



INTRODUCTION TO SPRING BOOT

Introduction

- **Spring**
 - An application development **framework** and **inversion of control** container for Java
- **Spring Boot**
 - **Makes it easy to create** stand-alone, production-grade **Spring Applications** and expose them **as services**

Introduction

- **Micro services**
 - **Break large system into number of independent** collaborating components.
- **Spring Cloud**
 - *Built on top of Spring Boot*
 - Provides a set of features to **quickly build and Deploy** Micro services with **minimal configurations**.

Software Requirement

- **Java 8.x or above**
- **Eclipse IDE with Spring Tool Suite or Spring Tool Suite**
- **Spring Distribution 5.X and Boot 2.1**
- **MySQL or Oracle RDBMS**
- **Tomcat Server**

Spring Framework

- Light-weight **comprehensive framework for building Java SE and Java EE applications**
- Created by **Rod Johnson**
- **Java Bean-based configuration** management,
- Uses **Inversion-of-Control principles**, specifically using the **Dependency Injection** technique

Key Features

- Integration with **persistence** frameworks.
- **MVC** web application framework
- **Aspect-oriented programming** (AOP) framework
- Publishing REST API's

History of Spring Framework

- The Spring Framework was first released in 2004
- **Spring 2.0**
 - provided XML namespaces and AspectJ support;
- **Spring 2.5**
 - embraced annotation-driven configuration;
- **Spring 3.0**
 - Support for Java 5+ across the framework codebase,
 - Java-based @Configuration model.

History of Spring Framework

- **Spring 4.0**
 - Fully support Java 8 features.
 - The minimum requirement is Java SE 6
- **Spring 5.0**
 - Full compatibility with JDK 9 for development and deployment.

IoC CONTAINER

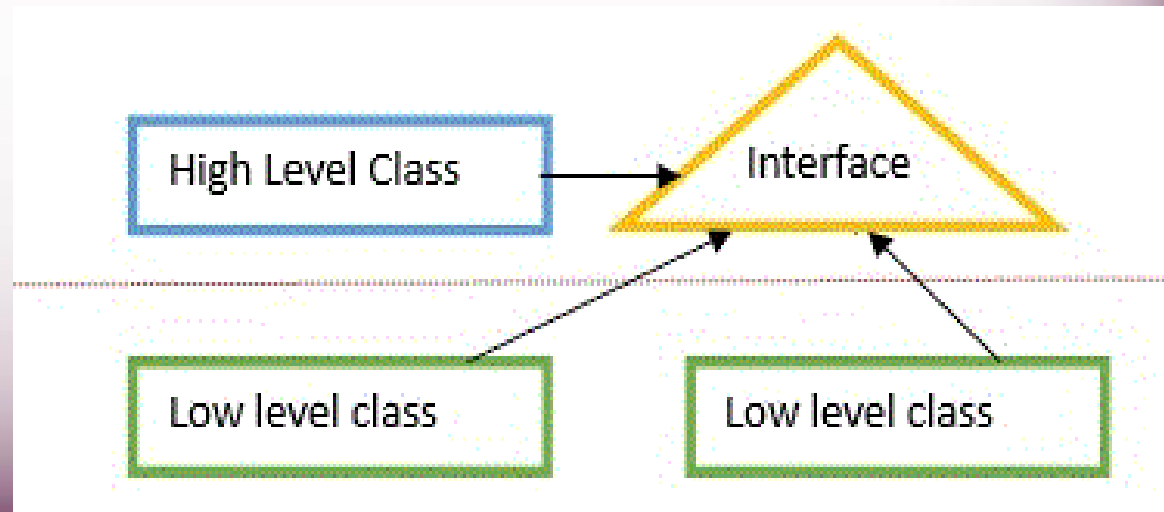
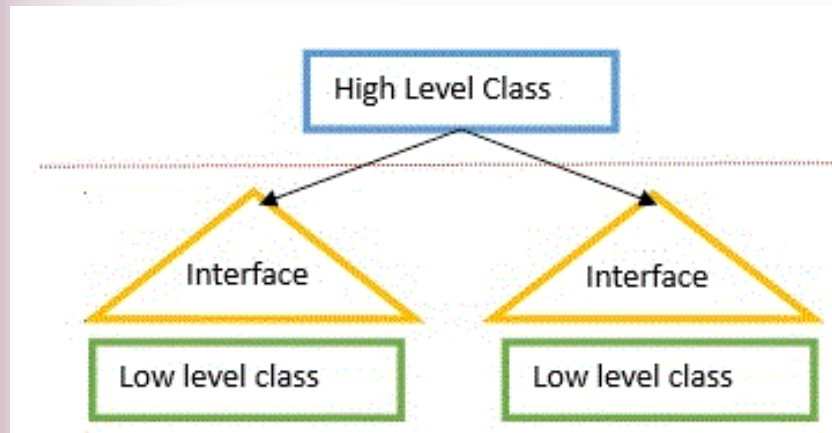
Dependency Injection

- A pattern of injecting a class's dependencies at runtime.
 - Achieved by defining the dependencies and then injecting in a concrete class
- Can swap in different implementations without having to modify the main class.
- Promotes **high cohesion** by promoting the **Single Responsibility Principle (SRP)** and **low coupling**.

Dependency Inversion Principle

- High level modules should not depend upon low level modules.
- Low and High Modules should depend upon abstractions.
- **Low modules should NOT** define a interface that Higher Level module depend on
- **Higher Level Module should** define an interface that lower level module must implement
- This gives the flexibility at the cost of increased effort.

Dependency Inversion Principle



Dependency Inversion Principle

```
public class Messaging {  
    MessageService service;  
    public void sendMessage(){  
        service.sendMessage();  
    }  
}
```



```
public interface MessageService {  
    public String sendMessage();  
}
```



```
public class Email implements MessageService {  
    public String sendMessage() { }  
}
```

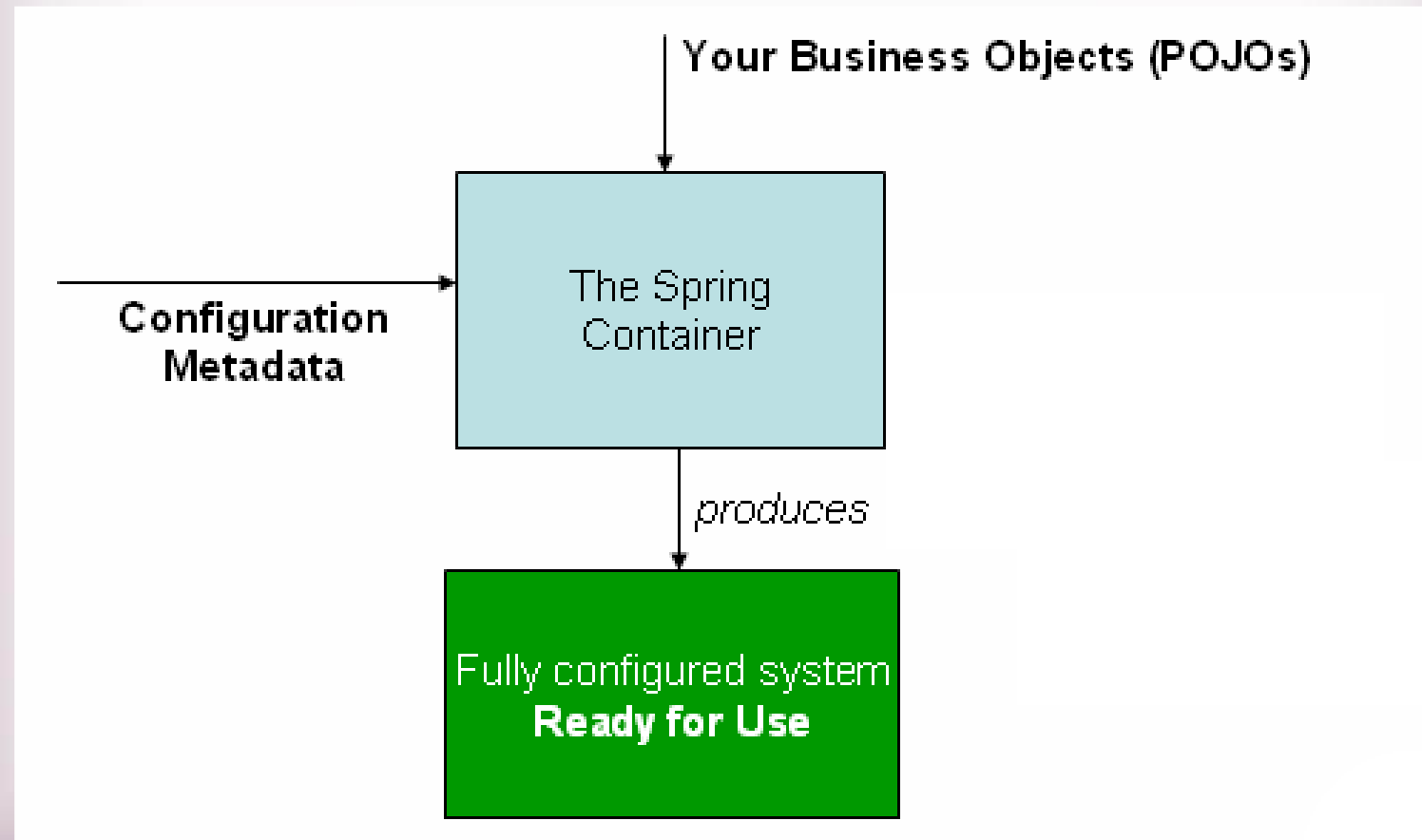


- High-level modules should not depend on low-level modules.
- Both should depend on abstractions.

Inversion of Control Container(IoC)

- A container that supports Dependency Injection.
- No Need to create objects but **describe how** they should be created.
- **Describe which services** are needed by which components in a configuration file.
- Objects are given their **dependencies at creation time** by the IoC
- **IoC** - inversion of responsibility with regard to **how an object obtains references** to collaborating objects.

The Container



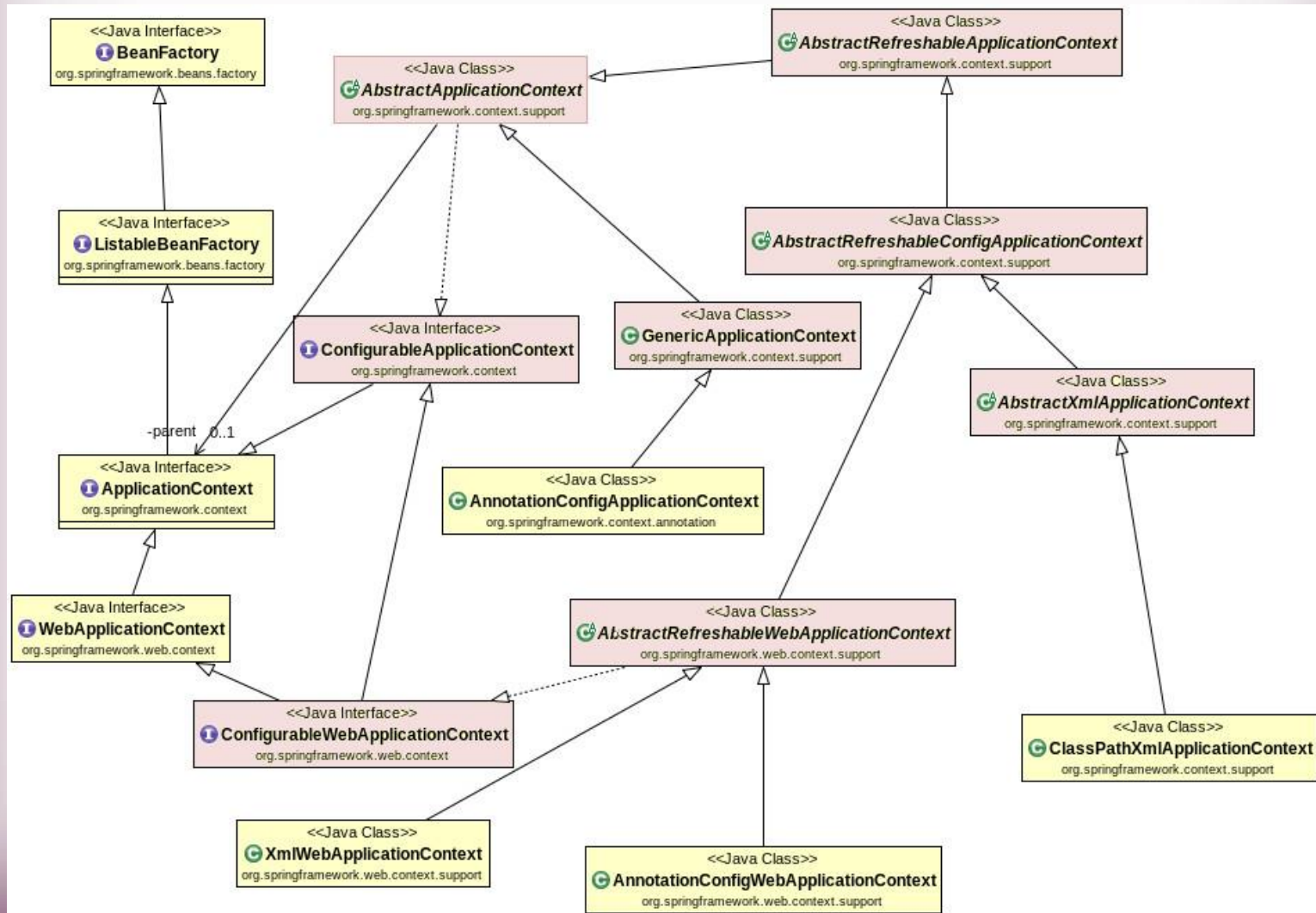
Container and Beans

- **org.springframework.beans.factory.BeanFactory** is the representation of the Spring IoC *container*
 - containing and managing beans,
 - instantiating or sourcing application objects
 - configuring such objects
 - assembling the dependencies between objects.
- **A bean** is an object that is instantiated, assembled, and otherwise **managed by a Spring IoC container**.
- Beans, and the *dependencies among them*, are reflected in the **configuration metadata used by a container**

Application Context

- **ApplicationContext** is a complete superset of the BeanFactory
- Containers gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata.
- There are many Implementations of the ApplicationContext

Container Hierarchy



Beans

- The term “bean” is used to refer any **component managed** by the **BeanFactory**
- The “beans” are in the form of JavaBeans (in most cases)
 - ***Present in a Named Package***
 - ***No Argument constructor***
 - ***getter and setter methods for the properties***
 - ***Can be controlled for scope, life cycle and callbacks.***
- **Beans are singletons** by default
- Properties the beans may be simple values or references to other beans

Configuration Meta Data

- Instructs IoC Container to instantiate, configure, and assemble the objects
- Can be done in **XML format**
 - Supported from Spring 2.0
- Can be done with **Annotation**
 - Supported from Spring 2.5
- Can be done with Just **Java**
 - Supported from Spring 3.0

Spring Boot

- Extension of the Spring framework
 - **A faster and more efficient development**
- Eliminates the boilerplate configurations required for setting up a Spring application.
 - Opinionated 'starter' dependencies to simplify build and application configuration
 - Embedded server to avoid complexity in application deployment
 - Metrics, Health check, and externalized configuration
 - Automatic configuration whenever possible

Spring Initializr

- <https://start.spring.io/>
 - **A web-based UI tool** provided by the Pivotal
 - Used to generate the structure of the **Spring Boot Project**.
 - Can configure the list of dependencies
 - Package downloaded as **Jar** or **War** file
 - Import as a Maven Project into STS

Spring Boot Starters

- Set of convenient dependency descriptors
- Eliminates the need to hunt through sample code
- No need to copy-paste loads of dependency descriptors.
- Contains lot of the dependencies that are required to get a project up and running
- Also has support of managed transitive dependencies.

Creating a Spring Boot Project

- In the Eclipse IDE Right-click in the package explorer and select New -> Spring Starter Project
- A screen opens with some details which can be changed or can accept the default values for a simple project
- **The entry point of a Spring Boot application is the class which is annotated with `@SpringBootApplication`:**
- Uses this class with *public static void main* entry-point to launch an embedded web server.

DEPENDENCY INJECTION

Dependency Injection

- Spring could inject dependencies between the beans by following way of injection types:
 - Setter Injection
 - Constructor Injection
 - Field Injection (@Autowired at field)
 - Method Injection
- Dependency can be wired in three ways
 - XML Based Configuration
 - Annotation Based Configuration
 - Java Based Configuration

Setter Injection

- All dependencies injected via setter methods
 - No need for constructors
- Allows flexible initialization
 - Any set of properties can be initialized
- Named properties allow for readable configuration data
 - Clear what is being injected
- Requires Java Bean conventions to be followed
 - Non standard method names are problematic
- Can result in invalid object configurations
 - Need post initialization checking methods

Constructor Dependency Injection

- Container invokes a class constructor with a number of arguments, each representing a dependency on the other class
- Dependency can be one of the following
 - primitive and String-based values
 - Dependent object (contained object)
 - Collection values
- *Constructor is annotated with @Autowired*
 - Can omit from Spring 4.3 there is single constructor

Constructor-based or setter-based DI?

- Constructor arguments for mandatory dependencies and setters for optional dependencies.
- Constructor dependency , disadvantage is that the object becomes less amenable to reconfiguration and re-injection.
- A legacy class may not expose any setter methods, and so constructor injection is the only available DI.

REGISTERING BEANS

@ Component

- Bean are Managed by IoC Container.
- **@Component**
 - Spring can scan all the beans through **auto scan if the class has this or similar annotation**
- By default the First character of the component is lower Cased
 - **‘CustomerService’ to ‘CustomerService’.**
- CustomerService cust =
(CustomerService)context.getBean("customerService");

@ComponentScan

@Configuration

@ComponentScan("com.training")

```
public class Config {  
  
}
```

- Class with @Component are candidates for component scanning.
- If required during component scanning components will be autowired together.

Java Based Configuration

- **XML is completely orthogonal to the Java**
- **Centralized Configuration is centralized, can also split**
- **Simplified autowiring.**

Configuration File

A Java Class analogues to bean.xml

- **@Configuration**
 - indicates that the class can be used by the Spring IoC container as a source of bean definitions.
- **@Bean**
 - Method with this annotation will return an object that should be registered as a bean in the Spring application context.

@Configuration

```
public class EmployeeConfig {
```

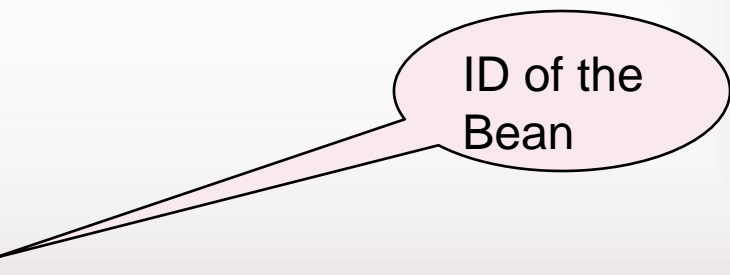
```
    @Bean
```

```
    public Employee employee() {
```

```
        return new Employee();
```

```
    }
```

```
}
```



ID of the
Bean

@Bean

- Method-level annotation
- Supports following attributes
 - init-method
 - destroy-method
 - autowiring
- Can be used in Class annotated with
 - @Configuration
- **By default, the bean name is the same as the method name.**

Customizing Bean Naming

- @Bean method's name is the name of the resulting bean.
- It Can be overridden with the name attribute

```
@Bean(name = "malar")
```

```
public Hospital hospital() {
```

```
    return new Hospital();
```

```
}
```

```
}
```

Bean Dependencies

- Bean-annotated method can have an arbitrary number of parameters
 - Each Parameter describe the dependencies required to build that bean.

@Bean()

public Catalog catalog() {

return new Catalog(101,"Classical Menu",address());

}

@Bean

public Address address() {

**return new Address("Gandhi Street","behru
Nagar","chennai",600040);**

}

Configuration File

- **@Configuration**
 - indicates that the class can be used by the Spring IoC container as a source of bean definitions.
- Classes annotated are bootstrapped using
 - AnnotationConfigApplicationContext
 - AnnotationConfigWebApplicationContext.

DEMO

Tourist Guide

@Data

@AllArgsConstructor

@NoArgsConstructor

public class TouristGuide {

private long id;

private String guideName;

private long mobileNumber;

}

Tour

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Tour {

private long tourId;

private String tourName;

private int duration;

private double price;

private TouristGuide guide;

}

Configuration

@Configuration

public class AppConfig {

// Setter DI

@Bean

public Tour lanka() {

Tour lanka = new Tour();

lanka.setTourId(101);

lanka.setPrice(23000);

lanka.setTourName("Superb Srilanka");

lanka.setDuration(12);

return lanka;

}

Configuration

// Constructor DI

@Bean

public Tour thailand() {

// Passing Reference of One Bean to Another Bean - Done Using
ref in earlier Versions

return new Tour(102,"Beautiful Bali",5,52000,swapnil());
}

@Bean

public TouristGuide swapnil() {

return new TouristGuide(101,"Swapnil Sawant",98495952);
}
}

Spring Application

```
public static void main(String[] args) {  
  
    ConfigurableApplicationContext ctx =  
        SpringApplication.run(BootlocApplication.class, args);  
  
    Tour lankaTour = ctx.getBean("lanka", Tour.class);  
  
    System.out.println(lankaTour);  
  
    System.out.println(ctx.getBean("thailand", Tour.class));  
  
    ctx.close();  
}
```

SpringApplication

- *SpringApplication*
 - Class to bootstrap a Spring application from a Java main method.
 - Creates an appropriate ApplicationContext instance
 - Registers a CommandLinePropertySource to expose command line arguments as Spring properties,
 - Refreshes the application context, loading all singleton beans, and triggers any CommandLineRunner beans.

Spring Boot Bootstrap

@SpringBootApplication

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

@SpringBootApplication

– Equivalent to using following with their default attributes

- **@Configuration**,
- **@EnableAutoConfiguration**,
- **@ComponentScan**

AUTO CONFIGURATION

Auto Configuration

- Spring Boot takes an **opinionated view of the Spring platform and third-party libraries**
 - Can get started with minimum effort
 - It tries to **read “.properties”** from various hard-coded locations.
 - It also reads the “spring.factories” file
 - Part of auto configure-module
 - Determines the Auto Configurations it should evaluate.

Auto Configuration

- Some Spring Boot Jars contain special JSON meta-data files that the editor looks for
 - These files contain information about the known configuration properties.
- **“spring-boot-autoconfigure-XXX.jar”**
 - META-INF/spring-configuration-metadata.json”.
 - Can find properties like server.port being documented there.

AutoConfigurations

- **Auto-registered @PropertySources**
- Spring Boot will *automatically* register these PropertySources
- It has a default set of property locations that it *always* tries to read
 - command line arguments
 - application.properties inside .jar file etc.

Auto Configurations

- *org.springframework.boot:spring-boot-autoconfigure.*
 - .jar file containing *all* of Spring Boot's magic.
 - Has a file **META-INF/spring.factories**
 - Contains the AutoConfigurations
 - With Many @Conditionals that Spring Boot evaluate on every application

Auto Configurations

- **Enhanced Conditional Support**
- **@Conditional**
 - A low level annotation
- Spring Boot has set of additional @Conditional annotations
- **@ConditionalOnBean(DataSource.class).**
 - The condition is true only if the user specified a DataSource @Bean in a @Configuration.
- **@ConditionalOnExpression("someSpELEExpression").**
 - The condition is true if the SpEL expression is true.
- **@ConditionalOnJava(JavaVersion.EIGHT).**
 - The condition is true if the current Java version is 8.

Conditional

```
public class EmployeeConditional implements Condition {
```

```
    @Override
```

```
    public boolean matches(ConditionContext context,  
        AnnotatedTypeMetadata metadata) {
```

```
        boolean response = true;
```

```
        return response;
```

```
    }
```

```
}
```

- **spring.application.name=Conditional**

Defining Conditions

@Bean

@Conditional(EmployeeConditional.class)

public Developer ramesh() {

return new Developer(102, "Ramesh", "Chennai", "Java",94949);

}

@Bean

@ConditionalOnProperty(name ="spring.application.name",havingValue =
"Conditional")

public Leader suresh() {

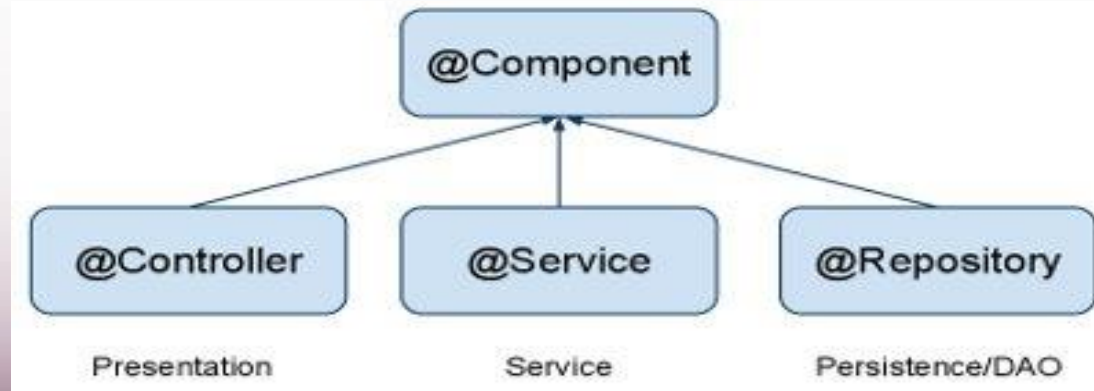
return new Leader(902, "Suresh", "Pune");

}

SPRING STEREOTYPES

Auto Component Scan Types

- **@Component**
 - Indicates a auto scan component.
- **@Repository**
 - Indicates DAO component in the persistence layer.
- **@Service**
 - Indicates a Service component in the business layer.
- **@Controller**
 - Indicates a controller component in the presentation layer.



Auto Wiring of Bean

- **@Autowired :**
 - Used to auto wire bean on the **setter** method, **constructor** or a **field**.
 - It uses auto wire bean by matching data type.
- **@Qualifier**
 - Is used to control which bean should be autowired on a field.
 - If the bean configuration file has two similar beans, can specify the bean to be wired by giving the bean name
- **@Required**
 - . When matching bean to wire is not found , it will throw an exception.
 - To disable this checking setting the “required” to false.
 - it will leave the property unset.

Auto Wiring of Bean

- **@Autowired :**
 - Used to auto wire bean on the **setter** method, **constructor** or a **field**.
 - It uses auto wire bean by matching data type.
- **@Qualifier**
 - Is used to control which bean should be autowired on a field.
 - If the bean configuration file has two similar beans, can specify the bean to be wired by giving the bean name
- **@Required**
 - . When matching bean to wire is not found , it will throw an exception.
 - To disable this checking setting the “required” to false.
 - it will leave the property unset.

Lazy Initialization

- Beans are created as they are needed rather than during application startup.
 - Enabling lazy initialization **May improve the startup time** of the application
 - In a web application, enabling lazy initialization will result in many web-related beans not being initialized until an HTTP request is received.

Lazy Initialization

- It can delay the discovery of a problem with the application.
- If a misconfigured bean is initialized lazily, a failure will no longer occur during startup and the problem will only become apparent when the bean is initialized.
- May reduce the number of beans created when the application is starting

Lazy initialization

- Can be enabled programmatically using Spring Application Builder
- Can be enabled using property from spring boot 2.2
 - **spring.main.lazy-initialization=true**

Can be enabled by using the @Lazy on the Factory Method

SpringApplicationBuilder

- A builder for SpringApplication and ApplicationContext instances
 - Convenient fluent API and context hierarchy support.

ConfigurableApplicationContext ctx=**new**

SpringApplicationBuilder(DemoApplication.class)

.lazyInitialization(false)

.run(args);

Lazy-initialized beans

@Bean()

@Lazy(value=true)

public Employee myBean() {

System.out.println("Loading LazyBean bean");

return new Employee(employee_Id,employee_Name);

}

Disambiguation options

- **getBean()**
 - accepts both a class and a bean name
 - bean ids must be unique, this call guarantees that the ambiguity cannot occur.
- `Service service = context.getBean(Service.class, "myService");`

@Primary

- Used to give higher preference to a bean, when there are multiple beans of same type.
- Used on any class directly or indirectly **annotated** with @Component or on methods **annotated** with @Bean
- If a bean has @Autowired *without* any @Qualifier
 - Multiple beans of the type exist
 - Candidate bean marked @Primary will be chosen
- @Qualifier should be used in conjunction with @Autowired always.
- @Primary should be used in conjunction with @Bean

@Primary

@Bean

@Primary

```
public Service myService() {  
    return new Service();  
}
```

@Bean

```
public Service backupService() {  
    return new Service();  
}
```

getBean(Service.class) will return the primary bean

Life Cycle Methods

@Component

public class Invoice {

@PostConstruct

public void myInit()

{

System.out.println("Inside init Method");

}

@PreDestroy

public void myDestroy()

{

System.out.println("Inside Destroy Method");

}

}

Life Cycle Java Based Configuration

```
public class Employee {
```

```
    public void start() { // Initialization Work    }
```

```
    public void close() {  
        // Destruction Work  
    }  
}
```

```
@Bean(initMethod="start",destroyMethod="close")
```

```
public Employee myBean() {  
    return new Employee(employee_Id,employee_Name);  
}
```

Life Cycle Methods

```
public class DeliveryExecutive {
```

```
    @Autowired
```

```
    private Environment env;
```

```
    public DeliveryExecutive{
```

```
        env.getActiveProfiles() -> Will throw Null Pointer Exception
```

```
    }
```

```
    public void init() {
```

```
        log.info("Init Method called");
```

```
        log.info(env.getActiveProfiles().toString());
```

```
        System.out.println(Arrays.asList(env.getDefaultProfiles().toString()));
```

```
    }
```

```
}
```

Using CRUD Repository

@Repository

public class BookService {

@Autowired

private BookRepository repo;

public Iterable<Book> getBooks(){
 return repo.findAll();
}

public Book addBook(Book book) {
 return repo.save(book);
 }
}

Properties

- **spring.datasource.url=jdbc:mysql://localhost:3306/test**
- **spring.datasource.username=root**
- **spring.datasource.password=srivatsan**

- **spring.datasource.hikari.connection-timeout=20000**
- **spring.datasource.hikari.minimum-idle=5**
- **spring.datasource.hikari.maximum-pool-size=12**
- **spring.datasource.hikari.idle-timeout=300000**
- **spring.datasource.hikari.max-lifetime=1200000**

- **spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect**
- **spring.jpa.properties.hibernate.id.new_generator_mappings = false**
- **spring.jpa.properties.hibernate.format_sql = true**

- **spring.jpa.hibernate.ddl-auto: update**

- **logging.level.org.hibernate.SQL=DEBUG**
- **logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE**

Spring Boot

- Extension of the Spring framework
 - **A faster and more efficient development**
- Eliminates the boilerplate configurations required for setting up a Spring application.
 - Opinionated 'starter' dependencies to simplify build and application configuration
 - Embedded server to avoid complexity in application deployment
 - Metrics, Health check, and externalized configuration
 - Automatic configuration whenever possible

Spring Boot

- Spring Boot 2.2.x RELEASE
- Requires Java 8 and is compatible up to Java 11
- Spring Framework 5.1.5.RELEASE or above
- Supports the Tomcat 9.0 ,Jetty 9.4

Spring Boot Starters

- Set of convenient dependency descriptors
- Eliminates the need to hunt through sample code
- No need to copy-paste loads of dependency descriptors.
- Contains lot of the dependencies that are required to get a project up and running
- Also has support of managed transitive dependencies.

Creating a Spring Boot Project

- In the Eclipse IDE Right-click in the package explorer and select New -> Spring Starter Project
- A screen opens with some details which can be changed or can accept the default values for a simple project
- **The entry point of a Spring Boot application is the class which is annotated with `@SpringBootApplication`:**
- Uses this class with *public static void main* entry-point to launch an embedded web server.

Spring Boot Bootstrap

@SpringBootApplication

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

- Spring Boot can scan all the classes in the same package or sub packages of Main-class for components.
- Equivalent to using **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan** with their default attributes,

LOMBOK

Introduction

- A java library that automatically plugs into editor and build tools
- Helps reduce the boiler plate code.
- Has various annotations which can be used within our code that is be processed during the compile time and appropriate code expansion would take place based on the annotation used.
- Does the code reduction in view time, after the compiling the byte code is injected with all the boiler plate.
- This helps keeping our codebase small, clean and easy to read and maintain.

Adding Lombok to Eclipse

- Downloaded from <https://projectlombok.org/download>
- Execute command in terminal: `java -jar lombok.jar`
- Will Open a window requesting Eclipse Destination
- Copy lombok.jar into Eclipse Parent Directory directory.
- Add the following command to end of Eclipse.ini
 - `-javaagent:lombok.jar .`
- Restart Eclipse and enable “Annotation Processing”

Lombok Annotation

- To generate Setters and Getters
 - @Getter
 - @Setter
- To generate constructors,
 - @NoArgsConstructor
 - @RequiredArgsConstructor
- Other Annotations
 - @ToString
 - @EqualsAndHashCode
 - @Data
 - @Slf4j
-

SPRING BEAN SCOPES

Setter Dependency Injection

@Configuration

public class AppConfig {

@Bean

public Hospital malar() {

Hospital malar = **new Hospital();**

malar.setId(101);

malar.setHospitalName("Malar Hospitals");

malar.setPhoneNumber(24482727L);

return malar;

}

}

Constructor Dependency Injection

```
@Bean(name = "apollo")
```

```
public Hospital apollo() {
```

```
    return new Hospital (102,"Apollo Hospital",4747472);
```

```
}
```

Bean Dependencies

- Bean-annotated method can have an arbitrary number of parameters
 - Each Parameter describe the dependencies required to build that bean.

@Bean()

public Catalog catalog() {

return new Catalog(101,"Classical Menu",address());

}

@Bean

public Address address() {

**return new Address("Gandhi Street","behru
Nagar","chennai",600040);**

}

Lazy-initialized beans

- ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process.
- A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

```
@Bean()
```

```
@Lazy(value=true)
```

```
public Employee myBean() {
```

```
    System.out.println("Loading LazyBean bean");
```

```
    return new Employee(employee_Id,employee_Name);
```

```
}
```

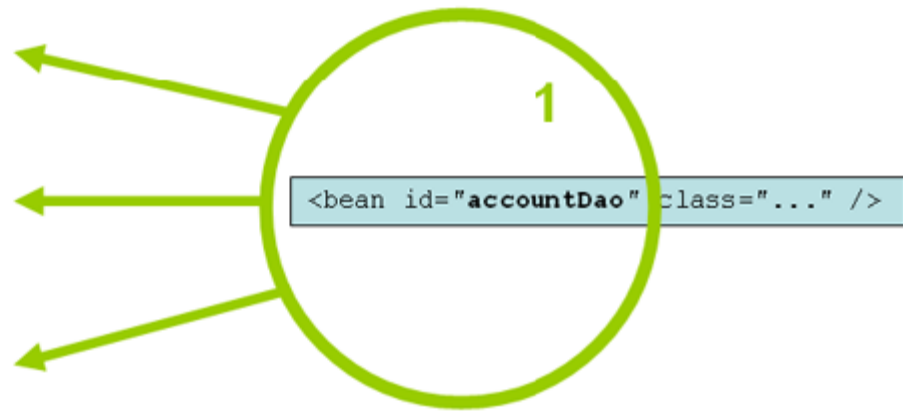
Singleton

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

Only one instance is ever created...



... and this same shared instance is injected into each collaborating object

Prototype Scope

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

A brand new bean instance is created...

1

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

2

```
<bean id="accountDao" class="..."  
  scope="prototype" />
```

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

3

... each and every time the prototype is referenced by collaborating beans

Bean Scope

- The Bean Scope for Standard Java SE Beans are singleton:
- **@Scope(scopeName=ConfigurableBeanFactory.SCOPE_SINGLETON)**
 - **Scopes a single bean definition to a single object instance per Spring IoC container, per container and per bean.**
 - **Single instance will be stored in a cache and all subsequent requests and references will result in cached object being returned.**
- **@Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE)**

Bean Scope

- The Bean Scope for Standard Java SE Beans are singleton:
- **Scopes a single bean definition to a single object instance per Spring IoC container, per container and per bean.**
- ***prototype:***
 - Prototype results in the creation of a new bean instance every time a request for that specific bean is made
 - Can change the Scope by
- **@Scope(value=ConfigurableBeanFactory.SCOPE_PROTOTYPE)**

Test the Scopes

```
DiscountService service = ctx.getBean(DiscountService.class);
```

```
DiscountNotification protoBean;
```

```
protoBean = service.getDiscount("april");
```

```
log.info("Discount :="+protoBean.showDiscount());
```

```
protoBean = service.getDiscount("may");
```

```
log.info("Discount :="+protoBean.showDiscount());
```

METHOD INJECTION

Method Injection

- Solution to the problem of injecting different scoped beans.
- Used for dynamically overriding a class and its abstract methods to create instances every time the bean is injected.
- Used in very special cases as it involves byte-code manipulation by Spring.
- Singletons are instantiated at context creation, it changes the way Spring uses the CGILIB to change the Bytecode.

Method Injection

- A Singleton Bean “A” which has a dependency of non singleton (prototype) bean “B” .
- Container will create only one instance of bean “A” thus will have only one opportunity to inject the prototype bean “B” in it and every time you makes a call to get bean B from bean A , always the same bean will be returned.

Using Method Injection

- Lookup()
- **A method annotated with `@Lookup` tells Spring to return an instance of the method's return type when we invoke it.**
- Spring will override annotated method and use method's return type and parameters as arguments to *BeanFactory#getBean*.
-

Prototype Bean

```
public class MyPrototype {
```

```
String name;
```

```
public int hashCode() {
```

```
return super.hashCode();
```

```
}
```

```
}
```

Singleton

```
public class MyRefinedSingleton {
```

```
    @Autowired
```

```
    MyPrototype proto;
```

```
    @Lookup
```

```
    public MyPrototype getProto() {
```

```
        log.info("Lookup Called");
```

```
        return proto;
```

```
    }
```

```
    public void setProto(MyPrototype proto) {
```

```
        this.proto = proto;
```

```
    }
```

```
}
```


Configuration

@Bean

@Scope(scopeName = ConfigurableBeanFactory.**SCOPE_SINGLETON**)

public MyRefinedSingleton refOne() {

log.info("Refined Singleton Bean Initialized");

return new MyRefinedSingleton();

}

@Bean

@Scope(scopeName = ConfigurableBeanFactory.**SCOPE_PROTOTYPE**)

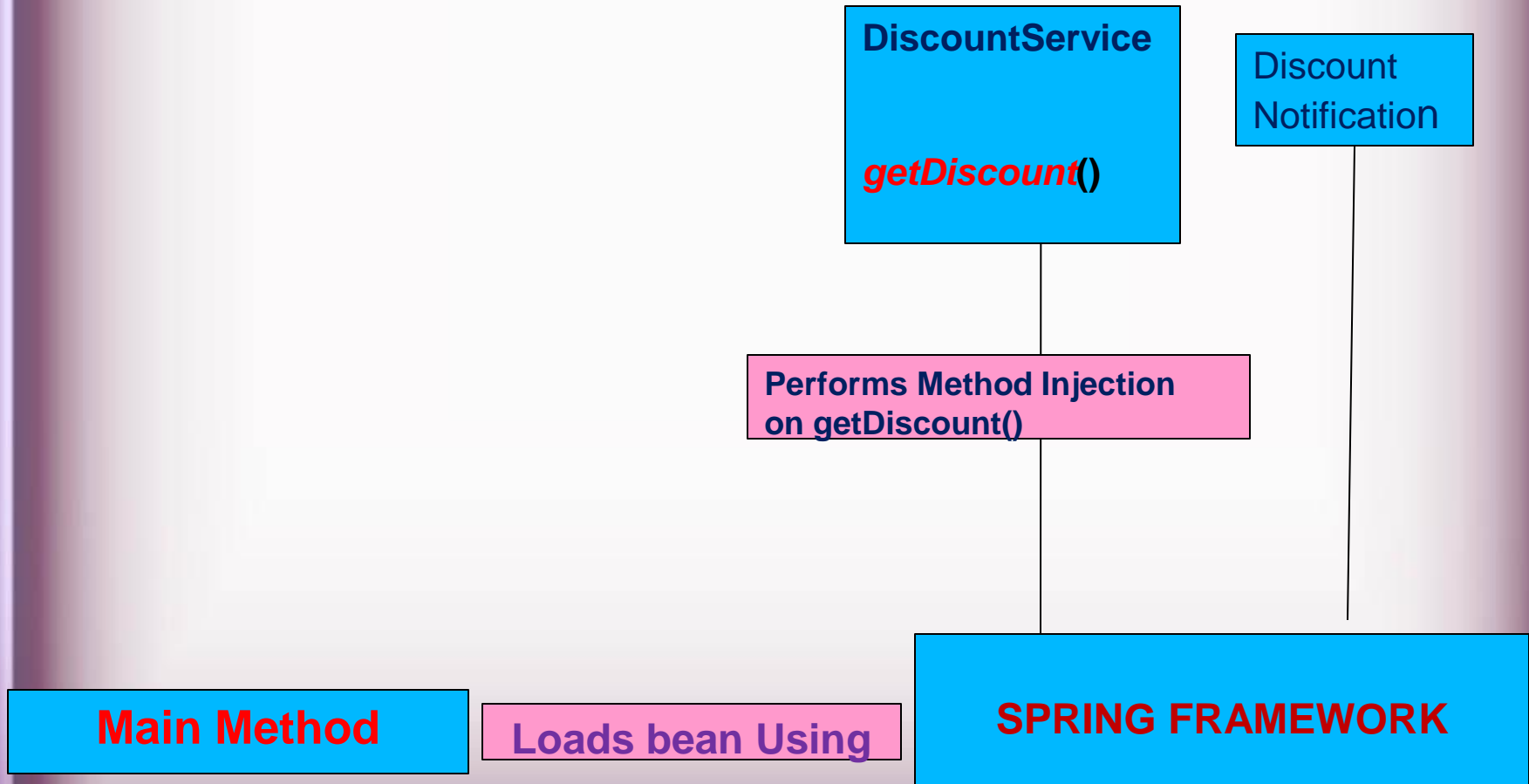
public MyPrototype many() {

log.info("====="+"PrototypeBean Intialized");

return new MyPrototype("ProtoBean");

}

Method Injection



Method Injection Example

@Scope("singleton")

public abstract class DiscountService {

@Lookup

protected abstract DiscountNotification

getDiscount(String month);

public void print() {

System.out.println("Hello");

}

}

Method Injection Example

@Scope("prototype")

public class DiscountNotification {

private String month;

public DiscountNotification(String month) {

super();

this.month = month;

}

public double showDiscount() {

if(month.equals("april")) {

return 0.50;

} else {

return 0.10;

}

}

}

Method Injection Example

@Bean

```
public DiscountService discountService() {
```

```
    return new DiscountService() {
```

@Override

```
protected DiscountNotification getDiscount(String  
    month) {
```

```
    return new DiscountNotification(month);
```

```
}
```

```
};
```

```
}
```

Test the Scopes

```
DiscountService service = ctx.getBean(DiscountService.class);
```

```
DiscountNotification protoBean;
```

```
protoBean = service.getDiscount("april");
```

```
log.info("Discount :="+protoBean.showDiscount());
```

```
protoBean = service.getDiscount("may");
```

```
log.info("Discount :="+protoBean.showDiscount());
```

CONFIGURATION PROPERTIES

@ConfigurationProperties.

- Spring Boot supports externalized configuration and easy access to properties defined in properties files.
 - Can also have custom properties defined
- But Spring Properties Editor won't be aware of these properties.
 - Need to make the spring Editor aware of these properties

@ConfigurationProperties

- *@ConfigurationProperties* works best with hierarchical properties that all have the same prefix.
 - Payment is a prefix
- Spring will automatically bind any property defined in our property file
- Spring Boot 2.2
 - Since version 2.2 *@ConfigurationProperties* classes are detected via classpath scanning.
 - *@Component* or *@EnableConfigurationProperties* is not required

Define Property in Java Bean

@Component

@ConfigurationProperties("payment")

@Data

```
public class PaymentProperties {
```

```
    private String currency = "INR";
```

```
}
```

application.yml

- application.yml

payment:

currency: USD

- **pom.xml**

The following Dependency is added to the pom.xml

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

Controller

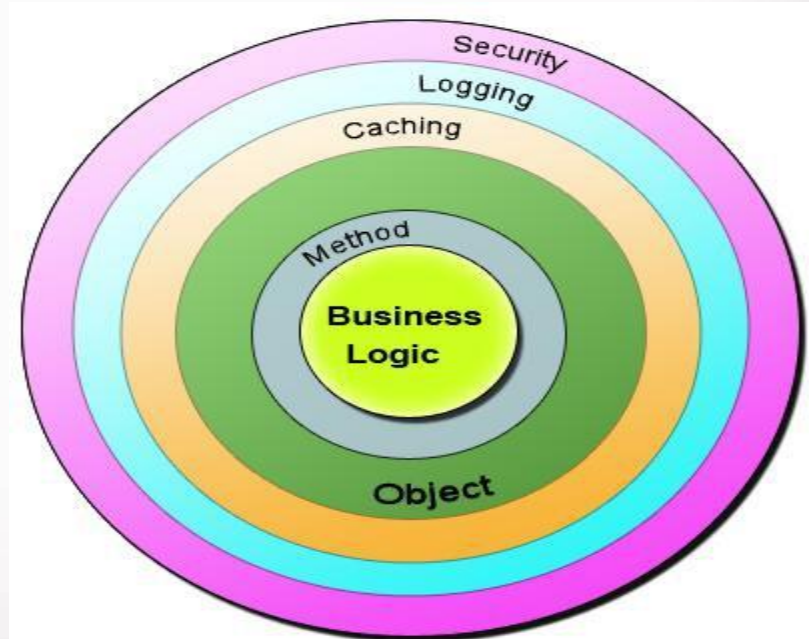
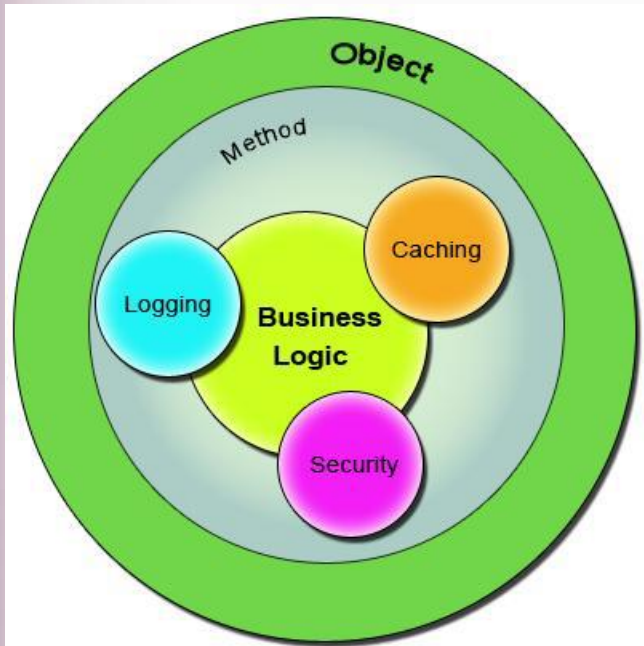
```
public class PaymentController {  
  
    @Autowired  
    private Payment payment;  
  
    @Value("${payment.currency}")  
    private String currency ;  
  
    @GetMapping(path = "/payment")  
    public Payment getPayment() {  
  
        payment.setCurrencyCode (this.currency);  
  
        return payment;  
  
    }  
}
```

SPRING AOP

Aspect Oriented Programming

- It's a Programming technique that promotes separation of concerns within a software system
- It is not a replacement for ***Object Oriented Programming***.
- **AOP** is another way of organizing Program Structure
- An aspect of a program which affect (crosscut) other concerns.
- **Cross Cutting Concerns**
 - Transaction management
 - Security
 - Auditing
 - Event handling

Bean in Spring container



Sample Code

```
public class Account {  
  
    public long deposit(long depositAmount){  
        newAmount = existingAccount + depositAccount;  
        currentAmount = newAmount;  
        return currentAmount;  
    }  
  
    public long withdraw(long withdrawalAmount){  
        if (withdrawalAmount <= currentAmount){  
            currentAmount = currentAmount – withdrawalAmount;  
        }  
        return currentAmount;  
    }  
}
```


The Challenge ?

- **Requirement -1**

- Add security to the Account class - only users with BankAdmin privilege is allowed to do the operations.

- **Requirement-2**

- Account class is to provide some kind of Logging

- **Requirement-3**

- Account class need to provide Transaction Management Facility.
- New requirements forces the methods and the logic to change a lot which is against the ***Software Design***.
- The code has to undergo the Software Development Lifecycle of Development, Testing, Bug Fixing, Development, Testing,

The Solution !

```
public void deposit(){  
    // Transaction Management  
    // Logging  
    // Checking for the Privileged User  
    // Actual Deposit Logic comes here  
}
```

```
public void withdraw(){  
    // Transaction Management  
    // Logging  
    // Checking for the Privileged User  
    // Actual Withdraw Logic comes here  
}
```

- The above logic that ***cross-cuts*** or overlaps the existing business logic as ***Concerns*** or ***Cross-Cutting Concerns***.
- These cross-cutting concerns from the application code can be modularized as different entities.

AOP Terminologies-Aspect

- An **Aspect** is a functionality or a feature that ***cross-cuts over*** objects.
- ***Logging*** and ***Transaction Management*** are the aspects.

```
public void businessOperation(BusinessData data)
{
    // Logging logger.info("Business Method Called");
    // Transaction Management Begin transaction.begin();
    // Do the original business operation here transaction.end();
}
```

AOP Terminologies-Joint Point

- **Join Points** defines the various **Execution Points** where an **Aspect** can be applied.
- In Spring AOP, a join point always represents a method execution.

```
public void someBusinessOperation(BusinessData data){
```

```
    //Method Start - > Possible aspect code here like logging.
```

```
    try{
```

```
        // Original Business Logic here.
```

```
    } catch(Exception exception) {
```

```
        // Exception -> Aspect code here when some exception is raised. }
```

```
    finally{
```

```
        // Finally -> Even possible to have aspect code at this point too.
```

```
    }
```

```
    // Method End -> Aspect code here in the end of a method.
```

```
}
```

AOP Terminologies-JoinPoints

- Points in the execution of the program
 1. *Start of the Method,*
 2. *End of the Method,*
 3. *Exception Block,*
 4. *Finally Block*
- In these points an **Aspect** can be made to execute.
- **These Execution Points** embedding **Aspects** are called **Join Points**.
- **An Aspect** need not be applied to all the possible **Join Points**.

AOP Terminologies-Pointcut

- **Join Points** refer to the **Logical Points** wherein a particular **Aspect** or a **Set of Aspects** can be applied.
- A **Pointcut** or a **Pointcut Definition** will exactly tell on which **Join Points** the **Aspects** will be applied.
- **Advice**
 - Advice is the code that implements the **Aspect**.
 - **Aspect** defines the functionality in a more abstract manner.
 - **Advice** provides a **Concrete code Implementation** for the **Aspect**.

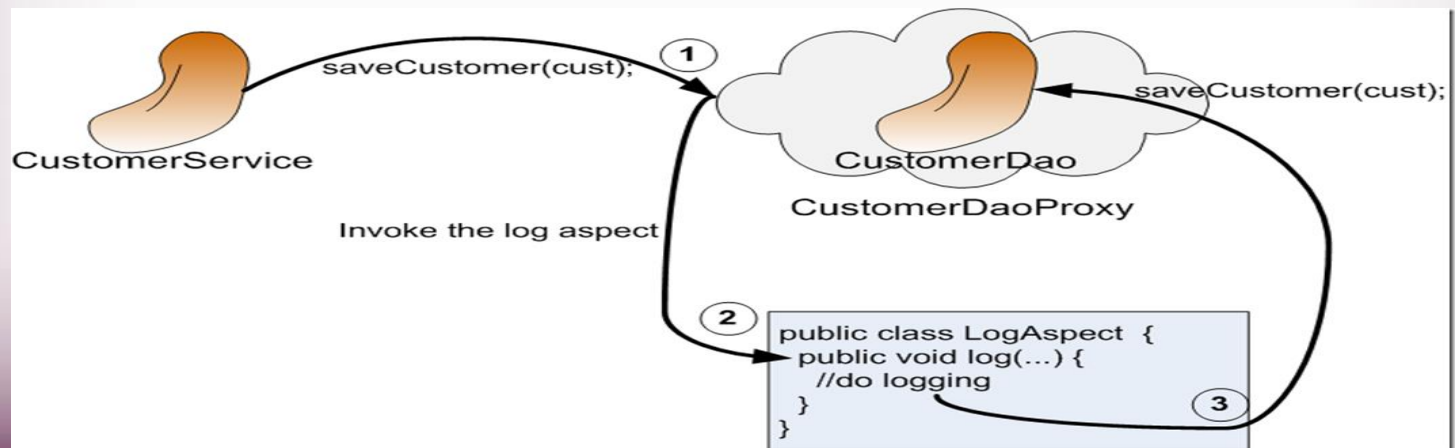
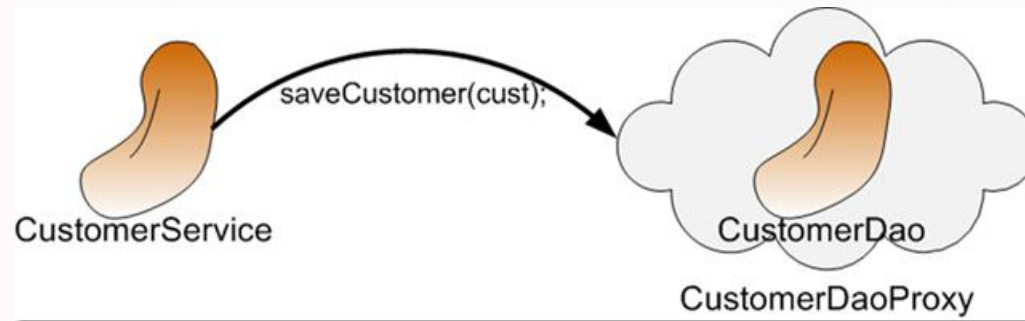
Advices

- **Before advice : @Before**
 - Executes before a join point,
 - Does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- **After returning advice : @AfterReturning**
 - Executed after a join point completes normally:

Advices

- **After throwing advice** : @AfterThrowing
 - Executed if a method exits by throwing an exception.
- **After advice** : @After
 - Executed when normal or exceptional return by which a join point exits
- **Around advice** : @Around
 - Advice that surrounds a join point

AOP Proxy Mechanism



Using Auto Proxy

- Can create proxies automatically for selected bean definitions.
- Built on Spring "bean post processor" infrastructure,
- Need for ProxyFactoryBean is eliminated

@Configuration

@EnableAspectJAutoProxy

public class AppConfig { }

Pointcut Designators

- ***execution***
 - Primary designator Spring AOP
 - Matching method execution join points
- ***Within***
 - Matching to join points within certain types
- ***@within***
 - Matching to join points within types that have the given annotation

Pointcut Designators

- ***this***
 - Matching to join points where the bean reference is an instance of the given type
- ***target***
 - Matching to join points where the target object is an instance of the given type
- ***@target***
 - Matching to join points where the class of the executing object has an annotation of the given type

Pointcut Designators

- ***args***
 - Matching to join points where the arguments are instances of the given types
- ***@args***
 - Matching to join points where the runtime type of the actual arguments passed have annotations of the given type(s)
- ***@annotation***
 - Matching to join points where the subject of the join point has the given annotation

Format of an execution expression

executor(modifier? return-type class-path? method-name(parameters?)
throws exception-type?)

```
execution(  
    modifiers-pattern  
    returning-type-pattern  
    declaring-type-pattern  
    name-pattern(param-pattern)  
    throws-pattern  
)
```

Spring Boot Dependency

`<dependency>`

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

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

`</dependency>`

Currency Converter

```
public class CurrencyConverter implements MyInterface {  
  
    public double dollarToRupee(double dlrAmt) {  
  
        return dlrAmt * 49.0d;  
    }  
  
    public double dollarToEuro(String dlrAmt)  
    {  
  
        double val =Double.parseDouble(dlrAmt);  
  
        return val * 70.00d;  
    }  
}
```


@Around Advice

@Aspect

public class MyAroundAdvice {

@Around("execution(* com.training.CurrencyConverter.*(..))")

**public Object aroundLog(ProceedingJoinPoint pjp) throws
Throwable {**

System.out.println("Called Before Method Invocation");

Object ret =pjp.proceed();

System.out.println("I am Around ");

return ret;

}

}

DEMO

Defining Pointcuts

@Component

public class CommonPointCutConfigs {

// Define The Various Pointcuts

}

@Aspect

@Configuration

@EnableAspectJAutoProxy(proxyTargetClass = **true**)

public class CustomAspect {

// Use the Point Cut Definitions

}

Execution and Around

```
@Pointcut(  
    value = "execution(* com.example.demo.services.*.*(..))")  
public void dataExecution(){}  

```

```
@Around("com.example.demo.config.CommonPointCutConfigs.dataExecution()")  
public Object around(ProceedingJoinPoint joinPoint) throws  
    Throwable {  
    long startTime = System.currentTimeMillis();  
    Object obj =joinPoint.proceed();  
    long timeTaken = System.currentTimeMillis() - startTime;  
    System.out.println(timeTaken);  
    return obj;  
}
```

Execution and After Returning

```
@Pointcut(value =  
    "execution(com.example.demo.domains.Weather  
    com.example.demo.services.*.*(..))")  
  
    public void afterReturn() {  
  
    } }
```

Execution and After Returning

```
@AfterReturning(returning = "mowsam"  
    ,value="com.example.demo.config.CommonPointCutConfigs.aft  
erReturn()")
```

```
public void usingReturn(JoinPoint joinPoint, Object  
mowsam) {
```

```
log.info(" @@@@ @@@ After  
Returning"+joinPoint.getKind());
```

```
log.info(" @@@@ -->" +mowsam);
```

```
}
```

```
}
```

Within and Before

```
@Pointcut(value = "within(com.example.demo.domains.*)")  
    public void usingWithin() {}
```

```
@Before("com.example.demo.config.CommonPointCutConfigs.usingWithin()")
```

```
public void before(JoinPoint joinPoint) throws Throwable{
```

```
    System.out.println(joinPoint.getTarget() + ", "
```

```
    +joinPoint.getSignature());
```

```
}
```

Target and After

```
@Pointcut(value ="target(com.training.ifaces.Converter)")  
    public void usingTarget() {}
```

```
@After("com.example.demo.config.CommonPointCutConfigs.using  
    Target()")  
    public void after(JoinPoint joinPoint) throws Throwable{  
  
        log.info("After ==="+joinPoint.getTarget() +", "  
        +joinPoint.getSignature());  
    }
```


Args and After

```
@Pointcut(value = "args(Double,String)")  
    public void usingMethodArgs() {}
```

```
@After("com.example.demo.config.CommonPointCutConfigs.using  
    MethodArgs()")  
    public void afterWithArgs(JoinPoint joinPoint) throws  
        Throwable{  
  
        log.info("Method Args ****"+joinPoint.getTarget() + ","  
            +joinPoint.getSignature());  
    }
```

Using Conditions

```
@Pointcut(value ="execution(* com.example.demo.services.*.*(..))  
"  
    + "&& args(com.example.demo.domains.City)")  
public void usingConditions() {  
  
    }  
  
}
```

Using Conditions

```
@Before("com.example.demo.config.CommonPointCutConfigs.usingConditions()")
```

```
public void conditionalBefore(JoinPoint joinPoint) throws  
Throwable{
```

```
log.info("conditional &&&&->" + joinPoint.getTarget() + ", "  
+ joinPoint.getSignature());
```

```
Arrays.asList(joinPoint.getArgs()).stream().forEach(System.out::  
println);
```

```
}
```

Creating Custom Advice

```
package com.example.demo.utils;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Retention(RUNTIME)
@Target(METHOD)
public @interface LogExecutionTime {

}
```

PROFILES

Profiles

- A way to segregate parts of application configuration and make it only available in certain environments.
- Used to load different types or sets of classes
- Spring profiles are Created with `@Profile` annotation.
- Profile-specific variants of `application.properties` can be set in the `application.properties`
- By Setting a property value
 - **`spring.profiles.active=test1`**

Profiles-Cont

- **application.properties**
 - **spring.profiles.active=prod**
 - server.port=4040
 - spring.application.name=Default
- **application-dev.properties**
 - server.port=5050
 - spring.application.name=Development
 - *logging.level.org.springframework=TRACE*
- **application-prod.properties**
 - server.port=6060
 - spring.application.name=Production
 - *logging.level.org.springframework=INFO*

Profiles -Example

@Data

@NoArgsConstructor

@AllArgsConstructor

public class UseProfile {

@Value("\${server.port}")

private int id;

@Value("\${spring.application.name}")

private String application;

Profiles -Example

@Autowired

private Environment env;

public void init() {

log.info("Init Method called");

Arrays.asList(env.getActiveProfiles()).forEach(System.out::println)
;

Arrays.asList(env.getDefaultProfiles()).forEach(System.out::println)
);

}

}

Profiles- Contd

```
@Bean(initMethod = "init")
```

```
@Profile("dev")
```

```
public UseProfile devProfileBean() {
```

```
    return new UseProfile();
```

```
}
```

```
@Bean(initMethod = "init")
```

```
@Profile("prod")
```

```
public UseProfile prodProfileBean() {
```

```
    return new UseProfile();
```

```
}
```

Logging

```
public static void main(String[] args) {
```

```
    ConfigurableApplicationContext ctx
```

```
        =SpringApplication.run(UsingProfilesApplication.class, args);
```

```
    UseProfile bean = ctx.getBean(UseProfile.class);
```

```
    log.info(bean.toString());
```

```
    ctx.close();
```

```
}
```

Building Application

- Using the Pom.xml
- **mvn package**
- Can Use the Generated jar to Execute the application.
- **\$ java -jar target/mymodule-0.0.1-SNAPSHOT.jar**

Using Jetty as Embedded Server

- Remove the tomcat dependency from the pom.xml and add the following jetty dependency.

```
<dependency>
```

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

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

```
  <exclusions>
```

```
    <exclusion>
```

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

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

```
    </exclusion>
```

```
  </exclusions>
```

```
</dependency>
```

Profiles

- A way to segregate parts of application configuration and make it only available in certain environments.
- Used to load different types or sets of classes
- Spring profiles are Created with `@Profile` annotation.
 - attach a profile name to an `@Configuration` annotation.
- Profile-specific variants of can be set in the `application.properties`
- Profile control two things:
 - Influence the application properties
 - Which beans are loaded into the application context.

Define the Profiles

```
public interface Profiles{
```

```
String DEVELOPMENT ="dev";
```

```
String PRODUCTION ="prod";
```

```
}
```

Defining Profiles

```
@Bean(initMethod = "init")  
@Profile(Profiles.DEVELOPMENT)  
public UseProfile devProfileBean() {  
  
return new UseProfile();  
}
```

```
@Bean(initMethod = "init")  
@Profile(Profiles.PRODUCTION)  
public UseProfile prodProfileBean() {  
  
return new UseProfile();  
  
}
```


Profiles -Example

```
public class MyDataSource {
```

```
private String driverClass;
```

```
private String jdbcURL;
```

```
return jdbcURL;
```

```
}
```

Profiles-Cont

@Configuration

@Profile("deve")

public class DevelopmentDataBase {

@Bean

public MyDataSource dataBase() {

return new MyDataSource("OracleDriver", "localhost");

}

}

Profiles- Contd

@Configuration

@Profile("prod")

public class ProductionDataBase {

@Bean

public MyDataSource dataBase() {

return new MyDataSource("SqlDriver", "192.100.10");

}

}

- **Application.properties**

spring.profiles.active=prod

server.port=4040

Activate Profiles

- Spring only acts on a profile if it's activated.
- `spring.profiles.active`
 - **Not recommended**
- `java -Dspring.profiles.active=foo -jar profiles-0.0.1-SNAPSHOT.jar`
- `spring-boot:run -Dspring-boot.run.profiles=dev`

Find the Current Profile

```
private Environment environment;
```

```
@PostConstruct
```

```
void postConstruct(){
```

```
String[] activeProfiles = environment.getActiveProfiles();
```

```
logger.info("active profiles: {}", Arrays.toString(activeProfiles));
```

```
}
```

```
}
```

Command Line Runner

- A Functional Interface
- A special bean that execute some logic after the application context is loaded and started.
 - They are Created within the same application context
 - Can create Multiple CommandLineRunner beans
 - It can be ordered using the Ordered interface or @Order annotation.
 - Has a run() method that accepts array of String as an argument
-

Command Line Runner

@Bean

```
public CommandLineRunner commandLineRunner() {
```

```
    return (args) -> {
```

```
        for(String eachArg:args) {
```

```
            System.out.println(" Info"+eachArg);
```

```
        }
```

```
    } ;
```

```
}
```

Application Runner

- A special bean that execute some logic after the application context is loaded and started.
 - **Has a run() method**
 - The arguments to main() method can be accessed in this method .
 - Can start any scheduler or log any message before application starts to run.
 - Run Method takes an Object of “ApplicationArguments”

Application Runner

@Bean

```
public ApplicationRunner applicationRunner() {
```

```
    return (appArgs) ->{
```

```
        System.out.println(appArgs.getSourceArgs());
```

```
    };
```

```
}
```

Building Application

- Using the Pom.xml
- **mvn package**
- Can Use the Generated jar to Execute the application.
- **\$ java -jar target/mymodule-0.0.1-SNAPSHOT.jar**

Creating a Aspect

@Aspect

@Component

public class CustomAdvice {

 @Around("@annotation(LogExecutionTime)")

 public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable
 {

 long start = System.currentTimeMillis();

 Object proceed = joinPoint.proceed();

 long executionTime = System.currentTimeMillis() - start;

 System.out.println(joinPoint.getSignature() + " executed in " + executionTime +
 "ms");

 return proceed;

 }

}

Adding Custom Advices

```
public class ConverterService {
```

```
    @LogExecutionTime
```

```
    public double dollarToRupees(double dlrAmount) throws  
        InterruptedException {
```

```
        Thread.sleep(200);
```

```
        return dlrAmount * 0.65;
```

```
    }
```

```
}
```

Configuration Class

```
package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.example.demo.service.ConverterService;

@Configuration
public class AppConfiguration {

    @Bean
    public ConverterService converterService() {

        return new ConverterService();
    }
}
```

Aspect and advice ordering

- order of advice in the **same aspect**
 - before
 - around
 - after finally
 - after returning or after throwing
- Spring **interface for ordering aspects**
 - org.springframework.core.Ordered
 - Spring **annotation**
 - org.springframework.core.annotation.Order

REST SERVICE USING SPRING DATA JPA

REST Overview

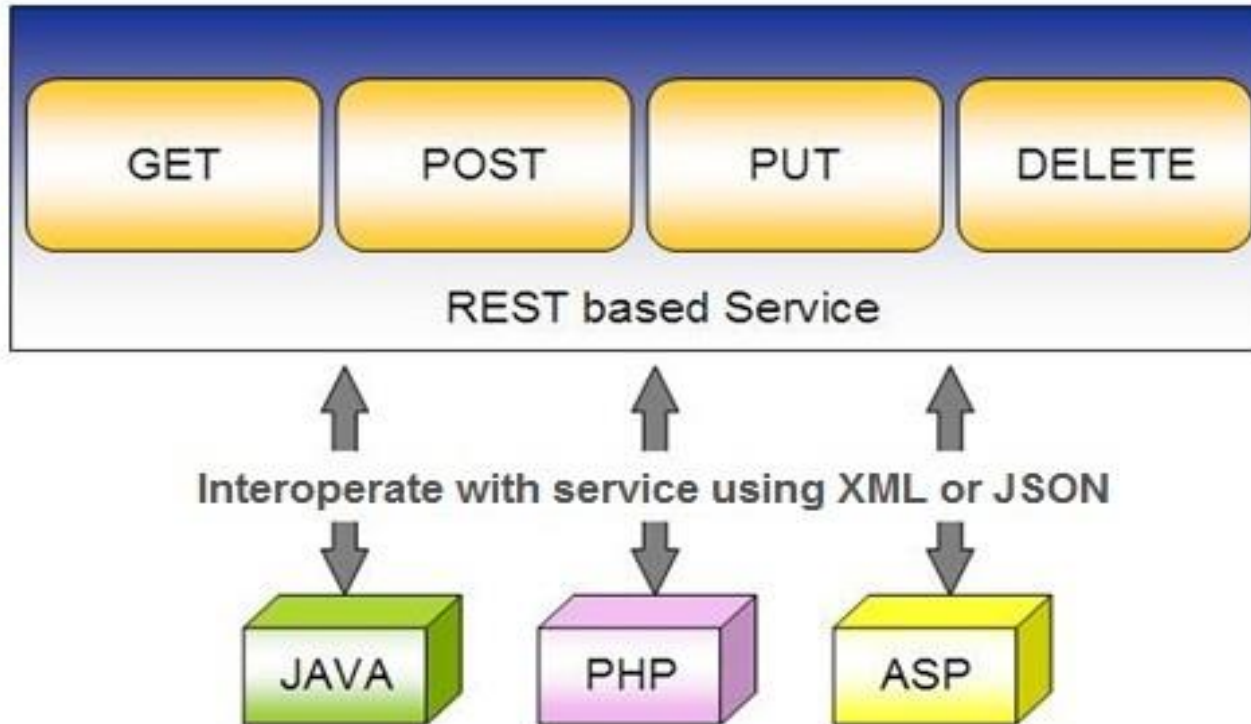
- Resource Oriented Architecture
 - **Data and functionality are considered resources**
 - Accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web.
- A resource is accessed via a common interface based on the HTTP standard methods.
- **A REST server** provides access to the resources and a REST client which accesses and modify the REST resources.
- Resources are identified by global ID's (which are typically URIs).

Lightweight Web Services

- REST is a lightweight alternative to Web Services and RPC.
- REST services is:
 - Platform-independent
 - Language-independent
 - Runs on top of HTTP Protocol
 - Used in the Presence of firewalls too.
- REST does not offer
 - built-in security features
 - encryption
 - session management
- ***These feature can be added by building on top of HTTP:***

Spring RESTful Services

- *REST* does not require the client to know anything about the structure of the API.
- Server needs to provide whatever information the client needs to interact with the service.



Richardson Maturity Model

- **Level 0 :**
 - Expose SOAP web services in REST style.
 - Expose action based services
 - (http://server/getPosts, http://server/deletePosts, http://server/doThis, http://server/doThat etc) using REST.
- **Level 1**
 - Expose Resources with proper URI's (using nouns).
 - Ex: http://server/accounts, http://server/accounts/10.
 - However, HTTP Methods are not used.

Richardson Maturity Model

- **Level 2**
 - Resources use proper URI's + HTTP Methods.
 - To update an account PUT and create a POST
- **Level 3**
 - HATEOAS- **H**ypermedia **A**s **T**he **E**ngine Of **A**pplication **S**tate
 - Need to give the required response to the client
 - Also need to give the next possible actions that he can perform
- When requesting information about a facebook user,
- REST service can return user details along with information about ***how to get his recent posts***, ***how to get his recent comments*** and ***how to retrieve his friend's list***.

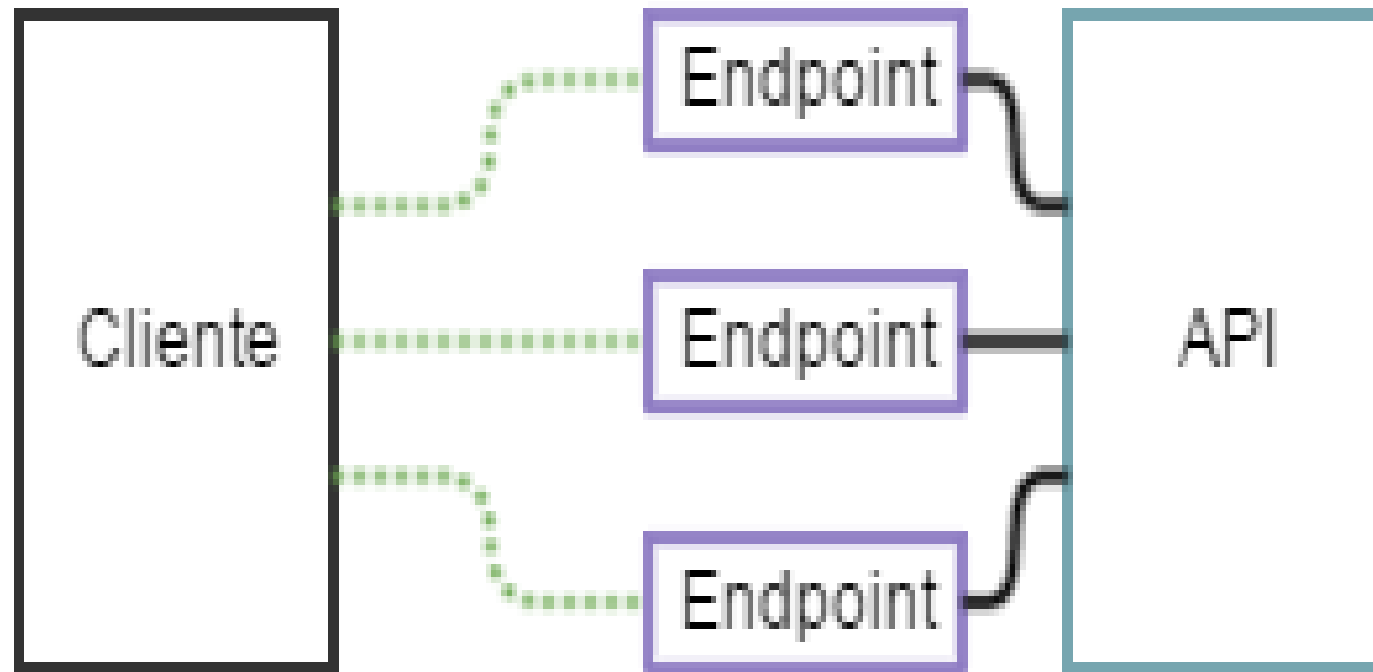
Using appropriate Request Methods

- **GET :**
 - Should not update anything.
 - Should be idempotent (same result in multiple calls).
 - Possible Return Codes 200 (OK) + 404 (NOT FOUND) +400 (BAD REQUEST)
- **POST :**
 - Should create new resource.
 - Ideally return JSON with link to newly created resource.
 - Same return codes as get possible.
 - In addition : Return code 201 (CREATED) is possible.

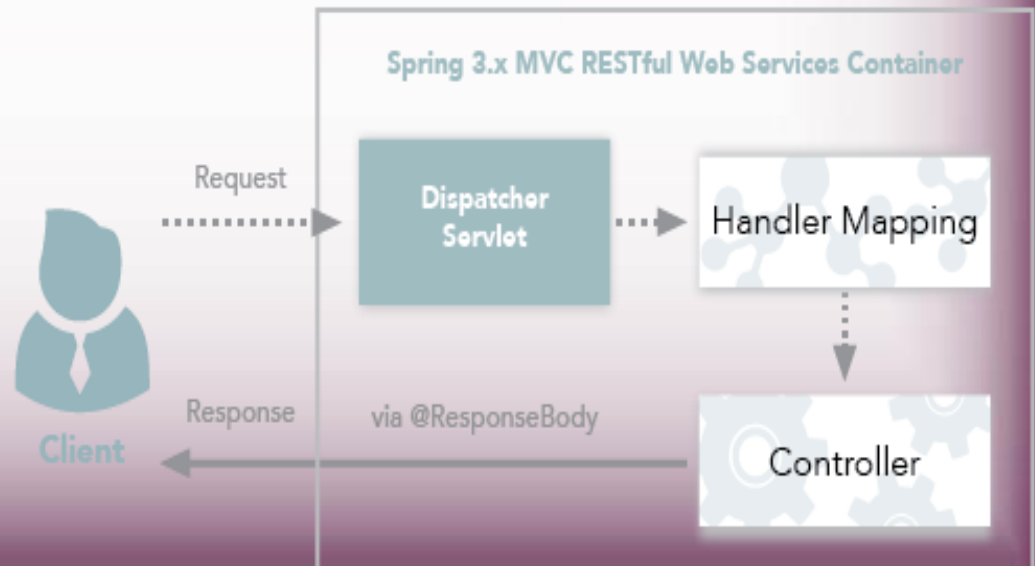
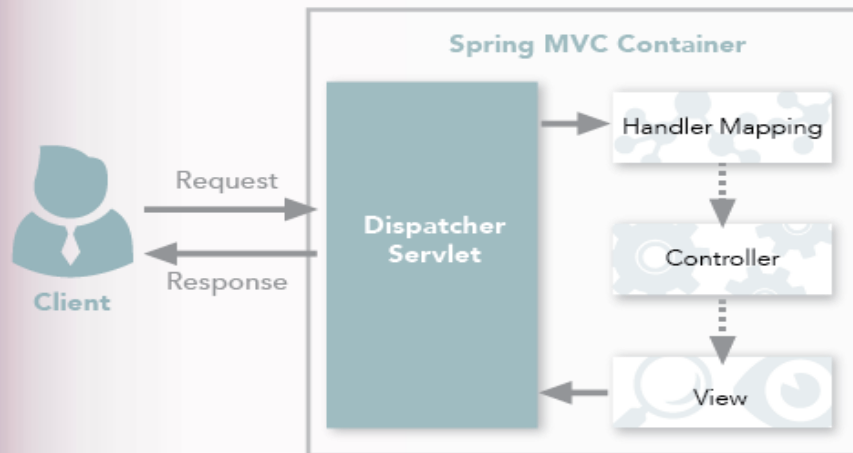
Using appropriate Request Methods

- **PUT :**
 - Update a known resource. ex: update client details.
 - Possible Return Codes : 200(OK)
- **DELETE :**
 - Used to delete a resource.
 - Possible Return Codes : 200(OK) or a 204 - no content

API and Endpoint



Spring MVC and Spring REST



SPRING DATA REST WITH JPA

Spring RESTful Services

- A RESTful architecture may expose multiple representations of a resource.
- From Spring 4 a `@RestController` annotation is added to controller
 - A combination of `@Controller` and `@ResponseBody`
- A Rest Controller method returns a domain object instead of a view.

REST Stereotypes

- **@RestController**
 - An implicit `@ResponseBody` is being added to the methods.
 - Allows Spring to render the returned `HttpEntity` and its payload, directly to the response.
- **@GetMapping**
 - To map HTTP GET requests onto specific handler methods.
 - *Composed annotation* for `@RequestMapping(method = RequestMethod.GET)`.

REST Stereotypes

- **@PathVariable**

- Indicates that the Method parameter should be bound to a URI template variable.

@GetMapping("/members/{id}")

public String getById(@PathVariable String id) { }

- **@PostMapping**

- Combined shortcut for @RequestMapping(method = RequestMethod.POST).

@PostMapping("/members")

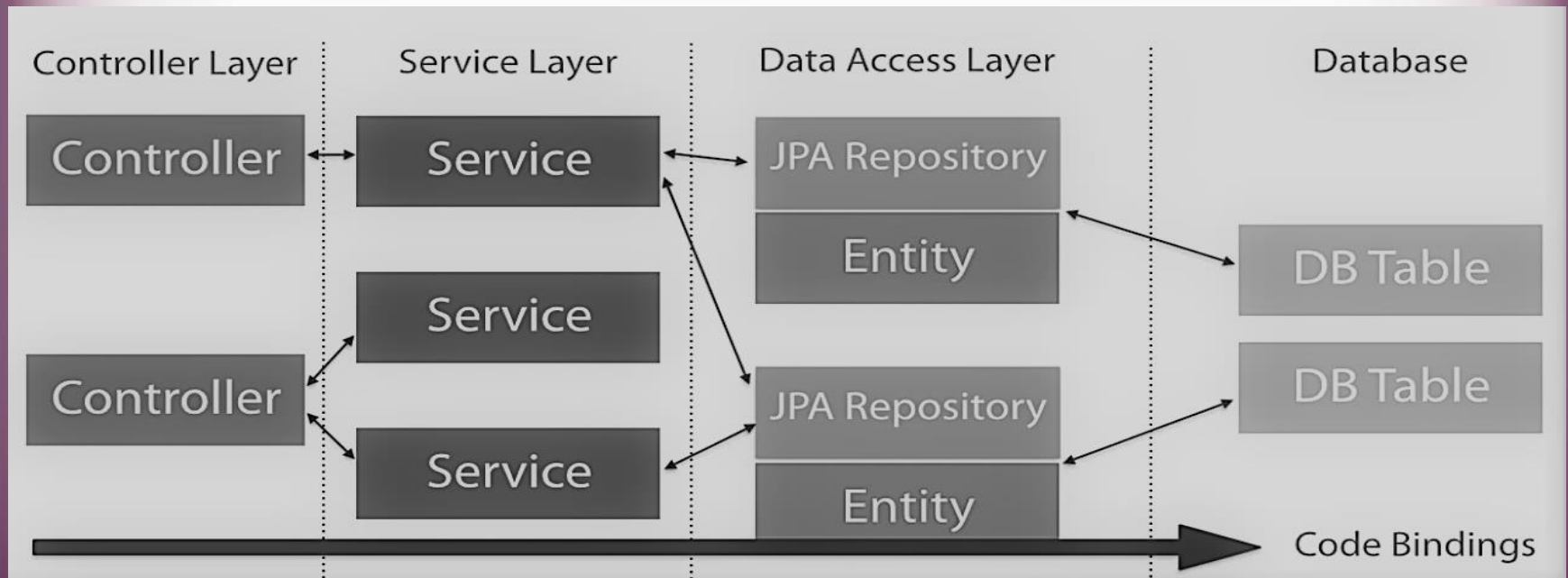
public void addMember(@RequestBody Member member) { }

@Repository

- There are three repository in Spring Data
- ***CrudRepository***
 - Extends Repository
 - provides CRUD functions
- ***PagingAndSortingRepository***
 - Extends CrudRepository
 - provides methods to do pagination and sort records
- ***JpaRepository***
 - Extends PagingandSortingRepository
 - Provides Methods such as flushing the persistence context and delete records in a batch
 - Querying methods return List's instead of Iterable's

Repository

- Controllers can be injected with services or repositories
- Services can be injected with repositories,
- Controller **SHOULD NOT** be injected into Services or repositories



Creating Spring Data Application

1. Create a repository interface and extend one of the repository interfaces provided by Spring Data.
 2. If required add custom query methods to the created repository interface
 3. Inject the repository interface to another component and use the implementation that is provided automatically by Spring.
- Need **NOT** create an implementation class
 - *Spring will automatically create its implementation class at runtime.*
 - The Repository class will be auto detected if suitably placed in the scan path

Custom Query Methods

- Custom Query Methods defined also exposed by their name
- Query method resources are exposed under the search resource.
- **@RestResource**
 - Can be Used to change the segment of the URL under which this query method is exposed,

Custom Query Methods

```
@RepositoryRestResource(path = "einvoices")
```

```
public interface InvoiceRepository extends  
JpaRepository<Invoice, Integer> {
```

```
@RestResource(path = "customerNames")
```

```
public List<Invoice> findByCustomerName(String name);  
}
```

- Then It can be Accessed in the following Way
 - <http://localhost:2020/api/v1/einvoices/search/customerNames?name=Ramesh>

Custom Repository Method

@Repository

```
public interface InvoiceRepository extends JpaRepository<Invoice,  
    Long> {
```

```
    List<Book> findByCustomerName(String customerName);
```

```
}
```

@Query

- Used to write the more flexible query to fetch data.
- Supports both JPQL and native SQL queries.
- By default will be using JPQL to execute the queries.
 - Using normal SQL queries, would get the query syntax error exceptions.
- Can use native SQL queries by setting nativeQuery flag to true.
 - Pagination and dynamic sorting for native queries are not supported in spring data jpa.

@Query

- Used to write the more flexible query to fetch data.
- Supports both JPQL and native SQL queries.
- By default will be using JPQL to execute the queries.
 - Using normal SQL queries, would get the query syntax error exceptions.
- Can use native SQL queries by setting nativeQuery flag to true.
 - Pagination and dynamic sorting for native queries are not supported in spring data jpa.

@Modifying

- @Query
 - Can also be used for queries that add, change, or remove records in database.
- @Modifying
 - Indicates a query method should be considered as modifying query as that changes the way it needs to be executed.
 - Not applied on custom implementation methods or queries derived from the method name
 - Those methods already have control over the underlying data access APIs or specify if they are modifying by their name.

@Modifying

- Method annotated with @Modifying also requires @Transactional
- By default, CRUD methods on repository instances are transactional.
 - For read operations, the transaction configuration readOnly flag is set to true.
 - For other operations are configured with a plain @Transactional
 - We need to add @Transactional annotation with required attribute values.

Examples:

- `SELECT o1 FROM Order o1, Order o2 WHERE o1.quantity > o2.quantity`
- `SELECT p FROM Person p WHERE p.age BETWEEN 15 and 19`
- `SELECT a FROM Address a WHERE a.country IN ('UK', 'US' , 'France')`
- `SELECT o FROM Order o WHERE o.lineItems IS EMPTY`
- `SELECT a FROM Address a WHERE a.phone LIKE '12%3'`
@Query(value = "select name,author,price from Book b where b.price>?1 and b.price<?2")
List<Book> findByPriceRange(long price1, long price2);

@Query - Example

@Repository

```
public interface TourGuideRepository extends  
    JpaRepository<TouristGuide, Integer> {
```

```
    public TouristGuide findByFirstName(String name);
```

```
        @Query(value = "SELECT * FROM tourguides WHERE  
            first_name = :firstName", nativeQuery = true)
```

```
        List<TouristGuide> findGuideByFirstName(@Param("firstName")  
            String firstName);
```


@Query - Example

```
@Query("FROM TouristGuide WHERE firstName =  
:firstName AND lastName = :lastName")
```

```
List<TouristGuide>
```

```
findByFirstAndLastName(@Param("firstName") String  
firstName, @Param("lastName") String lastName);
```

```
@Query("UPDATE TouristGuide SET firstName = :prefix ||  
firstName")
```

```
@Modifying
```

```
@Transactional
```

```
void updatePrefix(@Param("prefix") String prefix);
```

```
}
```

JPA Inheritance

@FieldDefaults(level = AccessLevel.**PRIVATE**)

@AllArgsConstructor()

@NoArgsConstructor

@Data

@Entity

@Table(name="catalog")

@Inheritance(strategy = InheritanceType.JOINED)

public class Catalog {

@Id

int id;

String description;

LocalDate releaseDate;

}

JPA Inheritance

@FieldDefaults(level = AccessLevel.*PRIVATE*)

@AllArgsConstructor

@Entity

@NoArgsConstructor

@Table(name="corpcatalog")

public class CorpCatalog extends Catalog {

private double discount;

public CorpCatalog(int id, String description, LocalDate releaseDate, double discount) {

super(id, description, releaseDate);

this.discount = discount;

}

}

Inheritance

@Repository

```
public interface CatalogRepository<T extends Catalog>  
    extends JpaRepository<T,Integer> {
```

```
    Catalog findByDescription(String desc);
```

```
}
```

One to Many

CrudRepository Interface

- Extends Repository Interface and has the following Methods
 - **<S extends T> S save(S entity)**
 - **<S extends T> Iterable<S> saveAll(Iterable<S> entities);**
 - **Optional<T> findById(ID primaryKey)**
 - **Iterable<T> findAll()**
 - **void delete(T entity)**
 - **long count()**
 - **boolean existsById(ID id);**

Paging And Sorting Repository

- Extends CrudRepository Interface and has the following Methods

```
Iterable<T> findAll(Sort sort);
```

```
Page<T> findAll(Pageable pageable);
```

JpaRepository

Extends PagingAndSortingRepository and QueryByExampleExecutor

Has the following Methods

List<T> findAllById(Iterable<ID> ids);

<S extends T> List<S> findAll(Example<S> example);

<S extends T> List<S> findAll(Example<S> example, Sort sort);

<S extends T> List<S> saveAll(Iterable<S> entities);

<S extends T> S saveAndFlush(S entity);

void deleteInBatch(Iterable<T> entities);

void deleteAllInBatch();

T getOne(ID id);

void flush();

Service

@Service

public class InvoiceService {

@Autowired

private InvoiceRepository repo ;

**public List<Invoice> findAll(){
return repo.findAll();**

}

**public Optional<Invoice> findById(int id) {
return repo.findById(id);**

}}}

Controller

@RestController

public class InvoiceController {

@Autowired

private InvoiceService service ;

@GetMapping(path = "/api/v1/invoice")

public List<Invoice> findAll() {

List<Invoice> details = service.findAll();

return details;

}

@PostMapping(path = "/api/v1/invoice")

public Invoice add(@RequestBody Invoice entity) {

return this.service.add(entity);

}

Put Mapping

```
@PutMapping(path = "/api/v1/invoice")
```

```
public Invoice update(@RequestBody Invoice entity) {
```

```
return this.service.add(entity);
```

```
}
```

Remove Method in Service

```
public Invoice remove(Invoice entity) {  
  
    Invoice resp = null;  
  
    if(repo.existsById(entity.getInvoiceNumber())) {  
  
        repo.delete(entity);  
  
        resp= entity;  
    }  
  
    return resp;  
}
```

Delete Mapping in Controller

```
@DeleteMapping(path = "/api/v1/invoice")
public Invoice remove(@RequestBody Invoice entity, HttpServletResponse
    response) {

    try {
        Invoice deleted = this.service.remove(entity);
        if (deleted==null){
            throw new
                ResponseStatusException(HttpStatus.NOT_FOUND, "Requested Entry Not
                Found");
        }
    } catch (ResponseStatusException e) {
        throw e;
    }

    return entity ;
}
```

CORS configuration

- Annotation enables cross-origin requests
- Can be added at the Method or controller Class level
- `@CrossOrigin(origins = "http://localhost:9000")`
- By default, it allows all origins, all headers, the HTTP methods specified in the `@RequestMapping` annotation
- It can be customized specifying the value of one of the annotation attributes:
 - origins,
 - methods,
 - allowedHeaders,

Yaml

- YAML is a superset of JSON
- A convenient format for specifying hierarchical configuration data.
- **SnakeYAML**
 - If this library is in the class path Spring Boot automatically supports YAML as an alternative to properties
- Spring provides two convenient classes that can be used to load YAML documents.
- **YamlPropertiesFactoryBean**
 - To load YAML as Properties
- **YamlMapFactoryBean**
 - To load YAML as a Map.

Yaml

- .yaml file is advantageous over .properties file
 - Has type safety,
 - Hierarchy
 - supports list
- YAML supports lists as hierarchical properties or inline list, for example
- my:
servers:
 - dev
 - prod
- servers: [dev, prod]

Yaml

server:

port: 2020

servlet:

context-path: /api/v1

spring:

datasource:

username: root

password: srivatsan

driver-class-name: com.mysql.cj.jdbc.Driver

url: jdbc:mysql://localhost:3306/test

Yaml

jpa:

hibernate:

ddl-auto: update

naming:

physical-strategy:

org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

show-sql: true

open-in-view: false

profiles:

active:

- dev

logging:

level:

org.hibernate.sql: debug

org.hibernate.type.descriptor.sql.BasicBinder: trace

DOCUMENTING REST API

Documentation

- Essential part of building REST APIs
- **Open Api Documentation**
 - *Standard, programming language-agnostic interface description for [REST APIs](#),*
 - *Allows both humans and computers to discover and understand the capabilities of a service*
 - *No need to access source code,*
 - *Can add additional documentation, or inspect network traffic*

Spring Documentation

- A tool that simplifies the generation and maintenance of API docs
- Based on the OpenAPI 3 specification, for Spring Boot 1.x and 2.x applications.
- Required dependency are added to *pom.xml*:

`<dependency>`

`<groupId>org.springdoc</groupId>`

`<artifactId>springdoc-openapi-ui</artifactId>`

`<version>1.3.0</version>`

`</dependency>`

View the Documentation

- Can be viewed after executing the application
- Its rendered in the browser in JSON format by default.
- **Default Path** : <http://localhost:8080/v3/api-docs/>
- *Can be Customised using `application.properties` file:*

springdoc.api-docs.path=/api-docs

- **Custom Path** : <http://localhost:8080/api-docs/>

Swagger

- springdoc-openapi is integrated with Swagger UI
- Access from the following URL
- <http://localhost:8080/swagger-ui.html>
- Can be customized using `application.properties`

`springdoc.swagger-ui.path=/swagger-ui-custom.html`

Custom Configuration

- Documenting at the Class Level

```
@SpringBootApplication
```

```
@OpenAPIDefinition( info = @Info(title = "Invoice Service")  
)
```

- Can also Document at the Method Level

```
@Operation(description = "Method to Add Invoice")
```

Can also Document at the Model Class

- Using JSR 303 Annotations

PAGING AND SORTING

PagingAndSortingRepository

- *findAll(Pageable pageable)*
 - *Pageable* object with following properties
 - Page size
 - Current page number
 - Sorting
- *findAll(Sort sort)*
 - *Sort Object with the Property on Which the sorting is to be done*
 - ***Sort.by(propName)***

Entity

@Data

@AllArgsConstructor

@NoArgsConstructor

@Entity

@Table(name = "**STATEDOCTOR**")

public class Doctor {

 @Id

private int doctorCode;

private String doctorName;

private String specialization;

private long handPhone;

private String location;

private String qualification;

}

Repository

@Repository

public interface DoctorRepository **extends**

PagingAndSortingRepository<Doctor,Integer> {

Iterable<Doctor> findAll(**Sort** sort);

Page<Doctor> findAll(**Pageable** pageable);

}

Service

- @Service

```
public class DoctorService {
```

```
    @Autowired
```

```
    private DoctorRepository repo;
```

```
    public Iterable<Doctor> findAll(String propName){
```

```
        Sort sortBy = Sort.by (propName);
```

```
        return this.repo.findAll (sortBy);
```

```
    }
```

Service

```
public List<Doctor> findByPage(int page, int size, String propName){  
  
    Pageable paging =  
        PageRequest.of(page,size,Sort.by(propName).ascending ());  
  
    Page<Doctor> pagedResult = repo.findAll(paging);  
  
    if(pagedResult.hasContent()) {  
        return pagedResult.getContent();  
    } else {  
        return new ArrayList<Doctor> ();  
    }  
}  
}
```

Rest Controller

- @RestController

```
public class DoctorController {
```

```
    @Autowired
```

```
    private DoctorService service;
```

```
    @GetMapping("/doctor/sort/{prop}")
```

```
    public Iterable<Doctor> findAll( @PathVariable("prop") String  
prop){
```

```
        return this.service.findAll (prop);
```

```
    }
```

Rest Controller

```
@GetMapping("/doctor/paging/{page}/{size}/{propName}")
```

```
    public List<Doctor> findByPage(@PathVariable("page") int  
page, @PathVariable("size") int size,  
@PathVariable("propName") String propName){  
  
        return this.service.findByPage (page,size,propName);  
    }  
}
```


HATEOAS

Hateos

- **Hypermedia As The Engine Of Application State.**
- When the browser loads a page
 - Can see all the content that the page has to offer.
 - Page also allows to perform a lot of actions around that data, :
- Provides extra information in REST API responses
 - Users can get additional endpoint details from a single call.
- Can build systems with dynamic API calls
 - Moving from one endpoint to another using the information retrieved from each call.

Hypertext Application Language

- HAL is a simple format that gives an easy, consistent way to hyperlink between resources in your REST API.
- Used as standard convention for defining hypermedia such as links to external resources within JSON

**@GetMapping(value="/reviews/{id}",
produces = "application/hal+json")**

Spring HATEOAS

- Can Add the spring-boot-starter-hateoas Maven dependency:
- If Using Spring Tool Suite select Spring Hateoas

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

Invoice

@Entity

@Table(name = "rest_invoice")

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Invoice {

@Id

private int id;

private String customerName;

@DateTimeFormat(iso = ISO.*DATE*)

private LocalDate invoiceDate;

private double amount;

}

@RepositoryRestResource

- Creates HATEOAS service with Spring JPA
- Operations will be exposed in HATEOAS format.
- Users calls Exposed REST Endpoints directly via REST calls.
- Doesn't require any controller/service/dao layers but can have direct access.
- Can also customize REST endpoints

@RepositoryRestResource

- @RepositoryRestResource
Interface InvoiceRepository extends CrudRepository<Invoice, Long> { }
- **http://localhost:8080/invoices/**
- *Spring exposes collection resources*
 - Named after the **uncapitalized, pluralized** version of the **domain class** the exported repository is handling.
- Can be customized by using
 - @RepositoryRestResource(path = "bills")
 - **http://localhost:8080/bills/**

@RepositoryRestResource

- Exposes the basic CRUD:
 - GET /invoices
 - POST /invoices
 - GET / invoices /{id}
 - PUT /invoices/{id}
 - POST /foos/{id}
 - PATCH /invoices/{id}
 - DELETE /foos/{id}

@RestResource

- **@RestResource**
 - Spring Data REST Repository export many methods
 - Used to Control and customize the methods
- **@RestResource(exported = false)**
 - To skip this method when triggering the HTTP method exposure:
- **@RestResource(path = "customerNames" ,rel="customerName")**
 - Used to **customize the URL path mapped to a repository method**
 - ***path*** for the URL path
 - ***rel*** for the link id

@RestResource

public interface InvoiceRepository extends
JpaRepository<Invoice, Integer> {

@RestResource(path = "customerNames", rel="customMethod")

public Page<Invoice> findByCustomerName(@Param("name")
String name, Pageable page);

}

- <http://localhost:2020/api/v1/invoices/search/>
- Can see *customMethod* in the links part of the JSON response
- <http://localhost:2020/api/v1/invoices/search/customerNames/?name=Ramesh&page=0&size=1>

Spring HATEOAS

- The basic building blocks are
 - Links
 - **RepresentationModel**
 - Container for a collection of Links.
- **EntityModel**
 - Sub class of Representation Model
 - Used for single resources
- **CollectionModel**
 - Sub class of Representation Model
 - Used for multiple resources and Page.

Links

- **Link**
 - Immutable object
 - Used to store URI or location of resource

```
"_links": {
```

```
"self": {
```

```
"href": "http://localhost:8080/doctors/1"
```

```
}
```

```
}
```

Links

- href
 - Pointing to the URI of the resource.
 - Its wrapped inside a self tag
 - Identifies the relationship with the entity.
- The returned resources might not be the full representation of itself.
 - A doctor can have a list of patients,
 - But all those details is not returned by default.
 - The links will help to navigate and access other details through the link.

Representation Models

- Represents a root class for all other Spring HATEOAS model classes.
- It contains a collection of Links and provides method to add/remove them.
- Model are created by **extending RepresentationModel** class.
- There are three Subtypes
 1. **Entity Model**
 2. **Collection Model**
 3. **Paged Model**

Representation Models

- **Entity Model:**
 - Used to represent a resource that corresponds to a single object.
 - Can wrap resource with the EntityModel and pass it to a calling service or return it through a REST endpoint.
- **Collection Model:**
 - Used to wrap resources that corresponds to a collection of objects.
- **Paged Model:**
 - Used to wrap pageable collections

RepresentationModel

```
public class Doctor extends RepresentationModel<Doctor> {  
    private int id;  
    private List<Patient> patientList;  
}
```


MVC LinkBuilder

- **WebMvcLinkBuilder**
 - Used to create links using *controller* classes
 - It Can be made to point methods.
 - Provides a Static linkTo Method

Link link =

linkTo(methodOn(DoctorController.class)

▪**getDoctorById(id)**

▪**withSelfRel());**

Relational Links

- **withRel()**
 - Used to specify the endpoint with which the linked resource can be accessed:
 - Links for resources which have a relationship between them or they point to a different resource

```
Link link = linkTo(methodOn(DoctorController.class)  
    .getDoctorPatients(doctor.getId())) .withRel("patientList");
```

- *User can get the patientList for the doctor object,*
- *using the getDoctorPatients() method inside DoctorController class.*

Relational Links

Generates the following link:

```
"_links": {  
  "patientList": {  
    "href": "http://localhost:8080/doctors/1/patients"  
  }  
}
```

Doctor

@Entity

@Table(name = "hateoas_doctor")

public class Doctor extends RepresentationModel<Doctor> {

@Id

private int id;

private String name;

private String speciality;

public Doctor(int id, String name, String speciality) {

super();

this.id = id;

this.name = name;

this.speciality = speciality;

}

**@OneToMany(mappedBy = "doctor", cascade =
CascadeType.ALL, fetch=FetchType.EAGER)**

private List<Patient> patientList= new ArrayList<Patient>();

}

Patient

@Entity

@Table(name = "hateoas_patient")

public class Patient extends RepresentationModel<Patient> {

 @Id

private int id;

private String name;

 @ManyToOne

 @JoinColumn(name="doctor_ref",referencedColumnName = "id")

 @JsonIgnore

private Doctor doctor;

}

Case History

```
public class CaseHistory {
```

```
    int patientId;
```

```
    double weight;
```

```
    double height;
```

```
    double hb1ac;
```

```
}
```

Repository

@Repository

```
public interface DoctorRepository extends  
    JpaRepository<Doctor, Integer> {
```

```
}
```

@Repository

```
public interface PatientRepository extends  
    JpaRepository<Patient, Integer> {
```

```
}
```

Case Details

@Component

```
public class CaseDetailService {
```

```
    HashMap<Integer,CaseHistory> map=new HashMap<>();
```

```
public CaseDetailService() {
```

```
    super();
```

```
    map.put(103, new CaseHistory(101,65.00,5.0,5.2));
```

```
    map.put(102, new CaseHistory(102,75.00,5.1,6.4));
```

```
}
```

```
public CaseHistory getDetails(int id){
```

```
    return map.get(id);
```

```
}
```

```
}
```


Doctor Controller

```
@GetMapping(path = "/doctors")
```

```
public CollectionModel<Doctor> findAll(){
```

```
List<Doctor> allDoctors = this.repo.findAll();
```

```
    for (Doctor eachDoctor : allDoctors) {
```

```
        int id = eachDoctor.getId();
```

```
        Link selfLink =
```

```
        WebMvcLinkBuilder.linkTo(DoctorController.class)
```

```
            .slash("doctors/"+id).withSelfRel();
```

```
        eachDoctor.add(selfLink);
```

Doctor Controller

```
for(Patient eachPatient:eachDoctor.getPatientList()) {  
    int patId = eachPatient.getId();  
    Link patLink =  
        WebMvcLinkBuilder.linkTo(DoctorController.class).slash("patient/"+patId).withSelfRel();  
    eachPatient.add(patLink) ;  
}  
}  
  
Link link =  
    WebMvcLinkBuilder.linkTo(DoctorController.class).withSelfRel(  
        );  
    CollectionModel<Doctor> result =  
        CollectionModel.of(allDoctors, link);  
    return result;  
}
```

Patient Controller

@Autowired

CaseDetailService service;

@GetMapping(path = "/patient/{id}")

**public EntityModel<CaseHistory> findById(@PathVariable("id")
Integer id){**

CaseHistory history = **this.service.getDetails(id);**

return EntityModel.of(history);

}

Entity

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Clinic {

private long id;

private String clinicName;

private String location;

}

Main Method

@SpringBootApplication

@EnableHypermediaSupport(type = HypermediaType.HAL)

public class HospitalHateosApplication {

public static void main(String[] args) {

 SpringApplication.run(*TouristGuideHateosApplication.class*,
 args);

}

}

SPRING BOOT UP TO 2.2

Entity

@Entity

@Table(name="docsreview")

@Data

public class DoctorReview {

 @Id

 private long reviewId;

 private String reviewerName;

 private String message;

 private double rating;

}

Clinic Controller

@RestController

public class ClinicController {

@GetMapping(value = "/clinics")

public Iterable<Clinic> findAll(){

return Arrays.asList(new Clinic(101, "dr.lal", "marol"));

}

}

Repository

@Repository

```
public interface ReviewRepository extends  
    CrudRepository<DoctorReview, Long> {  
  
}
```

Service

@Service

```
public class ReviewService {
```

```
    @Autowired
```

```
    private ReviewRepository repo;
```

```
    public DoctorReview add(DoctorReview entity) {
```

```
        return this.repo.save(entity);
```

```
    }
```

```
    public Iterable<DoctorReview> findAll(){
```

```
        return this.repo.findAll();
```

```
    }
```

```
        public DoctorReview findById(long id){
```

```
            return this.repo.findById(id).get();
```

```
        }
```

```
    }
```

Controller LinkBuilder

- API Provides ControllerLinkBuilder and Link classes
 - spring HATEOAS module.
- **Link**
 - Representation of link to be added in REST resource.
- **ControllerLinkBuilder**
 - helps in building the links using it's methods
- **method links**
 - which can be followed to perform some operations on individual resource.

Link link1 =

```
ControlLinkBuilder.linkTo(methodOn(ReviewController.class).findAll()).  
withRel("findAllDoctors");
```

Link link2 =

```
linkTo(methodOn(ClinicController.class).findAll()).withRel("findAllClinics");
```

Return Hateos Response

- Need to include related resources in the response.
- The return type of the Method becomes a Representation of Resource.
- Resource is a simple class wrapping a domain object and allows adding links to it.

```
Resources<DoctorReview> resource =  
    new Resources<>(allReviews,link1,link2);
```

Controller

```
import static
    org.springframework.hateoas.mvc.ControllerLinkBuilder.*;
import org.springframework.hateoas.*;
import org.springframework.hateoas.mvc.ControllerLinkBuilder;
import org.springframework.hateoas.config.*;

public class ReviewController {

    @Autowired
    private ReviewService service;

    @GetMapping(value = "/reviews")
    public Iterable<DoctorReview> findAll(){

        return this.service.findAll();
    }
}
```

Creating Links

```
@GetMapping(value="/reviews/{id}",produces = "application/hal+json")
public Resources<DoctorReview> getReviewById(@PathVariable("id") long
id) {

    DoctorReview review = this.service.findById(id);

    Iterable<DoctorReview> allReviews = Arrays.asList(review);

    Link link1 =
        linkTo(methodOn(ReviewController.class).findAll()).withRel("findAllDoctors");

    Link link2 =
        linkTo(methodOn(ClinicController.class).findAll()).withRel("findAllClinics");

    Resources<DoctorReview> resource = new
Resources<>(allReviews,link1,link2);

    return resource;
}
```

SPRING REST CLIENT-REST TEMPLATE

Consuming REST with REST Template

- REST web service can be consumed programmatically.
- Spring convenient template class is called RestTemplate.
 - **one-line that can bind the data to custom domain types.**
- It leverages the same marshallers that the server uses.
- Can use JAXB marshaller, as well as JSON marshaller

Rest Template

- Rest Template is used to create applications that consume REST
- Its Autowired to the controller class.
- **exchange()**
 - Returns ResponseEntity containing an object mapped from the response body
 - Can Communicate with any HTTP Method
 - Takes the following arguments
 - URL,
 - HttpMethod,
 - Return type for Exchange() method.

Rest Template

- **getForObject**(url, java.lang.Class<T>)
 - Retrieve an Entity by doing a GET on the specified URL.
 - Class<T> is the response class Type
 - retrieves an entity.
- **getForEntity**(url, java.lang.Class<T>)
 - Retrieve a representation by doing a GET on the URL
 - Class<T> is the response class Type
 - It returns **ResponseEntity<?>**

Rest Template

- **headForHeaders()** :
 - Retrieves all headers
- **delete()**
 - Deletes the resources at the given URL.
- **put()**
 - Creates the new resource for the given URL.
- **postForEntity():**
 - Creates a news resource using HTTP POST method.

Rest Template -Exchange

```
HttpHeaders headers = new HttpHeaders();
```

```
headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON  
));
```

```
HttpEntity<String> entity = new HttpEntity<String>(headers);
```

```
restTemplate.exchange(" http://localhost:8080/products",
```

```
    HttpMethod.GET, entity, String.class).getBody();
```

getForObject

@Autowired

RestTemplate *restTemplate*;

String obj =

restTemplate.getForObject("http://localhost:4040/person/all",
String.class);

System.*out.println(obj)*;

PostForObject

- Used to create a new Resource
- ***postForLocation()***
 - returns the URI of the newly created Resource
- ***postForObject()***
 - Returns the Resource itself.
- `RestTemplate restTemplate = new RestTemplate();`
- `HttpEntity<Book> request = new HttpEntity<>(new Book("Java"));`
- `Book bk = restTemplate.postForObject(url, request, Book.class);`

PostForLocation

- Returns the *Location* of that newly created Resource:

```
HttpEntity<Book> request = new HttpEntity<>(new Book("bar"));
```

```
URI location = restTemplate.postForLocation(url, request);
```

Put

```
RestTemplate restTemplate = new RestTemplate();
```

```
String url = "http://localhost:8080/spring-  
rest/data/putdata/{id}/{name}";
```

```
Map<String, String> map = new HashMap<String, String>();
```

```
map.put("id", "100"); map.put("name", "Ram");
```

```
Address address = new Address("Tirupur,"TN");
```

```
restTemplate.put(url, address, map);
```


Delete

- Used to remove an existing Resource

```
String url = "http://localhost:8080/spring-  
rest/data/delete/{name}/{village}";
```

```
Map<String, String> map = new HashMap<String, String>();
```

```
map.put("name", "Mahesh");
```

```
map.put("village", "Madipakkam");
```

```
restTemplate.delete(url, map);
```

headForHeaders

- **Used to Retrieve Headers**
 - Has overloaded Methods
- `HttpHeaders httpHeaders = restTemplate.headForHeaders(url);`
- `System.out.println(httpHeaders.getDate());`
- `System.out.println(httpHeaders.getContentType());`

Custom Exception Class

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class RecordNotFoundException extends RuntimeException
{
    public RecordNotFoundException(String message) {
        super(message);
    }
}
```

Exception Handling

```
@GetMapping("/employees/{id}")  
Employee getEmployeeById(@PathVariable Long id)  
{  
    return repository.findById(id)  
        .orElseThrow(() -> new  
            RecordNotFoundException("Employee id '" + id + "' does not  
            exist"));  
}
```

Best practices

- Use Nouns for Resource Identification
- Use Proper Http Headers for Serialization formats
- Get Method and Query Parameter should not alter the state
- Always use plurals when you name resources.
- Use Appropriate Http Response Status Codes
- Field Name casing Conventions
- Provide Searching, Sorting and Pagination
- Use Versioning of API
- Provide Links for Navigating through API – Hateos
- Handle Error Properly