomerEntity.java
      OrderEntity.java
    com.infy.repository
      CustomerRepository.java
      OrderRepository.java
    com.infy.service
      CustomerService.java
      OrderService.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
    main
    test
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Note: In our course, we are following the second approach.

Using Java-based configuration - @Configuration

Spring Boot gives us the flexibility to configure the project either using the .properties file or the .java file or the .xml file. So for a large multitier application if configuration related code(apart from Spring Boot auto-configuration) it's a good practice to write the configurations in a Java-based configuration file and divide them into multiple files as per the developer's convenience.

Writing the configurations in a Java-based configuration file serves compile-time checking and dividing them into multiple files serves more readability and maintainability.

So it's advisable to keep DAO related configurations in one file (DAOConfig.java). Web configurations in another file (WebConfig.java) which finally can be imported in the bootstrap class or main configuration file.

Let's analyze more on the above said best practice.

The developer needs to follow certain steps to put all the **@Configuration** files into a single **@SpringBootApplication** file OR **@Configuration** file(centralized configuration file). So, for putting all the configuration files into a single file the developer needs to use **@Import** annotation.

Let's discuss the steps.

Step 1: Create multiple configuration files using the Java-based configuration.

```
@Configuration
public class DAOConfig {
//Database related configurations
}



@Configuration
public class WebConfig {
//Web related configurations
}
```

Step 2: Multiple configuration files can be imported to a single centralized configuration file or to the bootstrap class of the Spring Boot application.

```
@Configuration
@Import({ DAOConfig.class, WebConfig.class })
public class ApplicationConfig extends ConfigurationSupport {
//This is a centralized class containing all the Configurations
/*All the @Bean methods from DAOConfig and WebConfig will be referred to in this
configuration class along with the @Bean methods in ApplicationConfig class*/
}
```

OR

```
@SpringBootApplication
@Import({ DAOConfig.class, WebConfig.class })
public class AppConfig implements CommandLineRunner {
//Bootstrap class
/*@Bean methods from DAOConfig and WebConfig will be referred here along with the
Spring Boot's auto configurations*/
}
```

Note: In our course as the demos are very small demos so we have used the .properties file to configure our Spring Boot projects.

Constructor injection or Setter injection for dependencies

There are three ways to achieve dependency injection in Spring. They are:

1. Constructor injection
2. Setter injection
3. Field injection

The Constructor injection and Setter injection can be mixed by a Spring Boot Developer during the application development but it is a very good practice if he/she can use Constructor injection while injecting the mandatory dependencies and Setter injection while injecting the optional dependencies.
Many Spring developers prefer to use Constructor injection over Setter injection as Constructor injection makes the bean class object immutable.

Using @Service Annotation Class for Business Layer

In a survey, it's observed that few developers directly call the Spring repository classes in the Controller classes which should be avoided. It's recommended to use a service class annotated with @Service annotation to write the business logic.

For example:

```
@Service
public class OrderServiceImpl implements OrderService{
//Assuming a valid interface exists as - OrderService
//All the business logics for OrderService are written here
//Repository method invocations
}
```

Spring Bean Naming Conventions

It is the default feature of a Spring Boot project to auto-configure based on the starter jars as well as to create the beans for all the classes which are annotated with @Controller/@Service/@Repository/@Component by lowering the first character of the class name. So, for example, if the class name is "CustomerService" then the ID for the same bean created in the Spring IoC container will be "customerService".

But when a developer wants to configure some additional beans by creating their own configuration file then, they need to follow the standard Java naming conventions when naming the beans. The standard bean name starts with lowercase and should be a camel-case format.

For example:

```
@Configuration
public class AppConfig{
@Bean("dataSource") //Bean id is dataSource
public DataSource dataSource(){
//Additional DataSource bean configurations
return new DataSource();
}
@Bean("xmlParser")//Bean id is xmlParser
public XMLParser xmlParser(){
//Additional XMLParser bean configurations
return new XMLParser();
}
}
```

Note: Don't use single characters for bean names like @Bean("a") or @Bean("x") or @Bean("A") etc.

Spring Data JPA Best Practices

Let us discuss the best practices which need to be followed as part of the Quality for Spring Data JPA applications. Once these best practices are applied they can help in improving the performance in JPA implementations.

Best Practices:

- Extended interface usage
- Don't fetch more data than you need
- @NamedQuery vs @Query
- JPQL in Custom Repository

Let us understand the reason behind these recommendations and their implications.
Note: For the demos already covered in the course, we would be applying these best-practices.

Extended interface usage

A Spring Data JPA developer can choose the most common way i.e. to extend the appropriate interface from the spring data jpa module. After inheriting the appropriate interface, it immediately provides the basic CRUD operations.

A developer can choose specific method names for parsing queries, which is one of the special features of Spring Data JPA.

But when the method names become more complex it's convenient to use HQL and native SQL.

Let's take a small example to demonstrate the above best practice.

```
@Repository
public interface OrderRepository extends JpaRepository<Order, Integer> {
// HQL
@Query("SELECT "
+ "  DISTINCT o "
+ "FROM "
+ "  Order o "
+ "INNER JOIN "
+ "  o.sender.friends f "
+ "WHERE "
+ "  (p.sender = ?1 OR f = ?1) "
+ "AND p.dateCreated < ?2")
Page<Post> findAllBySenderOrRecieverAndDateCreated(Person person, Date dateCreated
, Pageable page);
}
```

Note: It's a good practice to use HQL as it provides simplicity and more maintainable code.

Don't fetch more data than you need

Pagination can be directly used within the database. You can specify and fetch the required no. of rows.

```
public List<Customer> getAllCustomers() {
EntityManager em = emf.createEntityManager();
Query query = em.createQuery("From Customer");
int pageNumber = 1;
int pageSize = 10;
query.setFirstResult((pageNumber-1) * pageSize);
query.setMaxResults(pageSize);
List <Customer> custList = query.getResultList();
return custList;
}
```

- ***setFirstResult(int)***: Offset index is set to start the Pagination
- ***setMaxResults(int)***: Maximum number of entities can be set which should be included in the page

Column Select:

Fetch the required columns for the select query. If only Customer's name is required, then get only that.

```
public List<String> getAllCustomersName() {
EntityManager em = emf.createEntityManager();
Query query = em.createQuery("select c.name From Customer c");
List <String> custList = query.getResultList();
return custList;
}
```

@NamedQuery vs @Query

Although creating query creation using a method name is a suitable option but in certain scenarios, it's difficult to derive a method name for the required query and the method definition looks so nasty. In this case, below approached can be followed:

1. Using JPA NamedQueries: JPA named queries using a naming convention
2. Using @Query: Use @Query annotation to your query method

Named Query approach has the advantage as maintenance costs are less as the queries are provided through the class. However, the drawback is that for every new query declaration domain class needs to be recompiled.

Let us now understand JPA named queries:

Define annotation-based configuration for a named query at entity class with @NamedQuery annotation specifying query name with the actual query.

```
@Entity
//Use entity class name followed by the user-defined method name separated with a dot(.)
@NamedQuery(name = "FullTimeEmployee.findByEmail", query = "select e from
FullTimeEmployee e where e.email = ?1")
public class FullTimeEmployee {
@Id
private Integer employeeNumber;
private String firstName;
private String lastName;
private String department;
private String email;
-------
}
```

Note: As mentioned in the NamedQueries topic in the course for executing this NamedQuery one needs to specify an interface with method declaration.

For a small number of queries, the NamedQueries approach is valid and works fine.

@Query annotation is used to define query details at repository interface methods instead of specifying at entity class. This will also reduce the entity class from persistence related information.

@Query has a high priority over @NamedQuery.

@Query could be used to write more flexible queries to fetch data.

Declaration of the query at the method level with the help of @Query :

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
//Query string is in JPQL
@Query("select emp from FullTimeEmployee emp where emp.emailAddress = ?1")
FullTimeEmployee  findByEmailAddress(String emailAddress);
}
```

@Query annotation for JPQL and native SQL queries can be used.

By default, it supports JPQL. One has to set the native query attribute to true to support native SQL.

The disadvantage of writing queries in native SQL is that, they become vendor-specific database and hence portability becomes a challenge

UNIT-IV

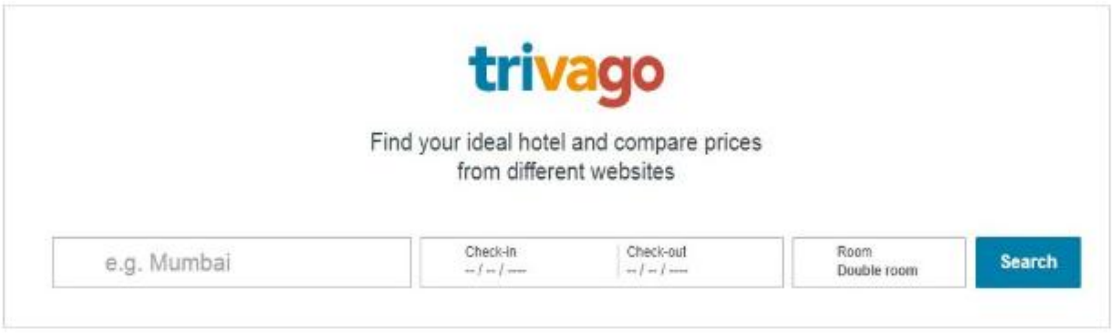
1. A. Briefly explain about the need of Web Services.

Real Time Scenarios

A lot of companies which make money every second like Amazon communicates with various vendors to get the product details.
Also, Amazon leverages different payment gateway options to pay for the purchases.
It is not guaranteed that all the vendor-specific applications and the payment gateway applications are compatible with Amazon in terms of technology and platform where they get executed.

Even then, the communication between these applications and Amazon happens successfully.

Have you ever wondered how this becomes possible?
Note: *All trademarks belong to their respective owners.*



Nowadays, our travel to different locations and stay are quite easy.

Travelling is made easier and comfortable through different forms of transportation.

Let us take the second aspect, stay, for our discussion. In order to make our stay comfortable and pleasant, we need to search for a good hotel that fits in our budget. We all know that different online hotel booking applications are there to make this search possible in a single click.

Now, let us talk about Trivago, a familiar online hotel booking site.

This application will give us the details of the hotels that are available in a particular locality based on our preference. As well, it will allow us to book rooms in a hotel for our stay.

Do you think, Trivago has access to the databases of the hotels to which it associates with?

The answer is a big NO. No one will allow an outsider to get access to their databases, simply because of the security reasons.

Or, do you think, these hotels push their room details to Trivago, periodically?

The answer is NO here as well, simply because when the updates happen periodically, there are situations where the end-customer gets misguided as Trivago and the partner hotels' databases are not in sync.

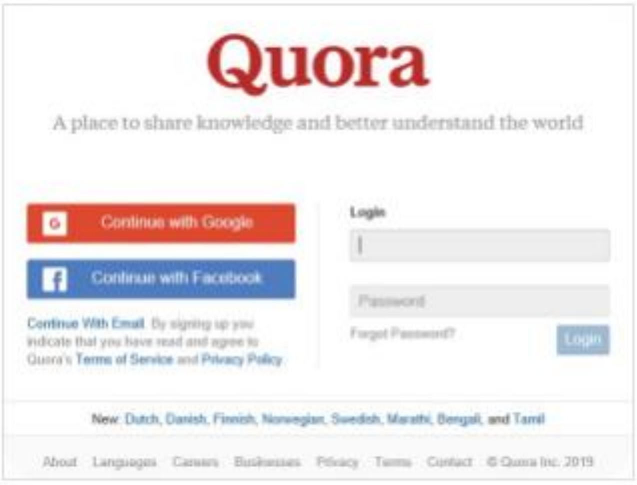
The correct mechanism behind this is, Trivago gives calls to the associated hotels' application services when the end customer requests.

And, calling those applications' services will be possible only when they are compatible with Trivago in terms of language and platform.

But, we cannot expect Trivago's compatibility with all its associated applications.

Then how does Trivago executes its calls successfully?

Note: *All trademarks belong to their respective owners.*


Quora
A place to share knowledge and better understand the world
Continue with Google
Continue with Facebook
Login
Password
Forgot Password?
Login
About Languages Careers Businesses Privacy Terms Contact © Quora Inc. 2019

Continue With Email. By signing up you indicate that you have read and agree to Quora's Terms of Service and Privacy Policy.

New: Dutch, Danish, Finnish, Norwegian, Swedish, Marathi, Bengali, and Tamil

Quora, as we all know, is an ideal place for sharing and gaining knowledge.

Users of this application can post queries about any field and get quality replies. Also, people can step forward and share their knowledge in their fields of interest.

These activities need signing in with Quora.

Quora gives us many sign-in options like Google and Facebook apart from its standard authentication.

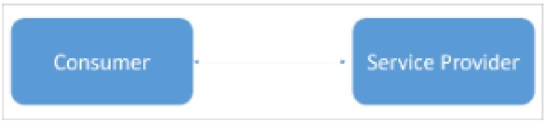
For example, if we have signed in to our Facebook already, it is not required for us to sign in to Quora individually. Rather, we can make use of an option called, "Continue with Facebook". Here, it is implicitly understood that Quora avails the login service of Facebook. And, we see a lot of applications work this way in our day-to-day life.

So, it is quite natural to get a query here. Will Google and Facebook be compatible with Quora in terms of language and platform? Definitely not!!!!

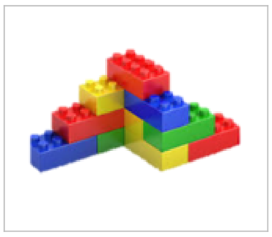
But, how communication happens still?
The answer is Web Services.
Note: All trademarks belong to their respective owners.

1 B. Explain about Service Oriented Architecture.

SOA expands to **Service Oriented Architecture**, is a software model designed for achieving communication among distributed application components, irrespective of the differences in terms of technology, platform, etc.,
The application component that requests for a service is the consumer and the one that renders the service is the producer.



SOA is an architecture that can help to build an enterprise application by availing the services/functionalities from different third-party entities without reinventing the existing functionalities.



SOA Principles

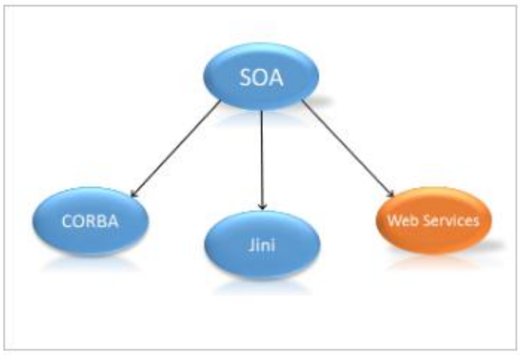
SOA that drives on high-value principles will take care of the following.

Principle	Explanation
Loose coupling	SOA aims at structuring procedures or software components as services. And, these services are designed in such a way that they are loosely coupled with the applications. So, these loosely-coupled services will come into picture only when needed.
Publishing the services	There should be a mechanism for publishing the services that include the functionality and input/output specifications of those services. In short, services are published in such a way that the developers can easily incorporate them into their applications.
Management	Finally, software components should be managed so that there will be a centralized control on the usage of the published services.

These principles are the driving factors behind SOA, ensuring a high level of abstraction, reliability, scalability, reusability, and maintainability.

SOA Implementation

Following is the pictorial representation that speaks about the different ways of realizing and implementing SOA.



CORBA: Common Object Request Broker Architecture is a standard that achieves interoperability among the distributed objects. Using CORBA, it is much possible to make the distributed application components communicate with each other irrespective of the location (the place where the components are available), language and platform.

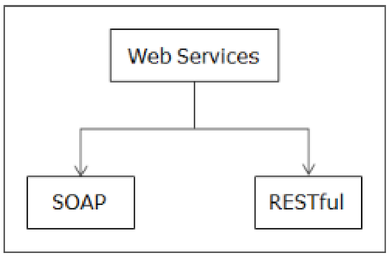
Jini: Jini or Apache River is a way of designing distributed applications with the help of services that interact and cooperate.

Web Services: Web services are a way of making applications interact with each other on the web. Web services leverage the existing protocols and technologies but, uses certain standards.

Web Service is the most popular solution as it eliminates the issues related to interoperability with pre-defined standards and processes in place.

2. Types of Web Services

Web Services are of two types.

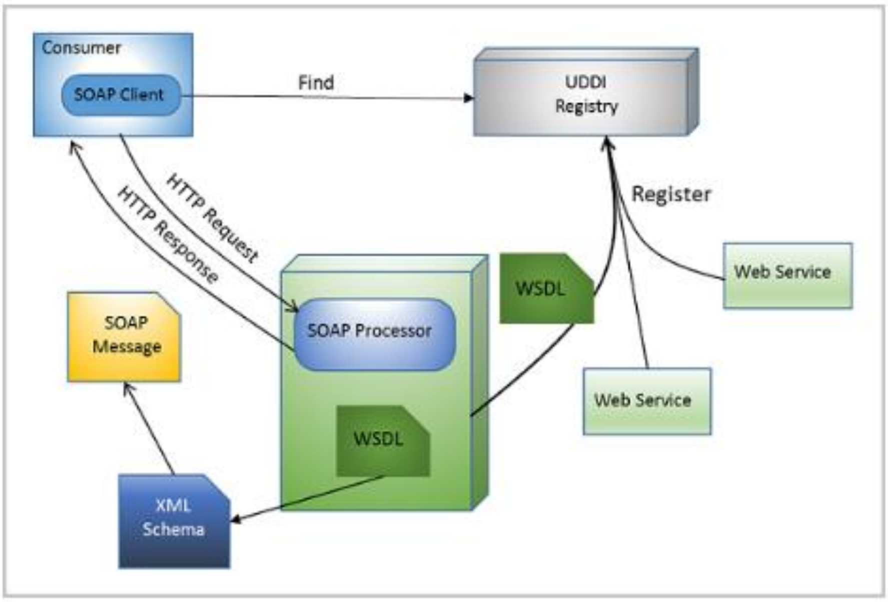


SOAP-based Web Services are **described, discovered and accessed** using the standards that are recommended by W3C.

RESTful Web Services are REST architecture based ones. As per REST architecture, everything is a resource. A resource (data/functionality/web page) is an object that has a specific type and associated data. Also, it has relationships to other resources and a well-defined set of methods that operate on it. RESTful web services are light-weight, scalable, maintainable and leverage the HTTP protocol to the maximum.

2 A. Explain about SOAP based Web Services.

SOAP (Simple Object Access Protocol) defines a very strongly typed messaging framework that relies heavily on XML and schemas. And, SOAP itself is a protocol (transported over HTTP, can also be carried over other transport layer protocol like SMTP/FTP, etc.) for developing SOAP-based APIs.



WSDL

Services are described using **WSDL (Web Services Description Language)**. And, this description is of the XML format.

UDDI

UDDI (Universal Description, Discovery, and Integration) is a registry service which is XML based, where Services can be discovered/registered.

SOAP

SOAP-based Web services are accessed using SOAP (Simple Object Access Protocol), an XML standard for defining the message format for information exchange.

SOAP based Web service - Sample

Here, we will

- Understand the basic concepts of SOAP
- Analyze the SOAP Requests and Responses

Let us look at a sample SOAP-based Web Service which when invoked by sending the name of the user will respond back with a string **"Hello <name>, Welcome to the world of Web Services"**.

This sample code is written in Java using **JAX-WS** API (API that helps to develop SOAP-based Web Services in Java)

GreetingService.java

```
/**
A Sample SOAP based Web Service written in Java using JAX-WS API
*/
package com.infy;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
//Below annotation is used to denote that this class is going to be exposed as a
//Soap based Web Service and also names your Web Service while generation the WSDL
@WebService(serviceName = "GreetingService")
public class GreetingService {
/**
This is a sample web service operation called "hello"
*/
@WebMethod(operationName = "hello")
public String myhello(@WebParam(name = "name") String uname) {
return "Hello " + uname + ",Welcome to the world of Web Services  !";
}
}
```

Here is the WSDL file generated for GreetingService.java using WSDL tool.

```
<?xml version="1.0"?>
<definitions targetNamespace="http://infy.com/" name="GreetingService">
<types>
<schema>
<import namespace="http://infy.com/"
schemaLocation="http://localhost:8080/GreetingService/GreetingService?xsd=1"/>
</schema>
</types>
<message name="hello">
<part name="parameters" element="tns:hello"/>
</message>
<message name="helloResponse">
<part name="parameters" element="tns:helloResponse"/>
</message>
<portType name="GreetingService">
```

```
<operation name="hello">
<input Action="http://infy.com/GreetingService/helloRequest" message="tns:hello"/>
<output Action="http://infy.com/GreetingService/helloResponse" message="tns:helloResponse"/>
</operation>
</portType>
<binding name="GreetingServicePortBinding" type="tns:GreetingService">
<binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
<operation name="hello">
<operation soapAction=""/>
<input>
<body use="literal"/>
</input>
<output>
<body use="literal"/>
</output>
</operation>
</binding>
<service name="GreetingService">
<port name="GreetingServicePort" binding="tns:GreetingServicePortBinding">
<address location="http://localhost:8080/GreetingService/GreetingService"/>
</port>
</service>
</definitions>
```

Sample SOAP Request for GreetingService:

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:hello xmlns:ns2="http://infy.com/">
<name>Rekha</name>
</ns2:hello>
</S:Body>
</S:Envelope>
```

Sample SOAP Response for GreetingService:

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:helloResponse xmlns:ns2="http://infy.com/">
<return>Hello Rekha,Welcome to the world of Web Services  !</return>
</ns2:helloResponse>
</S:Body>
```

```
</S:Envelope>
```

2 B. Explain about RESTful Web Services.

REST-REpresentational State Transfer is an architectural style for developing web services which uses HTTP for communication. The term, **REST** was coined by Roy Fielding in his doctoral dissertation in 2000.

In REST, everything is a **Resource** (data/functionality/web page) that is uniquely identified by URI (Uniform Resource Identifier).

Example:

Resource	**URI**
Data of an employee	http://www.infy.com/employees/john
Web Page	http://www.infy.com/about/infosys.html
Functionality that validates a credit card	http://www.infy.com/creditcards/1000001

The resources of REST can be **represented** in many ways that include JSON, XML, HTML, etc.,

For example, if the URI, http://www.infy.com/employees/john, is hit using HTTP GET method, the server may respond back with John's details in JSON format.

```
{
"name": "John",
"Level": 5,
"address":"Hyderbad",
"phoneno":998765432
}
```

There are possibilities to expose John's details in XML format as well.

```
<employee>
<name>John</name>
<address>hyderabad</address>
<level>5</level>
<phoneno>998765432</phoneno>
</employee>
```

This means the server can represent the resource in many ways.

So now what does **State Transfer** mean?

When we make a request to a resource like http://www.infy.com/employees/john using HTTP, we may be asking for the current "state" of this resource or its desired "state".

We can use the same URI to indicate that, we want to:

- Retrieve the current value of an Employee John (current state)
- Perform Update operation on employee John's data (desired state) etc.

The server will understand which operation has to be invoked based on the HTTP method that was used to make a call.

In short it means that a client and server exchange, representation of a resource, which reflects its current state(GET) or its desired state(PUT).

We will understand more about REST and REST principles in the next set of sections.

Why RESTful Web Services?

We have two types of web services in general. But, why should we go for RESTful web services?

To understand this, let us look at the GreetingService example that we had seen while discussing SOAP-based Web Services. This is how the SOAP request looked like.

Sample SOAP Request to GreetingService:

```
Sample SOAP Request for GreetingService:
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:hello xmlns:ns2="http://infy.com/">
<name>Rekha</name>
</ns2:hello>
</S:Body>
</S:Envelope>
```

This XML will be embedded as "payload" inside the **SOAP** request envelope which in turn will be wrapped using HTTP request and then, will be sent using HTTP POST.

The SOAP Response would be:

```
Sample SOAP Response for GreetingService:
<?xml version="1.0" encoding="UTF-8"?><S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:helloResponse xmlns:ns2="http://infy.com/">
<return>Hello Rekha,Welcome to the world of Web Services  !</return>
</ns2:helloResponse>
</S:Body>
</S:Envelope>
```

If the same functionality is developed as a RESTful Web service, the request will be something like this,

```
GET http://<server name>:<portno>/GreetingService/greetings?name=Rekha
```

And, the response will be like the one that follows.

```
{"response":"Hello Rekha, Welcome to the world of Web Services" }
```

3. A. Explain how to create RESTful Web Services.

RESTful Web services can be developed and consumed in Java using **JAX-RS API** or **Spring REST**.

JAX-RS

OR

{ REST } + spring

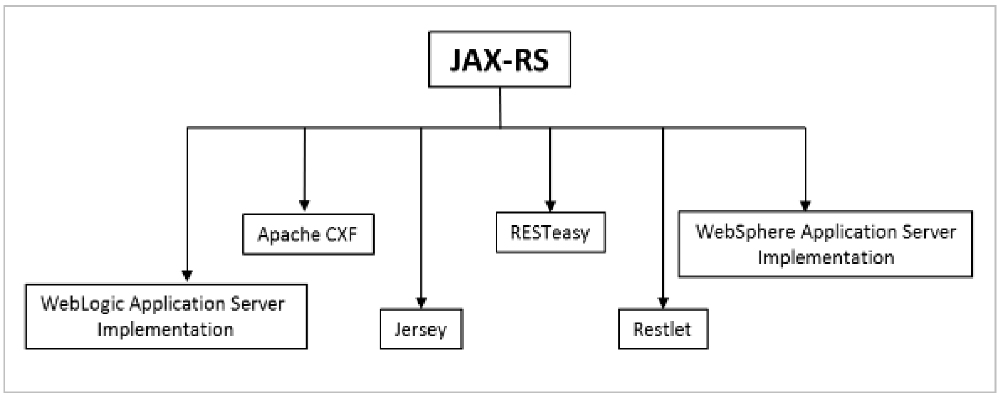
JAX-RS expands to Java API for RESTful Web services.

JAX-RS is a set of specifications to extend support for building and consuming RESTful Web services in Java.
JAX-RS 2.0 and above are annotation-driven which eases the development of Java-based RESTful services.

JAX-RS implementations

JAX-RS is simply a specification and we need actual implementations to write web services. There are many vendors in the market who adhere to the standards of JAX-RS such as, Jersey and RESTEasy.

Below are some of the JAX-RS implementations.


JAX-RS
Apache CXF
RESTeasy
WebSphere Application Server Implementation
WebLogic Application Server Implementation
Jersey
Restlet

Spring REST

Spring is an end to end framework which has a lot of modules in a highly organized way.

One among such modules is, Spring MVC that provides the support for REST in addition to the standard MVC support.

The developers who use Spring MVC can go for constructing RESTful services in an effortless way. Also, it is guaranteed that the application stays lightweight as the build path is not polluted with many external dependencies, just to support REST.

Note: Spring REST is not an implementation of JAX-RS unlike Jersey and RESTEasy.

3 B. Differentiate between SOAP based and RESTful Web Services.

Both SOAP and REST provide support for building applications based on SOA.

Below is the table that summarizes the differences between SOAP and RESTful web services.

SOAP-based Web Service	RESTful Web Service
SOAP is a protocol (defines the way how messages have to be sent) which is transported over a transport layer Protocol, mostly HTTP.	REST is an architectural style.
SOAP messages can be transported using other transport layer protocols (SMTP, FTP) as well.	REST leverages HTTP Protocol.
Data (XML/JSON) wrapped in a SOAP message, comes with an additional overhead of XML processing at the client and server side as well.	REST is straight forward in handling requests and responses. Since REST supports multiple data formats other than XML, there is no overhead of wrapping the request and response data in XML. Also, REST uses the existing HTTP protocol for communication. All these factors together make REST, lightweight.
SOAP-based reads cannot be cached because SOAP messages are sent using the HTTP POST method.	REST-based reads can be cached.
A predefined formal contract is required between the consumer and the provider.	Formal contract is not mandatory. Service description can be done if needed, using a less verbose way with the help of WADL.

UNIT-V

1. **A. Explain the procedure of creating a Spring REST Controller.**

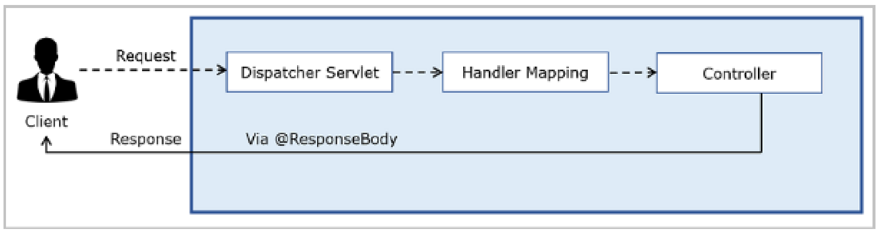
Spring REST Internals

Spring Web MVC module is the source of Spring REST as well.

Working Internals of Spring REST:

Spring REST requests are delegated to the DispatcherServlet that identifies the specific controller with the help of handler mapper. Then, the identified controller processes the request and renders the response. This response, in turn, reaches the dispatcher servlet and finally gets rendered to the client.

Here, ViewResolver has no role to play.



Steps involved in exposing the business functionality as a RESTful web service

What are the steps involved in exposing business functionality as a RESTful web service?

Step 1: Create a REST Resource

Step 2: Add the service methods that are mapped against the standard HTTP methods

Step 3: Configure and deploy the REST application

Since we are going to develop REST applications using Spring Boot, lot of configurations needed in Step-3 can be avoided as Spring Boot takes care of the same.

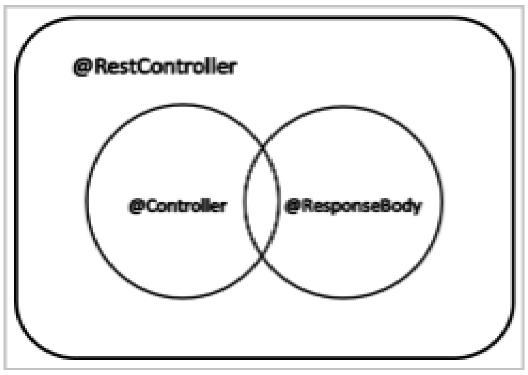
Let us look at each step in detail.

Creating a REST Resource

Any class that needs to be exposed as a RESTful resource has to be annotated with @RestController

@RestController

- This annotation is used to create REST controllers.
- It is applied on a class in order to mark it as a request handler/REST resource.
- This annotation is a combination of @Controller and @ResponseBody annotations.
- @ResponseBody is responsible for the automatic conversion of the response to a JSON string literal. If @Restcontroller is in place, there is no need to use @ResponseBody annotation to denote that the Service method simply returns data, not a view.
- @RestController is an annotation that takes care of instantiating the bean and marking the same as REST controller.
- It belongs to the package, org.springframework.web.bind.annotation.RestController



@RequestMapping

- This annotation is used for mapping web requests onto methods that are available in the resource classes. It is capable of getting applied at both class and method levels. At method level, we use this annotation mostly to specify the HTTP method.

```
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
@RestController
@RequestMapping("/customers")
public class CustomerController
{
@RequestMapping(method=RequestMethod.POST)
public String createCustomer()
{
//Functionality goes here
}
}
```

Adding request handler methods

REST resources have handler methods with appropriate HTTP method mappings to handle the incoming HTTP requests. And, this method mapping usually happens with annotations.

For example, in the Infytel application, developed for a telecom company, we would like to create a REST resource that can deal with operations like creating, deleting, fetching and updating the customers. Here comes the summary of HTTP operations and the corresponding mappings. An important point to be noted here is, Infytel is a Spring Boot application.

URI	**HTTP Method**	**CustomerController method**	**Method description**	**Annotation to be applied at the method level**	**New Annotation that can be applied instead of @RequestMapping at the method level**
/customer	GET	fetchCustomer()	Will fetch all the customers of Infytel App and return the same.	@RequestMapping(method = RequestMethod.GET)	@GetMapping
/customer	POST	createCustomer()	Will create a new customer	@RequestMapping(method = RequestMethod.POST)	@PostMapping
/customer	DELETE	deleteCustomer()	Will delete an existing customer	@RequestMapping(method = RequestMethod.DELETE)	@DeletMapping
/customer	UPDATE	updateCustomer()	Will update the details of an existing customer	@RequestMapping(method = RequestMethod.PUT)	@PutMapping

CustomerController with handler methods

Following is the code of a REST controller that has several handler methods with appropriate method mapping annotations.

```
@RestController
@RequestMapping("/customers")
public class CustomerController
{
//Fetching the customer details
@GetMapping
public String fetchCustomer()
{
//This method will fetch the customers of Infytel and return the same.
return "customers fetched successfully";
}
//Adding a new customer
@PostMapping
public String createCustomer()
{
```

```
//This method will persist the details of a customer
return "Customer added successfully";
}
//Updating an existing customer
@PutMapping
public String updateCustomer()
{
//This method will update the details of an existing customer
return "customer details updated successfully";
}
//Deleting a customer
@DeleteMapping
public String deleteCustomer()
{
//This method will delete a customer
return "customer details deleted successfully";
}
}
```

1 B. Explain about @RequestBody and ResponseEntity in Spring REST.

@RequestBody

@RequestBody is the annotation that helps map our HTTP request body to a Java DTO. And, this annotation has to be applied on the local parameter (Java DTO) of the request method.

Whenever Spring encounters @RequestBody, it takes the help of the registered HttpMessageConverters which will help convert the HTTP request body to Java Object depending on the MIME type of the Request body.

Example: In the below code snippet, the incoming HTTP request body is deserialized to CustomerDTO. If the MIME type of the incoming data is not mentioned, it will be considered as JSON by default and Spring will use the JSON message converter to deserialize the incoming data.

```
@PostMapping
public String  createCustomer( @RequestBody CustomerDTO customerDTO)
{
// logic goes here
}
```

We can specify the expected Mime type using the consumes attribute of the HTTP method matching annotation

Example: The above code is equivalent to the below code that promotes the application of the attribute, consumes.

```
@PostMapping(consumes="application/json")
public ResponseEntity<String> createCustomer( @RequestBody CustomerDTO
customerDTO)
{
```

```
// logic goes here
}
```

consumes attribute can be supplied with a single value or an array of media types as shown below

```
consumes = "text/plain"
          or
consumes = {"text/plain", "application/json"}
Some more valid values:
consumes = "application/json"
consumes = {"application/xml", "application/json"}
consumes = {"text/plain", "application/*"}
```

ResponseEntity

How to send a Java Object in the response?

Scenario-1
Java objects can be returned by the handler method just like how the normal Java methods can return an object.
Example: In the below example, the fetchCustomer() returns a list of Customer Objects. This list will be converted by Spring's message converter to JSON data.

```
@GetMapping
public  List<CustomerDTO> fetchCustomer()
{
//business logic goes here
return customerService.fetchCustomer();
}
```

Scenario-2
We can specify the MIME type, to which the data to be serialized, using the **produces** attribute of HTTP method matching annotations

```
@GetMapping(produces="application/json")
public  List<CustomerDTO> fetchCustomer()
{
//This method will return the customers of Infytel
return customerService.fetchCustomer();
}
```

Just like **consumes**, the attribute, **produces** can also take a single value or an array of MIME types

```
Valid values for produces attribute:
produces = "text/plain"
produces = {"text/plain", "application/json"}
produces = {"application/xml", "application/json"}
```

Setting response using ResponseEntity

While sending a response, we may like to set the HTTP status code and headers .To help achieving this, we can use ResponseEntity class.

ResponseEntity<T> Will help us add a HttpStatus status code and headers to our response.

Example: In the below code snippet, createCustomer() method is returning a String value and setting the status code as 200.

```
@PostMapping(consumes="application/json")
public ResponseEntity<String> createCustomer(@RequestBody CustomerDTO
customerDTO)
{
//This method will create a customer
String response = customerService.createCustomer(customerDTO);
return ResponseEntity.ok(response);
}
```

As shown in the code snippet, we can use the static functions of ResponseEntity class that are available for standard Http codes (ok() method is used, here). One more way of setting response entity is as follows.

```
ResponseEntity(T body, MultiValueMap<String,String> headers, HttpStatus status)
```

Example:

```
@PostMapping(consumes="application/json")
public ResponseEntity<String> createCustomer(@RequestBody CustomerDTO
customerDTO)
{
HttpHeaders responseHeaders = new HttpHeaders();
responseHeaders.set("MyResponseHeaders", "Value1");
String response = customerService.createCustomer(customerDTO);
return new ResponseEntity<String>(response, responseHeaders, HttpStatus.CREATED);
}
```

9. **ResponseEntity - Constructors and Methods**
10. Below is the list of constructors available to create ResponseEntity

Constructor	Description
ResponseEntity(HttpStatus status)	Creates a ResponseEntity with only status code and no body
ResponseEntity(MultiValueMap<String,String> headers, HttpStatus status)	Creates a ResponseEntity object with headers and statuscode but, no body
ResponseEntity(T body, HttpStatus status)	Creates a ResponseEntity with a body of type T and HTTP status
ResponseEntity(T body, MultiValueMap<String,String> headers , HttpStatus status)	Creates a ResponseEntity with a body of type T, header and HTTP status

11. **Note**: There are some ResponseEntity methods available as well.

ResponseEntity method	Description
ok(T body)	Method that creates a ResponseEntity with status, ok and body, T
ResponseBuilder badRequest()	Returns a ResponseBuilder with the status, BAD_Request. In case, body has to be added, then body() method should be invoked on the ResponseBuilder being received and finally build() should get invoked in order to build ResponseEntity. **Usage Example:**

ResponseEntity method	Description
	ResponseEntity.badRequest().body(message).build();
ResponseBuilder notFound()	Returns a ResponseBuilder with the status, NOT_FOUND. ResponseEntity.notFound().build();

2. A. Explain about handling of URI data using Parameter Injection concept through

i. @PathVariable ii. @RequestParam iii. @MatrixVariable

Path Variables

- Path variables are usually available at the end of the request URIs delimited by slash (/).
- @Pathvariable annotation is applied on the argument of the controller method whose value needs to be extracted out of the request URI.
- A request URI can have any number of path variables.
- Multiple path variables require the usage of multiple @PathVariable annotations.
- @Pathvariable can be used with any type of request method. For example, GET, POST, DELETE, etc.,

We have to make sure that the name of the local parameter (int id) and the placeholder ({id}) are same. Name of the PathVariable annotation's argument (@PathVarible("id")) and the placeholder ({id}) should be equal, otherwise.

```
@GetMapping("/{id}")
public String controllerMethod(@PathVariable int id){}
```

OR

```
@GetMapping("/{id}")
public String controllerMethod(@PathVariable("id") int empId){}
```

Example: Assume, there is a REST controller called CustomerController that has service methods to update and delete the customers. These operations are simply based out of the phone number of the customer that is passed as part of the request URI.

Below are the sample URIs that contain data as part of them.

```
URI: PUT:http://<<hostname>>:<<port>>/<<contextpath>>/customers/9123456789
DELETE:http://<<hostname>>:<<port>>/<<contextpath>>/customers/9123456789
```

The handler methods with the provision to extract the URI parameters are presented below.

```
@RestController
@RequestMapping("/customers")
public class CustomerController
{
//Updating an existing customer
@PutMapping(value = "/{phoneNumber}", consumes = "application/json")
public String updateCustomer(
@PathVariable("phoneNumber") long phoneNumber,
@RequestBody CustomerDTO customerDTO) {
//code goes here
}
// Deleting a customer
@DeleteMapping(value="/{phoneNumber}",produces="text/html")
public String deleteCustomer(
@PathVariable("phoneNumber") long phoneNumber)
throws NoSuchCustomerException {
//code goes here
}
}
```

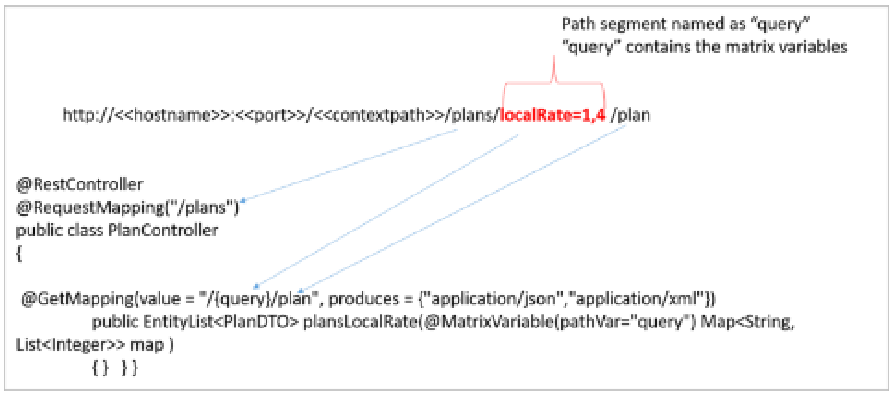
Matrix variables:

- Matrix variables are a block/segment of values that travel along with the URI. For example, **/localRate=1,2,3/**
- These variables may appear in the middle of the path unlike query parameters which appear only towards the end of the URI.
- Matrix variables follow **name=value** format and use semicolon to get delimited from one other matrix variable.
- A matrix variable can carry any number of values, delimited by commas.
- @MatrixVariable is used to extract the matrix variables.

Example: Assume, there is a REST controller called PlanController that has a service to return the details of Plans based on the search criteria, localRates.
Below is the URI path. Observe that a variable localRate with two values separated by "," appear in the middle of the path.

```
URI:http://<<hostname>>:<<port>>/<<contextpath>>/plans/localRate=1,4 /plan
```

See how the matrix variable, **localRate** gets extracted in the controller method.


Path segment named as "query"
"query" contains the matrix variables

```
http://<<hostname>>:<<port>>/<<contextpath>>/plans/localRate=1,4 /plan

@RestController
@RequestMapping("/plans")
public class PlanController
{

 @GetMapping(value = "/{query}/plan", produces = {"application/json","application/xml"})
            public EntityList<PlanDTO> plansLocalRate(@MatrixVariable(pathVar="query") Map<String,
List<Integer>> map )
            {}  }}
```

Code:

```
@RestController
@RequestMapping("/plans")
public class PlanController
{
//{query} here is a place holder for the matrix variables that travel in the URI,
//it is not mandatory that the client URI should hold a string literal called query
@GetMapping(value = "/{query}/plan", produces = {"application/json","application/xml"})
public EntityList<PlanDTO> plansLocalRate(
@MatrixVariable(pathVar="query") Map<String, List<Integer>> map ) {
//code goes here
}
}
```

@MatrixVariable(pathVar="query") Map<String, List<Integer>> map :The code snippet mentions that all the matrix variables that appear in the path segment of name **query** should be stored in a Map instance called **map.** Here, the map's key is nothing but the name of the matrix variable and that is nothing but **localRate**. And, the **value** of the map is a collection of **localRates (1,4)** of type Integer.

Note:If the matrix variable appears towards the end of the URI as in the below example,

```
1.   URI:http://localhost:8081/infytel-1/customers/calldetails/phoneNo=9123456789
```

we can use the following approach to read the matrix variable.

```
1.   @GetMapping("/customers/{query}")
2.   public ResponseEntity<CallDetails> getCallDetails(@MatrixVariable String phoneNo)
3.   {
4.   //code goes here
```

}

Request Param:

Demo 4 - Reading Request Parameters using @RequestParam

Objectives: To learn how to read query strings from the URI.

To create a Spring REST application using Spring Boot where the REST endpoint works with request parameters. Here, we will learn the

- Usage of @RequestParam

Scenario: An online telecom app called Infytel is exposing its customer management profile as a RESTful service. The application has a customer resource titled CustomerController allows us to create, fetch, delete and update customer details. This application has two more controllers, CallDetailsController and PlanController to deal with call and plan details respectively. This demo has the following HTTP operations.

Controller Class	Method Name	URI	HTTP Method	Remarks
CallDetailsController	fetchCallDetails()	/calldetails?calledBy=&calledOn=	GET	This fetches the calls made by a customer on a certain date.Uses @RequestParam to read query string values passed
CustomerController	createCustomer()	/customers	POST	To create a new customer
CustomerController	fetchCustomer()	/customers	GET	To fetch all the existing customers
CustomerController	updateCustomer()	/customers/{phoneNumber}	PUT	To update a customer details
CustomerController	deleteCustomer	/customers/{phoneNu	DELETE	To delete an

Controller Class	Method Name	URI	HTTP Method	Remarks
		mber}		existing customer

3. A. Explain about Exception Handling Mechanism in Spring REST.

Handling exceptions will make sure that the entire stack trace is not thrown to the end-user which is very hard to read and, possesses a lot of security risks as well. With proper exception handling routine, it is possible for an application to send customized messages during failures and continue without being terminated abruptly.

In simple words, we can say that exception handling makes any application robust.

@ExceptionHandler is a Spring annotation that plays a vital role in handling exceptions thrown out of handler methods(Controller operations). This annotation can be used on the

- Methods of a controller class. Doing so will help handle exceptions thrown out of the methods of that specific controller alone.
- Methods of classes that are annotated with @RestControllerAdvice. Doing so will make exception handling global.

The most common way is applying @ExceptionHandler on methods of the class that is annotated with @RestControllerAdvice. This will make the exceptions that are thrown from the controllers of the application get handled in a centralized way. As well, there is no need for repeating the exception handling code in all the controllers, keeping the code more manageable.

Example - Exception with no handler method

Example: An online telecom app called Infytel exposes its functionalities as RESTful resources. And, CustomerController is one among such resources that deal with the customers of Infytel. CustomerController has a method to delete a customer based on the phone number being passed.

Below is the code snippet for the same.

```
@RestController
@RequestMapping("/customers")
public class CustomerController
{
// Deleting a customer
@DeleteMapping(value = "/{phoneNumber}", produces = "text/html")
public String deleteCustomer(
@PathVariable("phoneNumber") long phoneNumber)
throws NoSuchCustomerException {
// code goes here
}
}
```

What if the phoneNo does not exist and the code throws NoSuchCustomerException?

If no Exception handler is provided, Spring Boot provides a standard error message as shown below.

```
{
"timestamp": "2019-05-02T16:10:45.805+0000",
"status": 500,
"error": "Internal Server Error",
"message": "Customer does not exist :121",
"path": "/infytel-7/customers/121"
}
```

Example - Exception with handler method

In case, a customized error message that is easy to understand has to be provided, we need to

- Create a class annotated with @RestControllerAdvice
- Have methods annotated with @ExceptionHandler(value=NameoftheException) which takes the exception class as the value for which the method is the handler

We can have multiple methods in the Advice class to handle exceptions of different types and return the custom messages accordingly.

Example: Below is the RestControllerAdvice class that holds a method which handles NoSuchCustomerException by returning a customized error message.

```
@RestControllerAdvice
public class ExceptionControllerAdvice {
@ExceptionHandler(NoSuchCustomerException.class)
public ResponseEntity<String> exceptionHandler2(NoSuchCustomerException ex) {
return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
}
}
```

Custom Error Message

It is important to map the exceptions to objects that can provide some specific information which allows the API clients to know, what has happened exactly. So, instead of returning a String, we can send an object that holds the error message and error code to the client back.

Let us look at an example:

The object which is going to hold a custom error message is **ErrorMessage** with two variables, errorcode and message.

```
public class ErrorMessage {
private int errorCode;
private String message;
//getters and setters go here
}
```

Now, the RestControllerAdvice will be modified as shown below.

```
@RestControllerAdvice
public class ExceptionControllerAdvice {
@ExceptionHandler(NoSuchCustomerException.class)
public ResponseEntity<ErrorMessage> exceptionHandler2(NoSuchCustomerException ex) {
ErrorMessage error = new ErrorMessage();
error.setErrorCode(HttpStatus.BAD_GATEWAY.value());
error.setMessage(ex.getMessage());
return new ResponseEntity<>(error, HttpStatus.OK);
}
}
```

Here, the handler method returns the instance of **ErrorMessage** that holds the error code and message, rather returning a String as the body of ResponseEntity.

3 B. Explain the Data Validation technique in Spring REST.

Some times, the RESTful web service might need data in certain standard that should pass through specific constraints.

What if the data is not in the format as needed by the RESTful service?

By default, Spring Boot provides some implementation for validation

- 415 - Unsupported Media Type : If the data is sent in a different format other than what is expected(eg:XML instead of JSON)
- 400 - Bad Request : If JSON data is not in proper format, for example.

Some times, we might send valid format and structure, but with missing data. Still, the request gets processed. In such situations, it becomes even more important to validate the data.
Example:

```
{
//empty JSON data
}
```

Hibernate Validator, is one of the implementations of the bean validation API. Let us see, how to use the Validation API to validate our incoming data.

Bean Validation API provides a number of annotations and most of these annotations are self explanatory.

- Digits
- Email
- Max
- Min
- NotEmpty
- NotNull
- Null
- Pattern

Applying validation on the incoming data

Let us apply validation on the customer data that comes to the createCustomer() method of CustomerController. Infytel expects the name field not to be null and the email field to be of proper email format.

```
public class CustomerDTO {
long phoneNo;
@NotNull
String name;
@Email(message = "Email id is not in format, please check")
String email;
int age;
char gender;
List<FriendFamilyDTO> friendAndFamily;
String password;
String address;
PlanDTO currentPlan;
// getter and setters go here
}
```

Have a look at the usage of @NotNull and @Email on the fileds, name and email respectively.

Adding the constraints simply on the bean will not carry out validation. In addition, we need to mention that validation is required while the JSON data is deserialized to CustomerDTO and get the validation error, if any, into org.springframework.validation.Errors object. This can be done by applying @Valid annotation on the handler method argument which captures the CustomerDTO object as shown below.We can inject the Errors object too into this method.

```
@PostMapping(consumes="application/json")
public ResponseEntity createCustomer(@Valid @RequestBody CustomerDTO customerDTO,
Errors errors)
{
String response = "";
if (errors.hasErrors())
{
response  = errors.getAllErrors().stream()
.map(ObjectError::getDefaultMessage)
.collect(Collectors.joining(","));
ErrorMessage error = new ErrorMessage();
error.setErrorCode(HttpStatus.NOT_ACCEPTABLE.value());
error.setMessage(response);
return ResponseEntity.ok(error);
}
else
{
response = customerService.createCustomer(customerDTO);
return ResponseEntity.ok(response);
}
}
```

Let us decode the above snippet.

Discussion on how the validation takes place

@Valid @RequestBody CustomerDTO customerDTO - indicates that CustomerDTO should be validated before being taken.

Errors - indicates the violations, if any, to get stored in Errors object.

Finally, the business logic will be divided into two courses of actions. That is, what should be done when there are no validation errors. And, what should be done, otherwise.

If there are errors: Collect all the errors together as a String.

Set the String that contains errors in the message field and appropriate error code in the errorCode field of ErrorMessage.

Finally, return the ResponseEntity that is set with the ErrorMessage object.

```
if (errors.hasErrors())
{
response  = errors.getAllErrors().stream().
map(x->x.getDefaultMessage()).
collect(Collectors.joining(","));
ErrorMessage error = new ErrorMessage();
error.setErrorCode(HttpStatus.NOT_ACCEPTABLE.value());
error.setMessage(response);
return ResponseEntity.ok(error);
}
```

The ErrorMessage class:

```
public class ErrorMessage {
private int errorCode;
private String message;
//Getters and Setters goes here
}
```

2. If there are no errors, invoke the service layer to create the customer and return the response back to the client.

```
response = customerService.createCustomer(customerDTO);
return ResponseEntity.ok(response);
```

Let us put all these together and look at an example.

Rich set of annotations

We are done dealing with validating the incoming objects/DTOs. Also, we finished traversing a set of annotations that impose validation on the individual fields/attributes of the DTO.

Apart from the annotations that we discussed, we have few other interesting annotations in place. For example,
@PastOrPresent
@Past
@Future
@FutureOrPresent
@DateTimeFormat - part of **org.springframework.format.annotation** unlike other annotations being mentioned here which are part of standard **javax.validation.constraints**

These annotations can only be applied on the fields/attributes of type, date/time, for example, LocalDate/LocalTime.

One more interesting annotation that needs a mention here is **@Positive** which is used to make sure a number should possess only positive values for example, cost of an article.

Last but not the least to be brought in for discussion is **@NotBlank** which makes sure that the string is not null and the trimmed length should be greater than zero.

Further Reading : Explore javax.validations.constraints API to get the complete list of validation annotations that can be applied on the fields/attributes.

URI parameter validation

Like how it is important to validate the incoming objects/DTOs, it is essential to validate the incoming URI parameters as well. For example, when the customer details need to be deleted based on the phone number and if the same is reaching the REST endpoint as URI parameter (path variable, for example), it becomes essential that the URI parameter, phone number should also be validated (should be of 10 digits, for example).

The code snippet given below helps serving the purpose discussed above.

```
public String deleteCustomer(@PathVariable("phoneNumber")
@Pattern(regexp = "[0-9]{10}",message="{customer.phoneNo.invalid}")
String phoneNumber) throws NoSuchCustomerException
```

Here, the message that should be rendered during the validation failure need not be hard-coded always. It can be fetched from the external property file as well and the name of the file is ValidationMessages.properties that Spring Boot searches for. If the file name is other than ValidationMessages.properties, MessageSource and LocalValidatorFactoryBean need to be configured in the bootstrap class.

One important thing that needs attention while validating the URI parameters is, **@Validated.**

The controller where the URI parameter validation is carried out should be annotated with @Validated. The validation on URI parameters will not be triggered, otherwise. Here, the CustomerController has validations pertaining to phoneNo that is received as URI parameter (DELETE and UPDATE). So, @Validated annotation needs to be applied on the same as follows.

```
@Validated
public class CustomerController
```

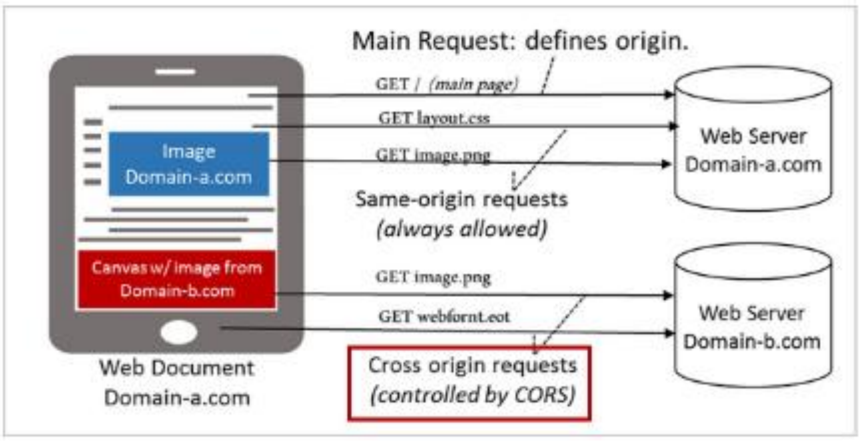
4. A. Explain the procedure of enabling CORS in Spring REST.

Why CORS?

Typically, scripts running in a web application of one origin (domain) cannot have access to the resources of a server available at a different origin. If the clients of different origins try to access the REST endpoints of an application, CORS error will be thrown simply. And, below is a sample of the same.

```
Cross-Origin Request Blocked: The Same Origin Policy disallows
reading the remote resource at https://domain-1.com
```

Look at the image below, which depicts the request for resources by the clients of the same origin and different origins as well.


Main Request: defines origin.
GET / (main page)
GET layout.css
GET image.png
Web Server Domain-a.com
Image Domain-a.com
Same-origin requests (always allowed)
Canvas w/ image from Domain-b.com
GET image.png
GET webfornt.eot
Web Server Domain-b.com
Web Document Domain-a.com
Cross origin requests (controlled by CORS)

Request for resources from a different domain will encounter CORS error.

To overcome this error, we need to enable cross-origin requests at the server-side so that the browsers which run the client code can allow scripts to make a Cross-origin call.

Let us see, how can it be done in Spring REST.

How to enable CORS?

Cross-origin calls can be enabled for Spring REST resources in three ways by applying:

- CORS configuration on the REST controller methods
- CORS configuration on the REST controller itself
- CORS configuration globally (common to the entire application)

CORS configuration on controller methods:

In this style, we need to apply @CrossOrigin on the handler method.

Here, the annotation, @CrossOrigin enables cross-origin requests only for the specific handler method.

By default, it allows all origins, all headers, all HTTP methods specified in the @RequestMapping annotation. Also, a lot more other defaults are available.

We can customize this behavior by specifying the value for the following attributes that are available with Http method mapping annotations (@GetMapping, @PostMapping, etc.,)

- origins - list of allowed origins to access the method
- methods - list of supported HTTP request methods
- allowedHeaders - list of request headers which can be used during the request
- exposedHeaders - list of response headers which the browser can allow the clients to access
- allowCredentials - determines whether the browser can include cookies that are associated with the request

Example:

```
@Controller
@RequestMapping(path="/customers")
public class CustomerController
{
@CrossOrigin(origins = "*", allowedHeaders = "*")
@GetMapping()
public String homeInit(Model model) {
```

```
return "home";
}
}
```

How to enable CORS?
CORS configuration on the controller:
It is possible to add @CrossOrigin at the controller level as well, to enable CORS for all the handler methods of that particular controller.

```
@Controller
@CrossOrigin(origins = "*", allowedHeaders = "*")
@RequestMapping(path="/customers")
public class CustomerController
{
@GetMapping()
public String homeInit(Model model) {
return "home";
}
}
```

Global CORS configuration:

As an alternative to fine-grained, annotation-based configuration, we can also go for global CORS configuration as well.

Look at the code below. The starter class has been modified to implement WebMvcConfigurer and override addCorsMappings() method that takes the CorsRegistry object as argument using which we can configure the allowed set of domains and HTTP methods as well.

```
@SpringBootApplication
public class InfytelApplication implements WebMvcConfigurer {
public static void main(String[] args) {
SpringApplication.run(InfytelDemo8BApplication.class, args);
}
@Override
public void addCorsMappings(CorsRegistry registry) {
registry.addMapping("/**").allowedMethods("GET", "POST");
}
}
```

In our demo, we will enable CORS at the global level.

4 B. Explain the concept of versioning a Spring REST endpoint.

Versioning a service becomes important when we want to create a new version of an already existing service. But at the same time, we need to support the earlier version as well, to ensure backward compatibility.

There are different ways of achieving API versioning such as

- URI versioning
- **Request Parameter Versioning**
- Custom Header Versioning

- Accept Header versioning

Let us take a look at each one of them.
Note: Request Parameter Versioning is used in the demos

URI versioning

URI versioning involves different URI for every newer version of the API.
Example: There is a need for two versions of functionality that fetches the plan details of Infytel telecom application. And, this fetch operation is based on the planId being passed as a path variable.

version-1: Fetches the complete plan details that include plainId, planName, localRate and nationalRate
version-2: Fetches only the localRate and nationalRate.
Below is the URI proposed to be used

```
GET http://localhost:8080/infytel/plans/v1/{planId}
eg:http://localhost:8080/infytel/plans/v1/1
GET http://localhost:8080/infytel/plans/v2/{planId}
eg:http://localhost:8080/infytel/plans/v2/1
```

Now, let us look at how the handler method can be written.

For Version-1

```
@GetMapping(value = "v1/{planId}")
public ResponseEntity<PlanDTO> getPlan(@PathVariable("planId")String planId)
{
//Return the complete plan details
}
```

For Version-2

```
@GetMapping(value = "v2/{planId}")
public ResponseEntity<PlanDTO> getPlanv2(@PathVariable("planId")String planId)
{
//Returns only the localRate and nationalRate
}
```

Observe, how a functionality with two different versions or approaches is made available in a REST resource. Also, focus on how the consumers call the different versions of the API method using the URIs that carry information about versioning.

Versioning with Custom Headers

This type of versioning requires the inclusion of custom headers in the request URI to map to the correct version of the API endpoint.

Let us take the service that fetches plan details, here as well.

Below is the URI proposed to be used

```
GET http://localhost:8080/infytel/plans/{planId}
headers=[X-API-VERSION=1]
eg:http://localhost:8080/infytel/plans/1  and pass the header information headers=[X-API-VERSION=1]
GET http://localhost:8080/infytel/plans/{planId}
```

```
headers=[X-API-VERSION=2]
eg:http://localhost:8080/infytel/plans/1 and pass the header information headers=[X-API-VERSION=2]
```

Now, let us look at how the handler method can be written

For Version-1

```
@GetMapping(value = "{planId}",headers = "X-API-VERSION=1")
public ResponseEntity<PlanDTO> getPlan(@PathVariable("planId")String planId)
{
//Return the complete plan details
}
```

For Version-2

```
@GetMapping(value = "{planId}",headers = "X-API-VERSION=2")
public ResponseEntity<PlanDTO> getPlanv2(@PathVariable("planId")String planId)
{
//Returns only the localRate and nationalRate
}
```

Observe, how a functionality with two different versions or approaches is made available in a REST resource. Also, focus on how the consumers call the different versions of the API method using the URIs that carry information about versioning as part of the request headers.

Accept Header Versioning

Accept Header Versioning approach is one other way of versioning the API methods that insists the usage of Accept Header in the request.

```
GET http://localhost:8080/infytel/plans/{planId}
headers=[Accept=application/vnd.plans.app-v1+json]
eg:http://localhost:8080/infytel/plans/1  and pass the header information headers=[Accept=application/vnd.plans.app-v1+json]
GET http://localhost:8080/infytel/plans/{planId}
headers=[Accept=application/vnd.plans.app-v2+json]
eg:http://localhost:8080/infytel/plans/1 and pass the header information headers=[Accept=application/vnd.plans.app-v2+json]
```

Now, let us look at how the handler method can be written

For Version-1

```
@GetMapping(value = "/{planId}", produces = "application/vnd.plans.app-v1+json")
public ResponseEntity<PlanDTO> getPlan(@PathVariable("planId")String planId)
{
//Return the complete plan details
}
```

For Version-2

```
@GetMapping(value = "/{planId}", produces = "application/vnd.plans.app-v2+json")
public ResponseEntity<PlanDTO> getPlanv2(@PathVariable("planId")String planId)
{
```

```
//Returns only the localRate and nationalRate
}
```

Observe, how a functionality with two different versions or approaches is made available in a REST resource. Also, focus on how the consumers call the different versions of the API method using the URIs that carry information about versioning as part of the request headers named, **accept**. One more important thing to be noted here is the inclusion of custom MIME type.

Request Parameter versioning

Request parameter versioning can be achieved by providing the request parameter in the URI that holds the version details.

```
GET http://localhost:8080/infytel/plans/{planId}?version=
eg:http://localhost:8080/infytel/plans/1?version=1
GET http://localhost:8080/infytel/plans/{planId}?version=
eg:http://localhost:8080/infytel/plans/1?version=2
```

Now, let us look at how the handler method can be written

For Version-1

```
@GetMapping(value = "/{planId}", params = "version=1")
public ResponseEntity<PlanDTO> getPlan(@PathVariable("planId")String planId)
{
//Return the complete plan details
}
```

For Version-2

```
@GetMapping(value = "/{planId}", params = "version=2")
public ResponseEntity<PlanDTO> getPlanv2(@PathVariable("planId")String planId)
{
//Returns only the localRate and nationalRate
}
```

Observe, how a functionality with two different versions or approaches is made available in a REST resource. Also, focus on how the consumers call the different versions of the API method using the URIs that carry information about versioning in the form of request parameter.

5. A. Explain the procedure of creating a REST Client.

There are situations where a Spring REST endpoint might be in need of contacting other RESTful resources. Situations like this can be hadled with the help of REST client that is available in the Spring framework. And, the name of this REST client is RestTemplate.

RestTemplate has a wonderful support for standard HTTP methods.
The methods of the RestTemplate need to be provided with the absolute URI where the service can be found, input data, if any and, the response type that is expected out of the service.

Calling a RESTful service of HTTP request method Get:

The following code snippet shows how to consumes a RESTful service exposed by url :http://localhost:8080/infytel/customers using HTTP GET.

```
RestTemplate restTemplate = new RestTemplate();
String url="http://localhost:8080/infytel/customers";
List<CustomerDTO> customers = (List<CustomerDTO>) restTemplate.getForObject(url,
List.class);
for(CustomerDTO customer:customers)
{
System.out.println("Customer Name: "+customer.getName());
System.out.println("Customer Phone: "+customer.getPhoneNo());
System.out.println("Email Id: "+customer.getEmail());
}
```

Calling a RESTful service of HTTP request method Post:

URI of the service:http://localhost:8080/infytel/customers

```
CustomerDTO customerrequest= new CustomerDTO();
//populate CustomerDTO object
//....
String url="http://localhost:8080/infytel/customers"
RestTemplate restTemplate = new RestTemplate();
ResponseEntity<String> response =
restTemplate.postForEntity(url,customerrequest,String.class);
System.out.println("status code :" + response.getStatusCodeValue());
System.out.println("Info       :" + response.getBody());
```

Similarly, RESTful services of various other HTTP methods can be called using RestTemplate.

5 B. Explain about Securing Spring REST endpoints.

Securing a Spring Rest API With Spring Security

Need for securing a RESTful service arises when the authorized entities alone have to access to the end points.

Securing REST APIs here, follows the same steps involved in securing any other web application that is developed using Spring Boot. As we all know, Spring Boot minimizes configuration and that goes good with security as well.

We can secure the Spring REST endpoints using,

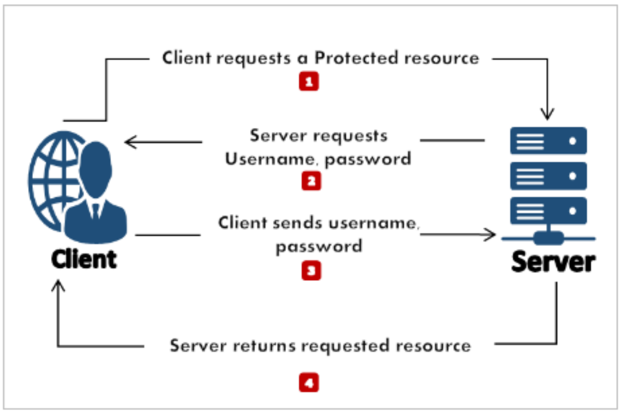
- **Basic Authentication** (one that will be used in our course)
- OAUTH2 authentication

Basic Authentication:

When client requests for a protected resource, server demands the client to supply the authentication credentials.

Client will get the requested resource only when the credentials pass authentication at server side.

Steps involved in basic authentication are shown below.


Client requests a Protected resource
1
Server requests Username, password
2
Client sends username, password
3
Server returns requested resource
4
Client
Server

With Basic Authentication in place, clients send their Base64 encoded credentials with each request using HTTP Authorization header.

This type of authentication does not mandate us to have any Login page which is the very standard way of authenticating the client requests in any web application. In addition, there is no need to store the credentials or any related info in the session.

Each request here, is treated as an independent request which in turn promotes scalability.

Our example for implementing security will follow this approach.

Enabling Basic Authentication

Enabling security in a Spring Boot application needs the inclusion of the following dependency as the very first step.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

This will include the SecurityAutoConfiguration class – containing the initial or default security configuration.

By default, authentication gets enabled for the whole application.

There are some predefined properties as shown below which needs to be configured in the application.properties file.

- spring.security.user.name
- spring.security.user.password

If there is no password configured using the predefined property **spring.security.user.password,** Spring Boot generates a password in a random fashion during the application start. And, the same will be found in the console log.

```
Using default security password: c8be15de-4488-4490-9dc6-fab3f9
```

Following steps are required only when we need to customize the security related aspects.

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
}
```

Need to override the below methods

```
//Configuring username and password and authorities
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
auth.inMemoryAuthentication()
.withUser("admin").password(passwordEncoder().encode("infytel"))
.authorities("ROLE_USER");
}
//Configuring the HttpSecurity
@Override
protected void configure(HttpSecurity http) throws Exception {
http.authorizeRequests()
.antMatchers("/securityNone").permitAll()//URLS starting with securityNone need not be authenticated
.anyRequest().authenticated() //rest of the URL's should be authenticated
.and().
httpBasic().and()//Every request should be authenticated it should not store in a session
sessionManagement().sessionCreationPolicy(SessionCreationPolicy.NEVER).and().csrf().disable();
}
//The passwordEncorder to be used.
@Bean
public PasswordEncoder passwordEncoder() {
return new BCryptPasswordEncoder();
}
```