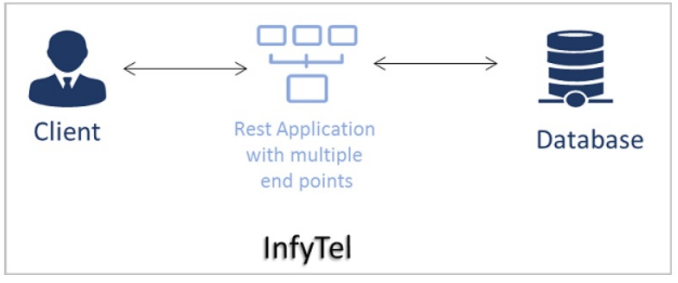
Spring 5 basics with Spring boot

Preledge



InfyTel is built as a three-tier application which consists of

- Presentation Layer
- Service/Business Layer
- Persistence Layer

✓ built service layer in this course

Class which will be there in our applicatn.

Below are the classes used in the InfyTel application.

- CustomerService.java -> Interface to define service methods
- CustomerServiceImpl.java -> A service class which implements the CustomerService interface
- Client.java-> A class for the main method.
- CustomerRepository.java-> A class for the persistence layer where all CRUD operations are performed.

Introduction to Spring framework

Why Spring ?

1)

```java
package com.infy.service;

public interface CustomerService {

    public String fetchCustomer();

    public String createCustomer(CustomerDto dto)

}
```

```java
public class CustomerServiceImpl implements CustomerService{

CustomerRepository customerRepository= new CustomerRepositoryImpl();

    public String createCustomer(CustomerDto dto) {
        return customerRepository.createCustomer(dto);

    }

    public String fetchCustomer() {
        return customerRepository.fetchCustomer();
    }
}
```

responsible to interact with database

creating a new object that's why tightly coupled and difficult for unit testing

2)

```java
public class CustomerServiceImpl implements CustomerService {

    private CustomerRepository customerRepository;

    public CustomerServiceImpl(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

     public String createCustomer(CustomerDto dto) {
        return customerRepository.createCustomer(dto);

    }
}
```

not creating a new object now so loosely coupled but difficult to wire all together with dependencies (that's why we use DI) dependency injection technique

Note :- The reversal of repositories is known as Inversion of Control (IoC)

What is Spring framework

Spring Framework is an open source Java application development framework that supports developing all types of Java applications such as enterprise applications, web applications, cloud based applications, and many more.

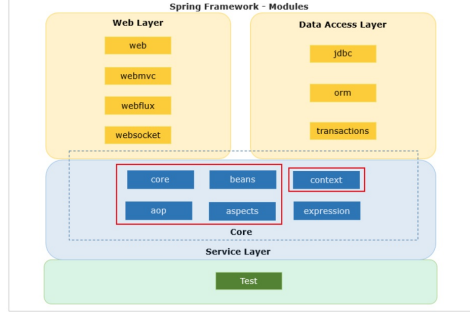
Java applications developed using Spring are simple, easily testable, reusable, and maintainable.

Spring modules do not have tight coupling on each other, the developer can pick and choose the modules as per the need for building an enterprise application.

- light weight
- non invasion
- loosly coupled
- IoC
- Spring container
- AoP

⇒ Spring framework has these core modules

- Core Container: These are core modules that provide key features of the Spring framework.
- Data Access/Integration: These modules support JDBC and ORM data access approaches in Spring applications.
- Web: These modules provide support to implement web applications.
- Others: Spring also provides few other modules such as the Test for testing Spring applications.



(big picture)

⇒ Spring Inversion of Control (IoC)

↳ IoC helps in creating more loosely coupled app^n

↳ Spring framework provide IoC implementat^n using Dependency Injection

Note :- ↳ Spring Container managed applications objects are called Beans

↳ no need to create object in DI, instead describe how object should be created through configuration.

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.

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 Provide 2 types of Containers

✓ BeanFactory: (supports lazy loading)

- 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: (supports eager loading)

- 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.

There are different ways to access bean in Spring

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

```
1. 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

```
1. CustomerServiceImpl service =  context.getBean(CustomerServiceImpl.class);
2.
```

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

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

Spring allows providing the configuration metadata using :

- XML Configuration
- Annotation Based configuration
- Java Based configuration

→ Java based configurat^n metadata is provided using
@Configuration
@Bean

```
@Configuration
public class SpringConfiguration {

    @Bean(name="service")
    public CustomerServiceImpl customerService() {

        return new CustomerServiceImpl();
    }
}
```

default the name of bean same as function name

→ Spring Bean

Spring Bean is nothing special, any object in the Spring framework that we initialize through Spring container is called Spring Bean. Any normal Java POJO class can be a Spring Bean if it's configured to be initialized via container by providing configuration metadata information.

⇒ Dependency Injection

In DI a developer need not to create object but specify how they should be created through configuration.

Spring container uses one of these two ways to initialize the properties:

- **Constructor Injection**: This is achieved when the container invokes a parameterized constructor to initialize the properties of a class
- **Setter Injection**: This is achieved when the container invokes setter methods of a class to initialize the properties after invoking a default constructor.

Constructor Injection

```java
package com.infy.service;

public class CustomerServiceImpl implements CustomerService {
    private int count;

    public CustomerServiceImpl(int count) {
        this.count = count;
    }

}
```

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

We can do this for more arguments as well

```java
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();
    }

}
```

no new object value is initialised bcz Spring is going to take care of that.

Spring is taking care about object initialization.

```java
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);
    }
}
```

⇒ Setters Injection

```java
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(){
    }
}
```

```java
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;
    }

}
```

```java
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;
    }
}
```

```java
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;
    }
}
```

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.

⇒ Auto Scanning

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();
    }
}
```

But Spring provide autoscanning to detect the beans to be injected and avoid even bean definition.

```
@Configuration
@ComponentScan(basePackages="com.infy")
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.

What are **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.

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

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

2) @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
}
```

(business layer)

3) @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
}
```

(persistance layer)

4) @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
}
```

(presentation layer)

⇒ Introduction to Spring boot

Sometimes spring boot reduce productivity and increase time bcz of
↳ Configuration
↳ Project dependency management

Ques What is spring boot ?

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

The main Spring Boot features are as follows:

- Starter Dependencies
- Automatic Configuration
- Spring Boot Actuator
- Easy-to-use Embedded Servlet Container Support

→ Spring boot Starter Parent dependecy
↳ handles configurations
↳ handles dependencies
↳ handles default plugin configurations

→ Project Structure

1. pom.xml

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

```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.


```java
@SpringBootApplication
public class DemoSpringBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoSpringBootApplication.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 the following annotations with their default attributes:

↳ @EnableAutoConfiguration
↳ @ComponentScan
↳ @Configuration

⇒ Autowiring and Scope of the bean

Ques What is autowiring ?

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.

- ↳ @Autowired annotation performs byType autowiring means dependency is injected based on bean type.
- ↳ @Autowired on setter methods are called setter injection.
- ↳ @Autowired on constructor are called constructor injection.
- ↳ When more than 1 bean is available to be injected then use @Qualifier
- ↳ We can assign a default value to a class property using @Value annotation.

→ **Scope of bean**

- ↳ Lifetime of bean depends on scope of bean
- ↳ 2 types
  - (default) ↳ singleton :- a single bean is created and will be provided to every bean request
  - ↳ Prototype :- a new bean instance is created for every bean request

Implimented using @Scope annotation

- ↳ @Scope("singleton")
- ↳ @Scope("prototype")

⇒ Logger

What is logging?

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

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

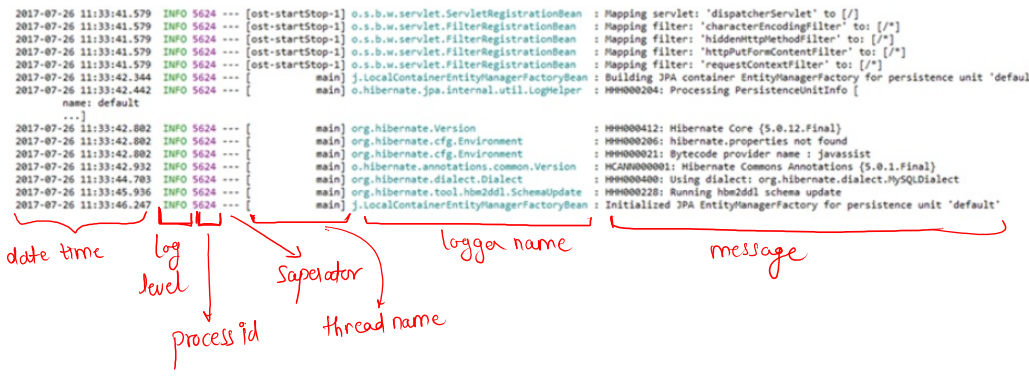
Level	Description
ALL	For all the levels (including user defined levels)
TRACE	Informational events
DEBUG	Information that would be useful for debugging the application
INFO	Information that highlights the progress of an application
WARN	Potentially harmful situations
ERROR	Errors that would permit the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

(levels of logger)

Note :-

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

Structure of logger consoled output by default



⇒ AOP (aspect oriented programming)

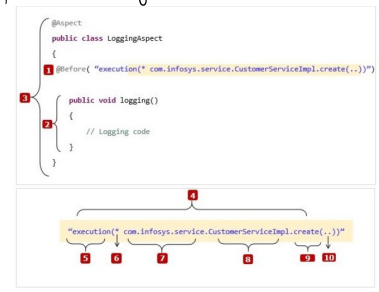
↳ It is an approach to programming that allows global properties of program to determine how it is compiled into an executable program.

↳ AOP is used for applying common behaviours like transaction, security, logging etc. to the applicatn.

↳ Aspect is a class that impliments cross cutting concern.

↳ Pointcut Represents an expression that evaluates the method name before and after which the advice needs to be executed.

→ AOP terminologies



Pointcut expressions have the following syntax:

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

not mandatory (default public)

optional

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.

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

AOP saperates the repeated logic

Authentication
Transaction
Logging

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

Note:-
- In Spring AOP, you 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.

Note:- dependency required

<spring-boot-starter-aop>

Note:- advice execution

↳ @Before (before actual method)

↳ @After (after actual method even if method throws an exception)

↳ @AfterReturning (after actual method only if method doesn't throws an except")

↳ @AfterThrowing (after actual method only if method throws an except")

↳ @Around (executed before and after the execution of target method)

@Pointcut

@Pointcut:

Pointcut represents an expression that evaluates the method name where advice should be executed. If you want to use single pointcut in multiple places, you can declare the pointcut using **@pointcut** annotation in one common class or in the sa me class and make use of it in different aspect classes and methods.

Declaring a pointcut:

A pointcut declaration has below parts:

1) Pointcut Signature
2) Pointcut Expression

ex:-

```
@Pointcut("execution(* com.infosys.service.CustomerServiceImpl.fetchCustomer(..))") //pointcut expression
public void logFetchCustomer() {    } // pointcut signature
```

↳ return type must be void

Pointcut Designators:

A pointcut expression starts with pointcut designators, which tells you where the pointcut will be executed. The designators are:

1) **execution**: used for matching any method name.(you have already seen demo for execution designator).
2) **within** : used for any method present under the class or package .
3) **this:** used for matching to the join points when the object reference created for the given type(refers to the proxy) .
4) **target** – used for matching to the join points when the target object is created for the given type(
5) **args** – used for matching to join points when the arguments datatype is object of the given types.
6) **@target** – used for matching to join points when the given annotation in class level.
7) **@within** – used for matching to join points when the target object is within the types of given annotation.
8) **@annotation** – used for matching to join points when the given annotation in method level.

⇒ Spring Profiles

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.

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.

@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.

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

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

→ Different ways to set profile(s) as active

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

→ in application.properties

```
spring.profiles.active=dev
```

Notes:-

- Java does not allow annotations placed on interfaces to be inherited by the implemented class so make sure that you place the spring annotations only on class, fields, or methods.
- As a good practice place @Autowired annotation before a constructor.
- 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.
  - While defining the scope of the beans choose wisely.

If you 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 is a good practice to use constructor injection for mandatory dependencies and Setter injection for optional dependencies.