JAVA Spring

Agenda

- JAVA Framework Introduction
- Spring Framework
  - What
  - Architecture
  - Loose Coupling via IoC, DIP, DI & IoC Container
  - Beans & its scope
  - Spring IoC Metadata Options
    - Dependency injection via XML
    - Dependency injection via Annotation
  - Annotations in Spring Framework
- Spring Data JPA
  - What & Why
  - Spring Data JPA API
  - How
    - JPA with Spring Boot (High level introduction)
    - Implementation details
    - Custom Queries
- Spring Boot
  - What & Why
  - Key concepts
    - Initializr
    - Auto configuration
    - Annotations
  - Spring Boot details with example
- AoP
  - Whay & Why
  - Core Concepts
  - Implementation
  - Implementation with Spring Boot

JAVA Frameworks

https://www.javatpoint.com/what-is-framework-in-java

Multiple frameworks are available in JAVA, solving different problems.

Popular JAVA Frameworks

Spring Framework: What

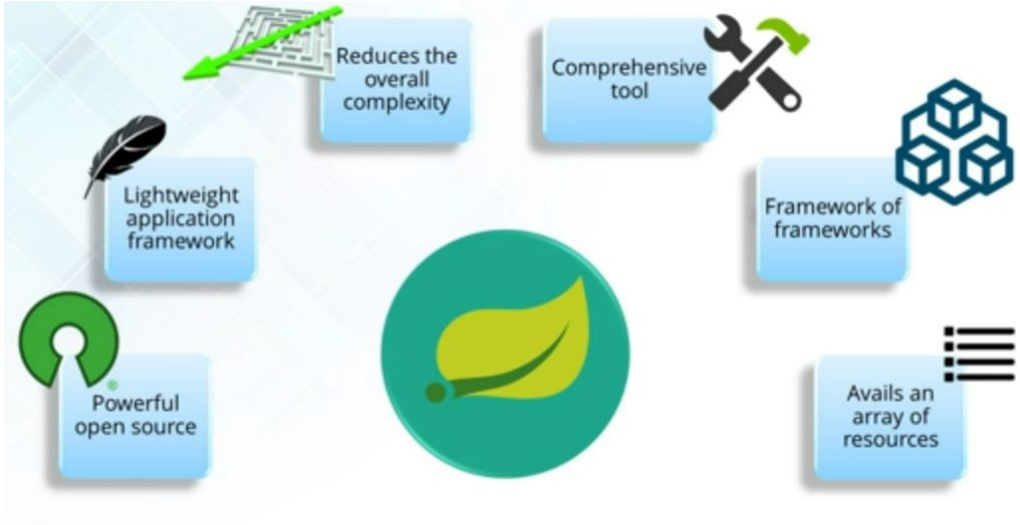
Spring is the **most popular application development framework** which greatly **simplifies** the development of java based enterprise applications.

Spring framework was initially written by **Rod Johnson** and was first open sourced under the **Apache 2.0 license** in June **2003**.

The Spring framework helps in developing **loosely coupled** and **highly cohesive** systems. Loose coupling is achieved by Spring's Inversion of Control (**IoC**) feature and high cohesion is achieved by Spring's Aspect oriented programming (**AOP**) feature.

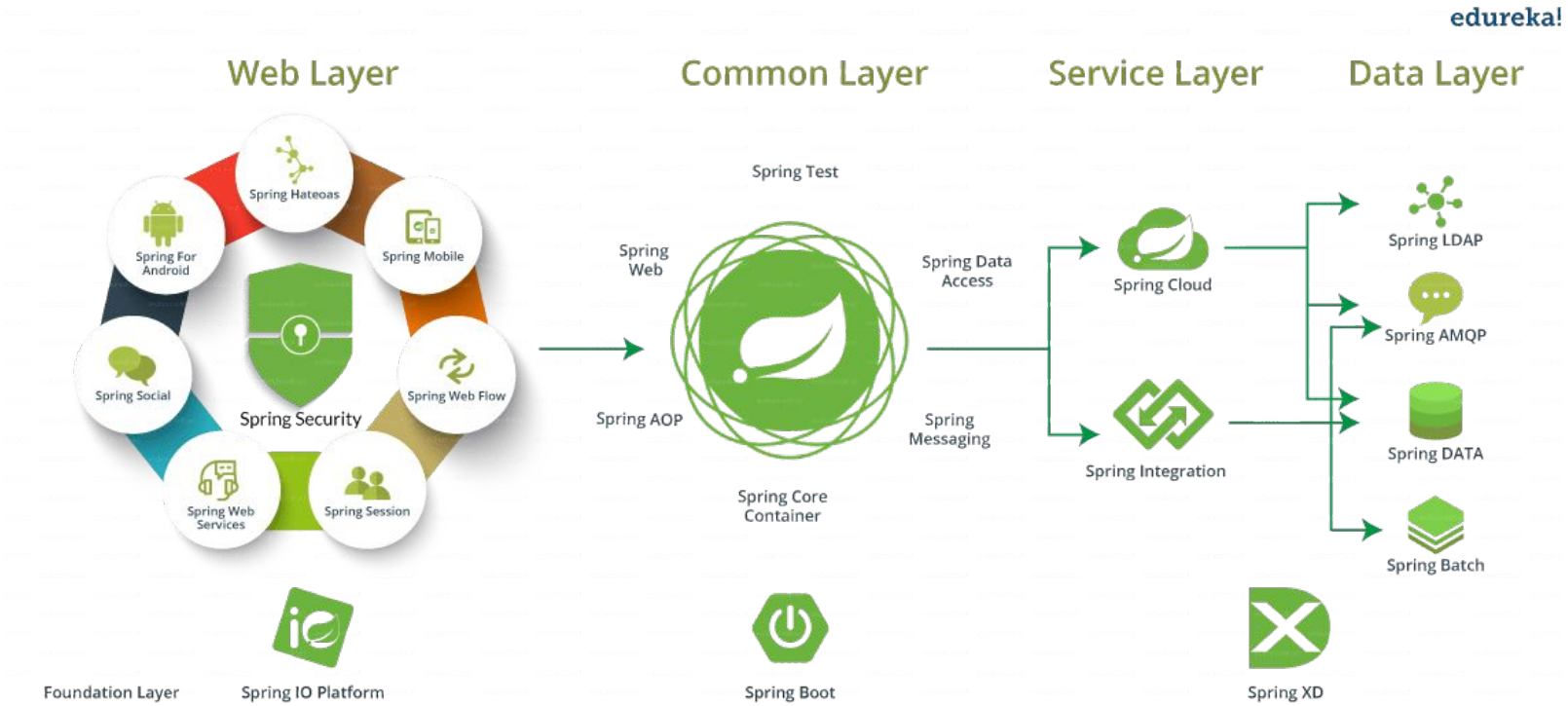
The term "Spring" means different things in different contexts. It can be used to refer to the Spring Framework project itself, which is where it all started. Over time, other Spring projects have been built on top of the Spring Framework. Most often, when people say "Spring", they mean the entire family of projects.

Features of Spring Framework



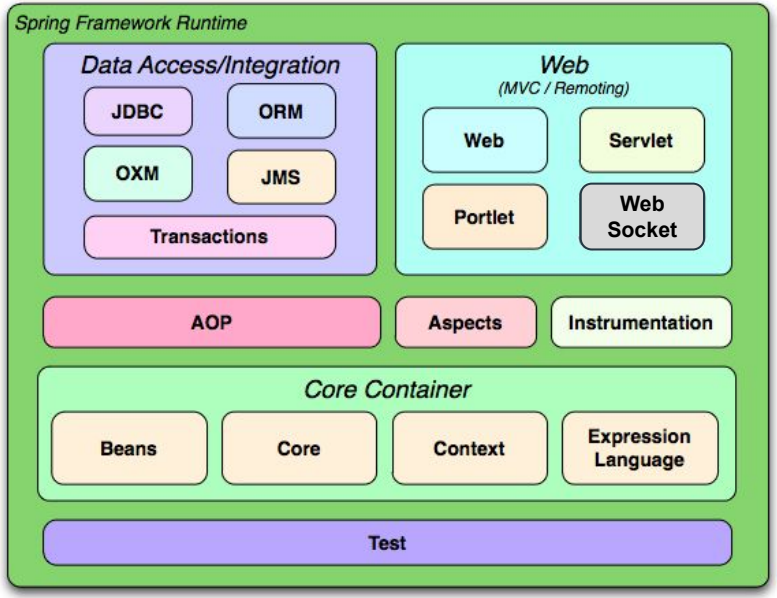
Spring Projects

https://spring.io/projects



Spring Framework Architecture

The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, Hibernate, and iBatis.
The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
The Java Messaging Service (**JMS**) module contains features for producing and consuming messages.
The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.



The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
The **Web-MVC** module contains Spring's implementation for web applications.
The **Web-Socket** module provides support for WebSocket-based, communication between the client and the server in web applications.
The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.

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

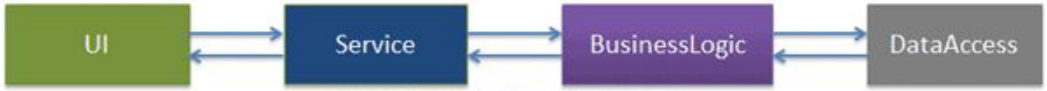
The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

IoC

Spring Framework: IoC

https://www.tutorialsteacher.com/ioc/inversion-of-control

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



In the above example, CustomerBusinessLogic and DataAccess are tightly coupled classes because the CustomerBusinessLogic class includes the reference of the concrete DataAccess class. It also creates an object of DataAccess class and manages the lifetime of the object.

Problems in the above example classes:

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

The following is an example of BusinessLogic and DataAccess classes for a customer.

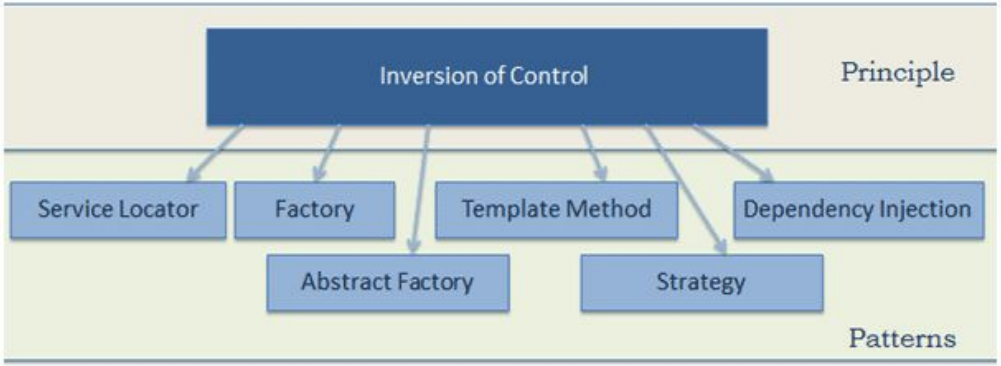
```
public class CustomerBusinessLogic
{
    DataAccess _dataAccess;

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

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

public class DataAccess
{
    public DataAccess()
    {
    }

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

Spring Framework: IoC

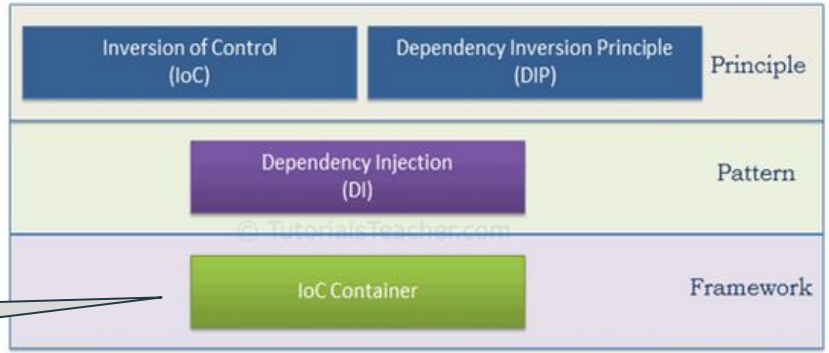
Solution: To solve all of the above problems and get a loosely coupled design, we can use the **IoC and DIP principles together.** Remember, IoC is a principle, not a pattern. It just gives high-level design guidelines but does not give implementation details. You are free to implement the IoC principle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

What: Inversion of Control (IoC) is a **design principle** (although, some people refer to it as a pattern). As the name suggests, it is used to **invert** different kinds of **controls** in **object-oriented design** to achieve **loose coupling**. Here, controls refer to any additional responsibilities a class has, other than its main responsibility. This include control over the flow of an dependent object creation and binding. The IoC principle helps in **designing loosely coupled classes** which make them **testable**, **maintainable** and **extensible**.

Patterns implementing IoC Principle



Relation among different entities

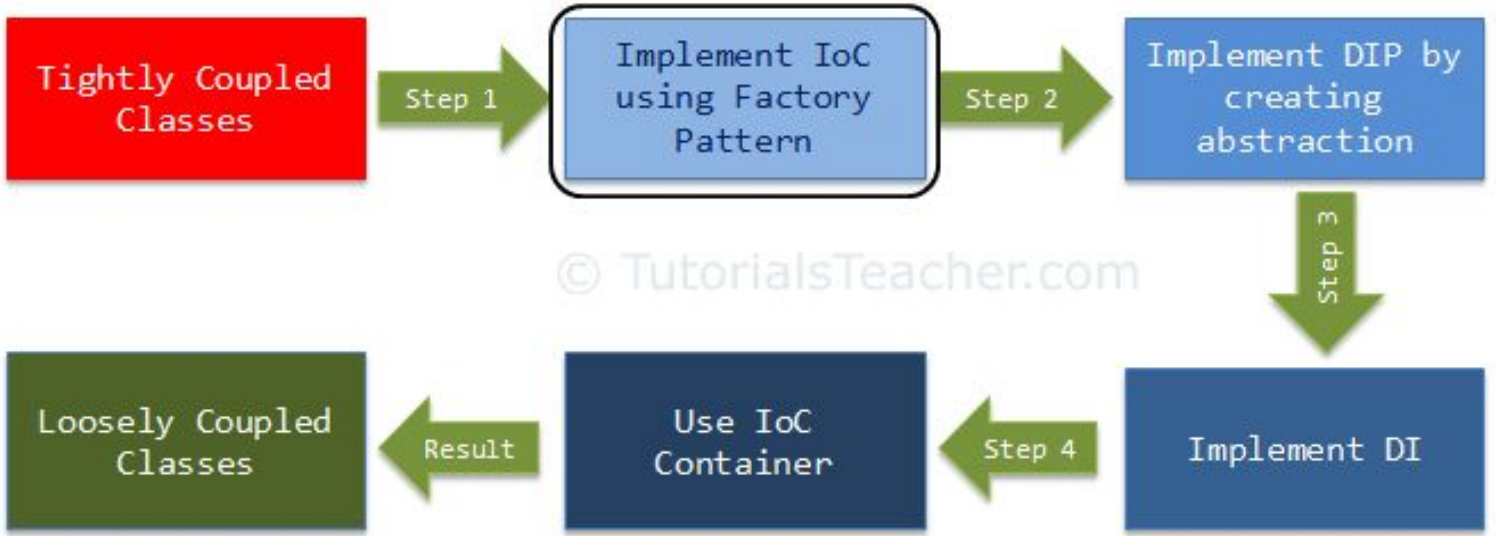


Spring Core

https://www.tutorialsteacher.com/ioc/introduction

Spring Framework: IoC

In coming slides we will try following steps to get Loose coupling



Spring Framework: Loose coupling: Factory Pattern

Step 1: ***Factory* pattern** to implement IoC in the above example, as the first step towards attaining loosely coupled classes.

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

Example: DataAccess Factory - C#

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

Problem which is solved: CustomerBusinessLogic class uses the DataAccessFactory.GetCustomerDataAccessObj() method to get an object of the DataAccess class instead of creating it using the *new* keyword. Thus, we have inverted the control of creating an object of a dependent class from the CustomerBusinessLogic class to the DataAccessFactory class.

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

Example: Use Factory Class to Retrieve Object - C#

```
public class CustomerBusinessLogic
{

    public CustomerBusinessLogic()
    {
    }

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

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

Problem which Persists: In the above example, we implemented the factory pattern to achieve IoC. But, the CustomerBusinessLogic class uses the concrete DataAccess class. Therefore, it is still tightly coupled, even though we have inverted the dependent object creation to the factory class.

Spring Framework: IoC: Factory Pattern + DIP

What: DIP (Dependency Inversion Principle) is one of the SOLID object-oriented principle invented by Robert Martin . As per DIP

1. High-level modules should not depend on low-level modules. Both should depend on the **abstraction**.
2. Abstractions should not depend on details. Details should depend on abstractions.

Problem which is solved: CustomerBusinessLogic does not depend on the concrete DataAccess class, instead it includes a reference of the ICustomerDataAccess interface. So now, we can easily use another class which implements ICustomerDataAccess with a different implementation without touching CustomerBusinessLogic class.

Problem which Persists: The problem with the above example is that we used DataAccessFactory inside the CustomerBusinessLogic class. So, suppose there is another implementation of DataAccessFactory and we want to use that new class inside CustomerBusinessLogic. Then, we need to change the source code of the CustomerBusinessLogic class as well. Also application developer needs to write factories and maintain them.

Note: DIP is needed, but DIP with Factory pattern is not a solution

Redinfed using Interface

Example: DIP Implementation - C#

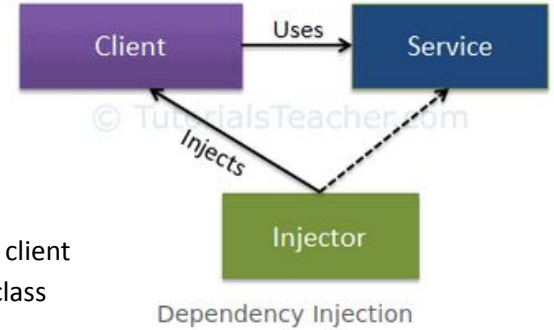
```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}

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

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

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

public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

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

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

Spring Framework: Loose coupling: DI

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

The Dependency Injection pattern involves 3 types of classes.

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



Dependency Injection

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

Types of Dependency Injection

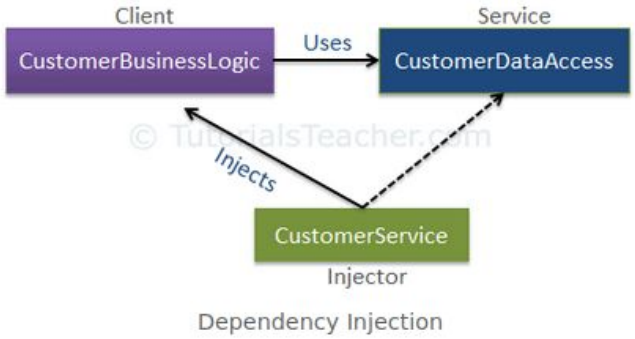
As you have seen above, the injector class injects the service (dependency) to the client (dependent). The injector class injects dependencies broadly in three ways: through a constructor, through a property, or through a method.

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

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

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

Spring Framework: Loose Coupling: DI



Problem which is solved:
CustomerBusinessLogic needn't be changed in any case

Problem which Persist:

Application developers needs to write injector module for dependency injection & management code

Example: Constructor Injection - C#

```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _dataAccess;

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

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

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

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

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

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

Example: Inject Dependency - C#

```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

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

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

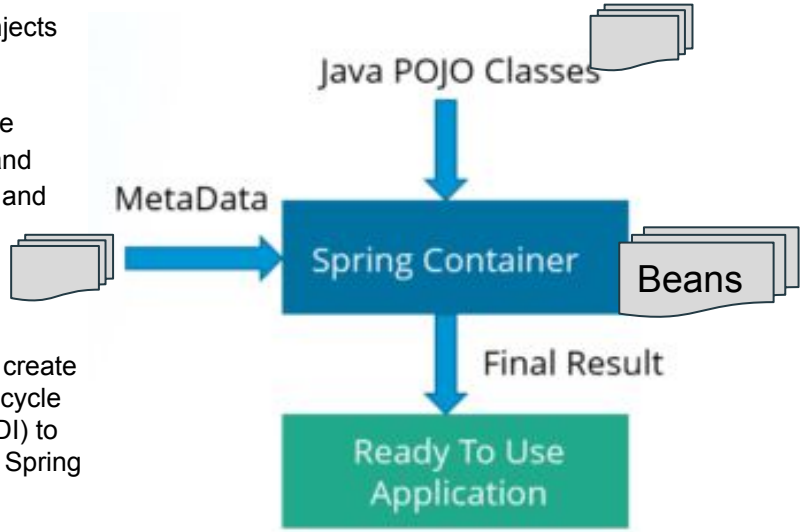
Spring Framework: Loose coupling: DI implementation
Framework: IoC container

IoC Container (a.k.a. DI Container) is a framework for implementing **automatic dependency injection**. It manages object creation and it's life-time, and also injects dependencies to the class.

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

SPRING is one of the IoC container

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application. These objects are called Spring Beans
The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram is a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes with configuration metadata to produce a fully configured and executable system or application.



Spring Framework: IoC container: Example

POJO Class

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

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

Metadata (XML based)

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

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

</beans>
```

Note:
Two Types of IoC containers in Spring

1. BeanFactory
2. **ApplicationContext**

The ApplicationContext container includes all functionality of the BeanFactory container, so it is generally recommended over the BeanFactory.

Ready to use application

```
package com.tutorialspoint;

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

public class MainApp {
   public static void main(String[] args) {

      ApplicationContext context = new FileSystemXmlApplicationContext
            ("C:/Users/ZARA/workspace/HelloSpring/src/Beans.xml");

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

Spring Framework: Bean

A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.
The bean definition contains the information called configuration metadata which is needed for the container to know the followings:

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies

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

Bean Properties

Properties	Description
class	This attribute is mandatory and specify the bean class to be used to create the bean.
name	This attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
scope	This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter.
constructor-arg	This is used to inject the dependencies and will be discussed in next chapters.
properties	This is used to inject the dependencies and will be discussed in next chapters.
autowiring mode	This is used to inject the dependencies and will be discussed in next chapters.
lazy-initialization mode	A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
initialization method	A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter.
destruction method	A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter.

Spring Framework: Bean Scope

The most commonly used ApplicationContext implementations are:
FileSystemXmlApplicationContext: This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
ClassPathXmlApplicationContext This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
WebXmlApplicationContext: This container loads the XML file with definitions of all beans from within a web application.

Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

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

Spring Framework: Metadata

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

1. XML based configuration file.
2. Annotation-based configuration

Originally, Spring supported defining beans through either properties or an XML file. Since the release of JDK 5 and Spring's support of Java annotations, Spring (starting from Spring 2.5) also supports using Java annotations when configuring ApplicationContext.

Q. So, which one is better, XML or annotations?

Ans. Each approach has its pros and cons. Using an XML file can externalize all configuration from Java code, while annotations allow the developer to define and view the DI setup from within the code. Spring also supports a mix of the two approaches in a single ApplicationContext. One common approach is to define the application infrastructure (for example, data source, transaction manager, JMS connection factory, or JMX) in an XML file, while defining the DI configuration (injectable beans and beans' dependencies) in annotations.

Spring Framework: Dependency Injection

```java
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker) {
      System.out.println("Inside TextEditor constructor." );
      this.spellChecker = spellChecker;
   }
   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}
```

```java
package com.tutorialspoint;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling() {
      System.out.println("Inside checkSpelling." );
   }

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
      <constructor-arg ref="spellChecker"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

constructor based dependency injection

Spring Framework: Dependency Injection

```java
package com.tutorialspoint;

public class TextEditor {
   private SpellChecker spellChecker;

   // a setter method to inject the dependency.
   public void setSpellChecker(SpellChecker spellChecker) {
      System.out.println("Inside setSpellChecker." );
      this.spellChecker = spellChecker;
   }
   // a getter method to return spellChecker
   public SpellChecker getSpellChecker() {
      return spellChecker;
   }

   public void spellCheck() {
      spellChecker.checkSpelling();
   }
}
```

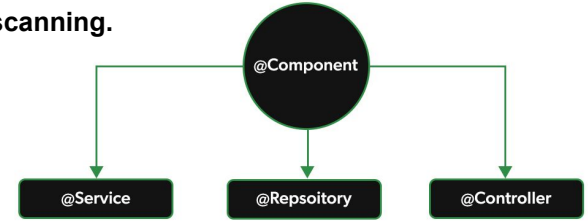
```java
package com.tutorialspoint;

public class SpellChecker {
   public SpellChecker(){
      System.out.println("Inside SpellChecker constructor." );
   }

   public void checkSpelling() {
      System.out.println("Inside checkSpelling." );
   }

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
      <property name="spellChecker" ref="spellChecker"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

setter based dependency injection

Spring Framework: IoC: Annotation

1. Can also declare beans using the *@Bean* annotation in a @Configuration class - Will skip
2. Can mark the class with one of the annotations from the *org.springframework.stereotype* package, and leave the rest to **component scanning.**



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

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

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

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

Example showing usage of @SpringBootApplication (includes ComponentScan), @Autowired, @Controller, @Service

StudentApplication.java

```java
package com.example.student;

import ...

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

StudentController.java

```java
package com.example.student;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class StudentController {

    2 usages
    @Autowired
    private StudentService studentService;

    @GetMapping
    public String getStudents() { return studentService.getStudents();

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
    public String saveStudents(@RequestBody StudentModel student) {
        System.out.println(student);
        return studentService.saveStudents(student);
    }
}
```

StudentService.java

```java
package com.example.student;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

1 usage
@Service
public class StudentService {

    2 usages
    @Autowired
    private StudentRepository repository;

    1 usage
    public String getStudents() {
        return repository.findAll().toString();
    }

    1 usage
    public String saveStudents(StudentModel student) {
        repository.save(student);
        return "Saved";
    }
}
```

StudentRepository.java

```java
package com.example.student;

import org.springframework.data.jpa.repository.JpaRepository;

1 usage
public interface StudentRepository extends JpaRepository<StudentModel, Long> {
}
```

Spring Framework: IoC: Annotation

https://www.javatpoint.com/spring-boot-annotations
https://stackoverflow.com/questions/47954852/why-do-we-need-componentscan-in-spring
https://www.baeldung.com/spring-componentscan-vs-enableautoconfiguration

Spring Boot and Web Annotations

Use annotations to configure your web application.

T **@SpringBootApplication** - uses @Configuration, @EnableAutoConfiguration and @ComponentScan.

T **@EnableAutoConfiguration** - make Spring guess the configuration based on the classpath.

T **@Controller** - marks the class as web controller, capable of handling the requests.

T **@RestController** - a convenience annotation of a @Controller and @ResponseBody.

M T **@ResponseBody** - makes Spring bind method's return value to the web response body.

M **@RequestMapping** - specify on the method in the controller, to map a HTTP request to the URL to this method.

P **@RequestParam** - bind HTTP parameters into method arguments.

P **@PathVariable** - binds placeholder from the URI to the method parameter.

Spring Framework Annotations

Spring uses dependancy injection to configure and bind your appliation together.

T **@Configuration** - used to mark a class as a source of the bean definitions.

T **@ComponentScan** - makes Spring scan the packages configured with it for the @Configuration classes.

T **@Import** - loads additional configuration. This one works even when you specify the beans in an XML file.

T **@Component** - turns the class into a Spring bean at the auto-scan time.

T **@Service** - tells Spring that it's safe to manage @Components with more freedom than regular components.

C F M **@Autowired** - wires the application parts together, on the fields, constructors, or methods in a component.

M **@Bean** - specifies a returned bean to be managed by Spring context. The returned bean has the same name as the factory method.

M **@Lookup** - tells Spring to return an instance of the method's return type when we invoke it.

T M **@Primary** - gives higher preference to a bean when there are multiple beans of the same type.

C F M **@Required** - shows that the setter method must be configured to be dependency-injected with a value at configuration time.

C F M **@Value** - used to assign values into fields in Spring-managed beans. It's compatible with the constructor, setter, and field injection.

T M **@DependsOn** - makes Spring initialize other beans before the annotated one.

T M **@Lazy** - makes beans to initialize lazily. @Lazy annotation may be used on any class directly or indirectly annotated with @Component or on methods annotated with @Bean.

T M **@Scope** - used to define the scope of a @Component class or a @Bean definition and can be either singleton, prototype, request, session, globalSession, or custom scope.

T **@Profile** - adds beans to the application only when that profile is active.

Legend: T - Class F - Field Annotation C - Constructor Annotation M - Method P - Parameter

Spring Data JPA

https://www.youtube.com/watch?v=8SGI_XS5OPw

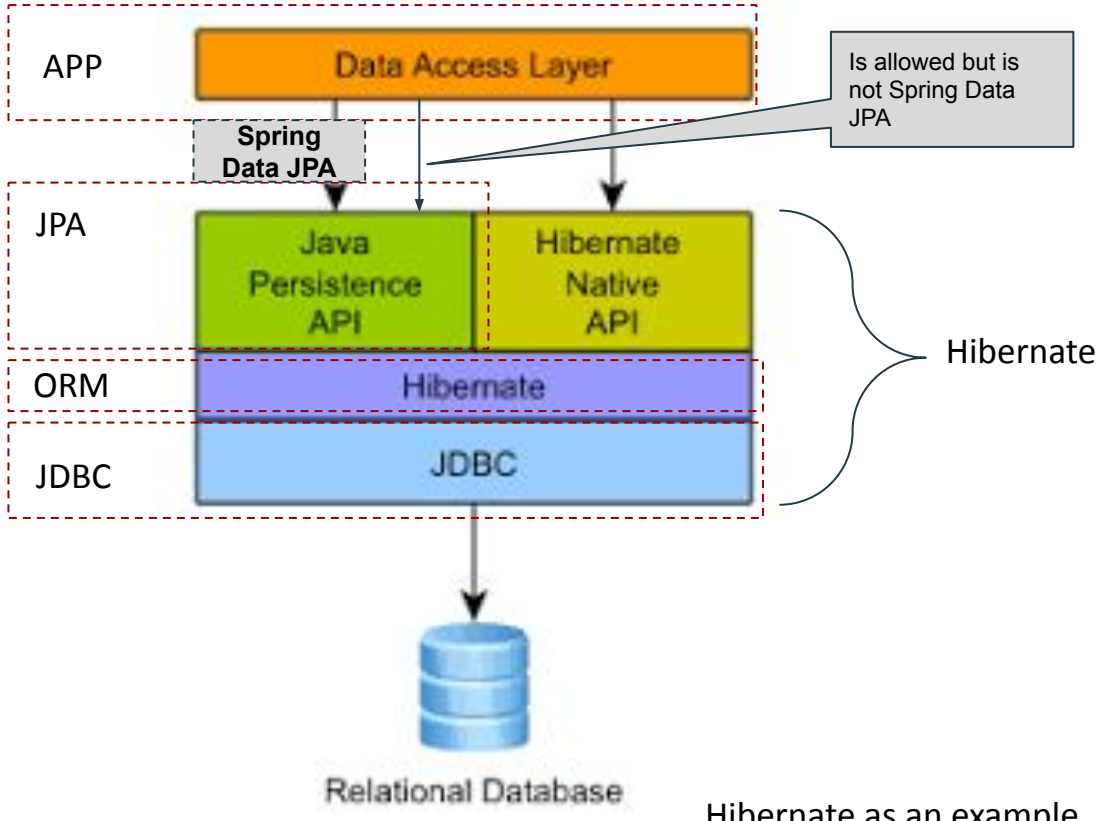
Spring Data JPA

What:
Extra layer of abstraction on top of JPA providers like hibernate.

Why:
Reduce the amount of boilerplate code required to implement a data access layer

Example: To save any entity, earlier user needs to write below logic to find whether to persist or to merge as part of DAO layer, which is provided by Spring Data JPA

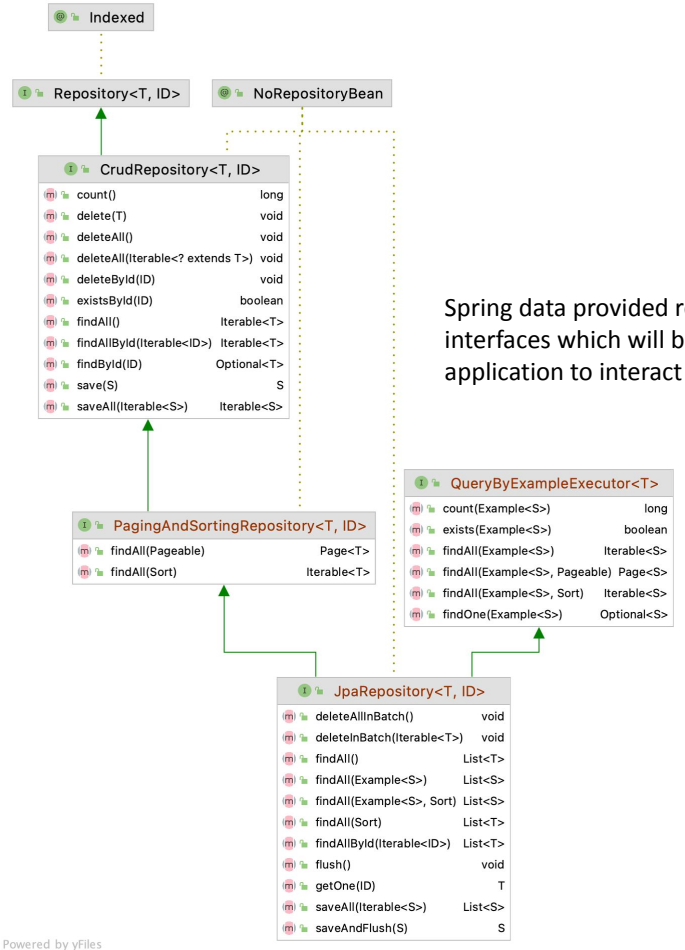
```
@Transactional
public <S extends T> S save(S entity) {
    if (this.entityInformation.isNew(entity)) {
        this.em.persist(entity);
        return entity;
    } else {
        return this.em.merge(entity);
    }
}
```



Hibernate as an example

Spring Data JPA API

APIs - Many APIs will ease every application to write boiler plate logic in DAO layer like findAll, saveAll, findById etc.
This all comes ready made with Spring JPA



Spring data provided repository interfaces which will be used by application to interact with DB

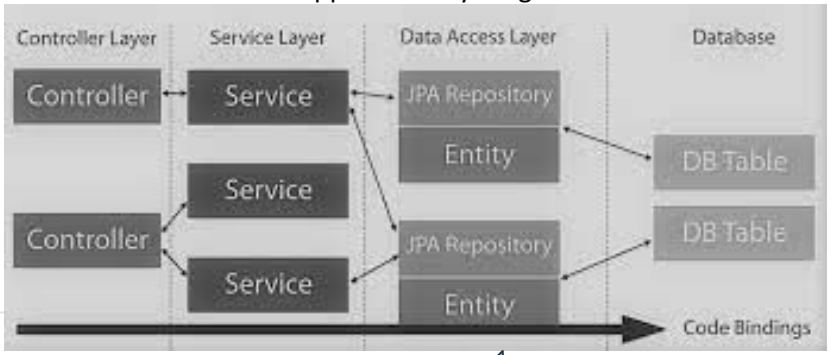
Example on Without Spring Data JPA:
https://www.javatpoint.com/spring-and-jpa-integration

Spring Data JPA - How

Application developer just need to define entity and define repository which extends one of the three interface defined in previous slide per entity.

Note: There are alternatives to define repository per entity

Application layering



The Repository and Entity Bean represent the DAO layer in the application

Java

Sample JPA Repository

```java
package com.example.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.example.demo.modal.Employee;

// @Repository is a Spring annotation that
// indicates that the decorated class is a repository.
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long>{
    ArrayList<Employee> findAllEmployee();
}
```

Spring Data JPA with Spring Boot

Configurations in application yaml

application.yaml

```yaml
spring:
  datasource:
    url: jdbc:postgresql://${INVENTORY_DB_HOST:mecm-postgres}:${INVENTORY_DB_PORT:5432}/${INVENTORY_DB:inventorydb}
    username: ${INVENTORY_DB_USER:inventory}
    password: ${INVENTORY_DB_PASSWORD:}
    initialization-mode: always
    schema: classpath:db/migration/V1__Add_Inventory_Task_Table.sql
  jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.PostgreSQLDialect
        ddl-auto: validate
```

JDBC connectivity

JPA Configuration

Spring boot provided JPA starter kit

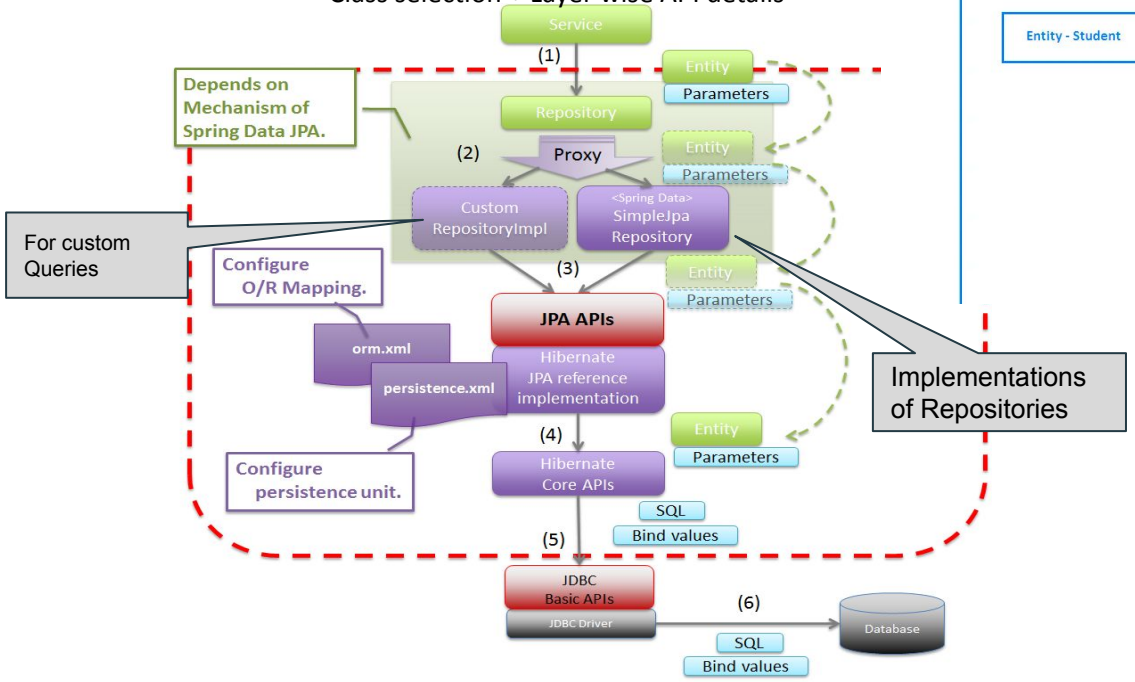
pom.xml (demo)

```xml
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mariadb.jdbc</groupId>
            <artifactId>mariadb-java-client</artifactId>
            <scope>runtime</scope>
        </dependency>
```

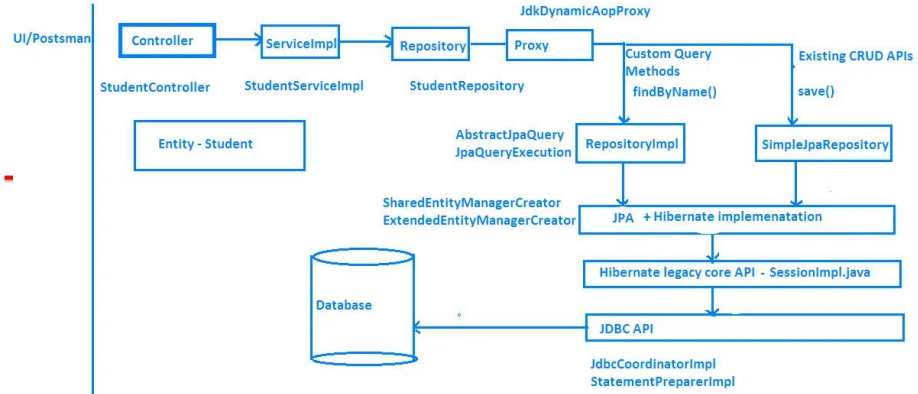
Spring Data JPA - Implementation Class Selection

Spring Data JPA uses generics and reflection to generate the concrete implementation of the interface we define.

Class selection + Layer wise API details



Example with Student Entity



Spring Data JPA - Custom Queries

```java
import java.util.List;
import javax.transaction.Transactional;
import org.edgegallery.mecm.inventory.model.AppDnsRule;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

/**
 * App Dns rule repository.
 */
SeanGao123 +1
public interface AppDnsRuleRepository extends CrudRepository<AppDnsRule, String>, BaseRepository<AppDnsRule> {

    1 usage   SeanGao123
    @Transactional
    @Modifying
    @Query("delete from AppDnsRule m where m.tenantId=:tenantId")
    void deleteByTenantId(@Param("tenantId") String tenantId);

    3 usages   SeanGao123
    @Query(value = "SELECT * FROM appdnsruleinventory m WHERE m.tenant_id=:tenantId", nativeQuery = true)
    List<AppDnsRule> findByTenantId(@Param("tenantId") String tenantId);

    shashikanth.vh@huawei.com <shashikanth.vh@huawei.com>
    @Query(value = "SELECT * FROM appdnsruleinventory m WHERE m.role=:role", nativeQuery = true)
    List<AppDnsRule> findByUserRole(@Param("role") String role);
}
```

Example: User defined APIs backed by custom SQL queries

Spring Boot

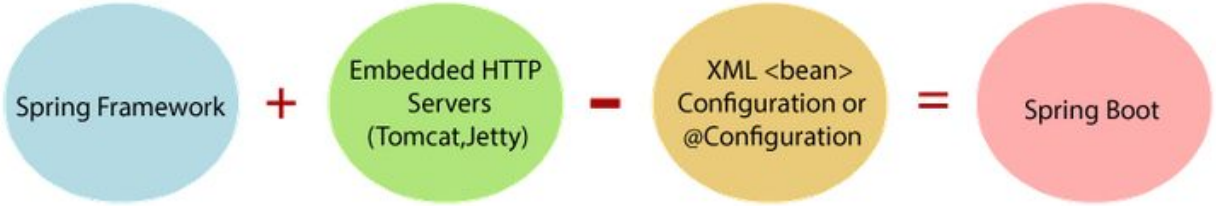
https://www.youtube.com/watch?v=35EQXmHKZYs
https://www.youtube.com/watch?v=9SGDpanrc8U

https://www.javatpoint.com/spring-boot-tutorial - Good read for SpringBoot details

Spring Boot: What & Why

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

It is a Spring module that provides the **RAD (*Rapid Application Development*)** feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.



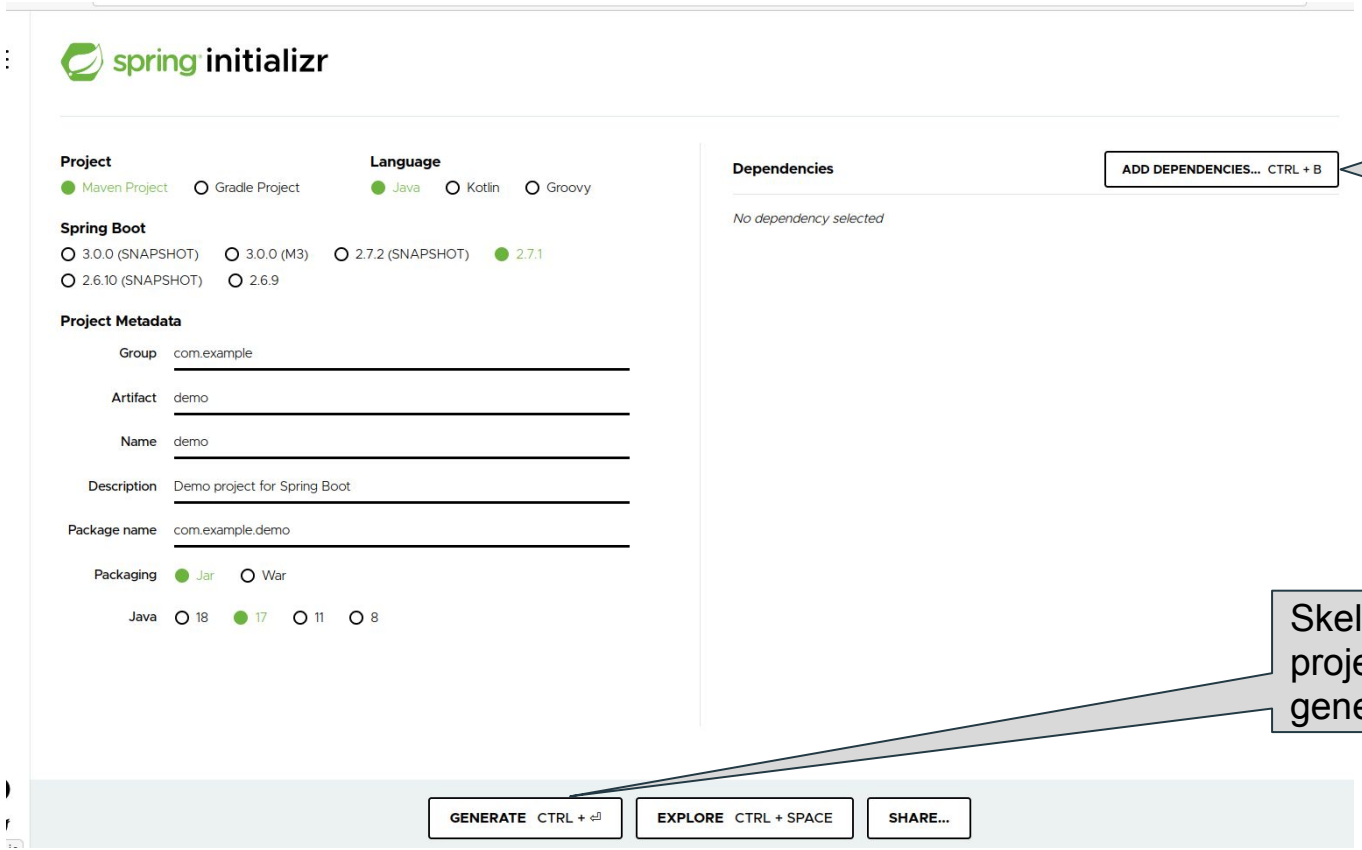
Why Spring Boot?

You can choose Spring Boot because of the features and benefits it offers as given here −

- Provides multiple starters: Ready to use starter kits (have seen one for Spring data JPA).
- Auto-Configuration: It provides a flexible way to configure Database Transactions (implementation classes automatically injected), XML configurations etc.
- It includes Embedded Servlet Container

Spring Boot: Initializr

https://start.spring.io/



User add dependencies

Skeleton of project is generated

Spring Boot: Auto Configuration

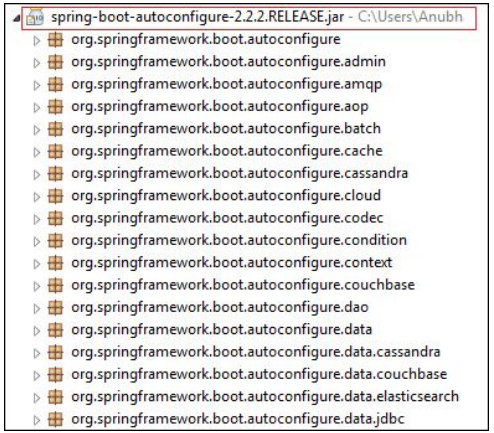
https://www.javatpoint.com/spring-boot-auto-configuration

Spring Boot auto-configuration automatically configures the Spring application based on the jar dependencies that we have added. We can enable the auto-configuration feature by using the annotation **@EnableAutoConfiguration**

1. if the H2 database Jar is present in the classpath and we have not configured any beans related to the database manually, the Spring Boot's auto-configuration feature automatically configures it in the project.
2. When we add the **spring-boot-starter-web** dependency in the project, Spring Boot auto-configuration looks for the Spring MVC is on the classpath. It auto-configures **dispatcherServlet**, a default **error page,** and **web jars**.
3. Similarly, when we add the **spring-boot-starter-data-jpa** dependency, we see that Spring Boot Auto-configuration, auto-configures a **datasource** and an **Entity Manager**.

All auto-configuration logic is implemented in **spring-boot-autoconfigure.jar**, as shown in the following figure.

Kindly refer attached link to see what all manual task application needs to do without auto configuration feature.



Spring Boot: Annotation

Spring Boot Annotations

@EnableAutoConfiguration

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

@SpringBootApplication

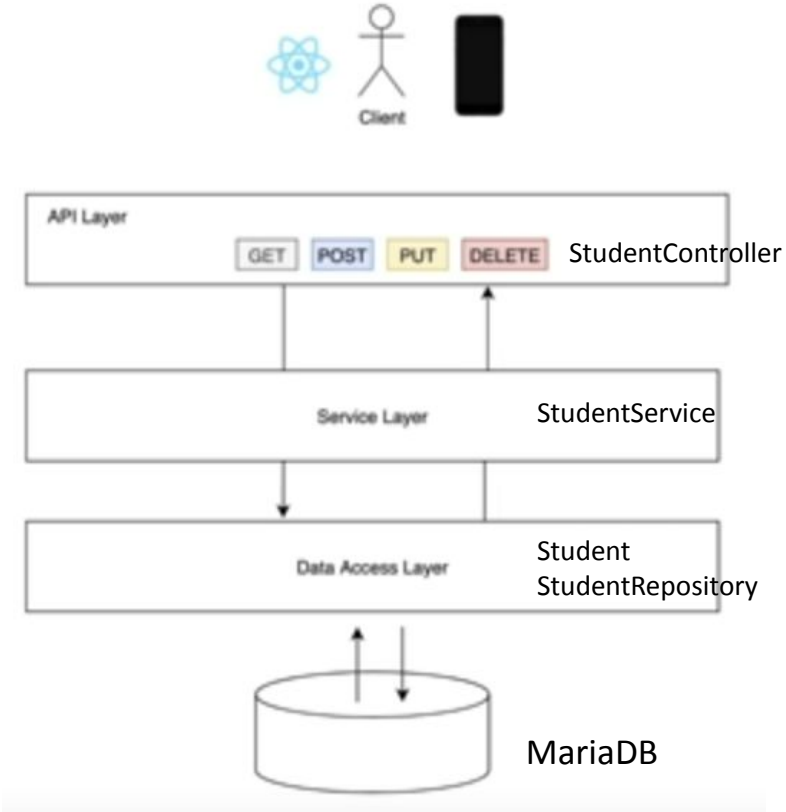
This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the @SpringBootApplication must be kept in the base package. The one thing that the @SpringBootApplication does is a component scan. But it will scan only its sub-packages. As an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

- @Configuration
- @EnableAutoConfiguration
- @ComponentScan

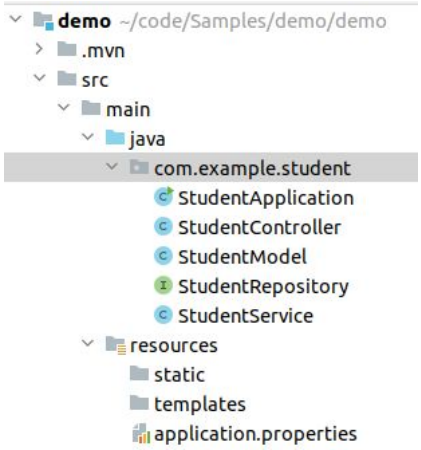
Spring Boot: Details with example

Pre-Requisite: DB instance is UP with user & db configured

1. Project skeleton creation: Create project skeleton with all required dependencies using Spring Initializr - https://start.spring.io/
2. HTTP server creation: Add Controller class using @RestController, Add methods using CRUD operation specific annotations - Verify REST operation.
3. Add Service class and wire it to controller class - Verify Rest Operation
4. Add Application YAML properties for JDBC and JPA
5. Add Model/Entity - Verify schema creation in DB
6. Add repository & Autowire Repository to Service
7. Test POST & then GET



Spring Boot: Details with example



StudentApplication.java

```java
package com.example.student;

import ...

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

application.properties

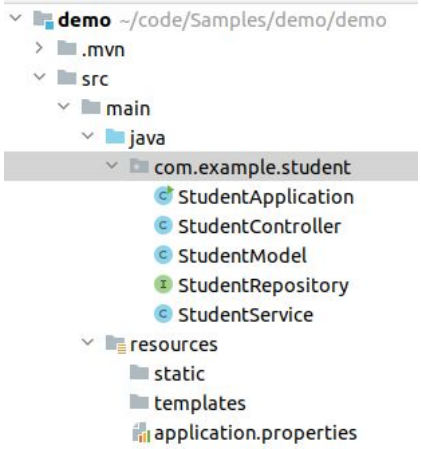
```
spring.datasource.url=jdbc:mariadb://localhost:3306/sdnctl
spring.datasource.username=sdnctl
spring.datasource.password=sdnctl
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDB106Dialect
```

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="htt
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https:/
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mariadb.jdbc</groupId>
            <artifactId>mariadb-java-client</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
```

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

Spring boot maven plugin to produce executable jar

Spring Boot: Details with example



StudentController.java

```java
package com.example.student;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class StudentController {

    2 usages
    @Autowired
    private StudentService studentService;

    @GetMapping
    public String getStudents() { return studentService.getStudents(); }

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
    public String saveStudents(@RequestBody StudentModel student) {
        System.out.println(student);
        return studentService.saveStudents(student);
    }
}
```

StudentService.java

```java
package com.example.student;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

1 usage
@Service
public class StudentService {

    2 usages
    @Autowired
    private StudentRepository repository;

    1 usage
    public String getStudents() {
        return repository.findAll().toString();
    }

    1 usage
    public String saveStudents(StudentModel student) {
        repository.save(student);
        return "Saved";
    }
}
```

StudentRepository.java

```java
package com.example.student;

import org.springframework.data.jpa.repository.JpaRepository;

1 usage
public interface StudentRepository extends JpaRepository<StudentModel, Long> {
}
```

StudentModel.java

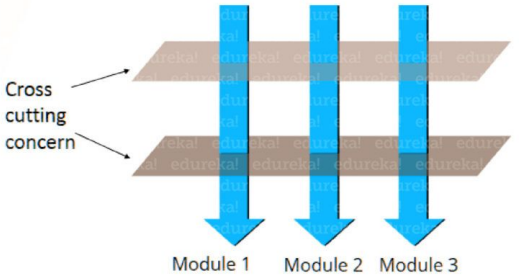
```java
package com.example.student;

import lombok.*;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

3 usages
@Getter
@Setter
@ToString
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "student_details")
public class StudentModel {
    @Id
    private String studentName;
    private long studentId;
}
```

AoP

https://www.javatpoint.com/spring-aop-tutorial

https://www.youtube.com/watch?v=Ft29HgsePfQ - AOP with springboot

AOP: What & Why



What: Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

Why: It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

```
class A{
public void m1(){...}
public void m2(){...}
public void m3(){...}
public void m4(){...}
public void m5(){...}
public void n1(){...}
public void n2(){...}
public void p1(){...}
public void p2(){...}
public void p3(){...}
}
```

First, let's understand Scenario- Here, I have to maintain a log and send notification after calling methods that start from m. So what is the problem without AOP? Here, We can call methods (that maintains a log and sends notification) from the methods starting with a. In such a scenario, we need to write the code in all the 5 methods. But, in case if a client says in future, I don't have to send a notification, you need to change all the methods. It leads to a maintenance problem. So with AOP, we have below solution.

The solution with AOP– With AOP, we don't have to call methods from the method. We can simply define the additional concern like maintaining a log, sending notification etc. in the method of a class. Its entry is given in the XML file. Suppose in future, if a client says to remove the notifier functionality, we need to change only in the XML file. So, maintenance is easy in AOP.

AOP: Core Concepts

Spring AOP consists of 7 core concepts which are depicted in the following diagram:

1. **Aspect**: The *aspect* is nothing but a class that implements the JEE application concerns which cut through multiple classes, such as transaction management, security etc. Aspects can be a normal class configured through Spring XML configuration. It can also be regular classes annotated using @Aspect annotation.
2. **Joinpoint:** The *joinpoint* is a *candidate* point in the program execution where an aspect can be plugged in. It could be a method that is being called, an exception being thrown, or even a field being modified.
3. **Advice:** *Advice* are the specific actions taken for a particular joinpoint. Basically, they are the methods that get executed when a certain joinpoint meets a matching pointcut in the application.

   5 types of advice: @Before, @AfterReturning, @AfterThrowing, @After, @Around

4. **Pointcut:** A *Pointcut* is an expression that is matched with join points to determine whether advice needs to be executed or not.
5. **Target Object:** These are the objects on which advices are applied. In Spring AOP, a subclass is created at runtime where the target method is overridden and advices are included based on their configuration.
6. **Proxy:** It is an object that is created after applying advice to the target object. In clients perspective, object, the target object, and the proxy object are same.
7. **Weaving:** *Weaving* is the process of linking an aspect with other application types or objects to create an advised object.

1.

AOP: Implementations

AOP implementations are provided by: 1. **AspectJ** 2. Spring AOP 3. JBoss AOP

The **Spring Framework** recommends you to use **Spring AspectJ AOP implementation** over the Spring 1.2 old style dtd based AOP implementation because it provides you more control and it is easy to use.

There are two ways to use Spring AOP AspectJ implementation:

1. By annotation: We are going to learn it here.
2. By xml configuration (schema based): We will learn it in next page.

@Before Example: The AspectJ Before Advice is applied before the actual business logic method. You can perform any operation here such as conversion, authentication etc.

Create a class that contains actual business logic.

File: Operation.java

1

```
package com.javatpoint;
public  class Operation{
  public void msg(){System.out.println("msg method invoked");}
  public int m(){System.out.println("m method invoked");return 2;}
  public int k(){System.out.println("k method invoked");return 3;}
}
```

Now, create the aspect class that contains before advice.

File: TrackOperation.java

2

```
package com.javatpoint;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TrackOperation{
  @Pointcut("execution(* Operation.*(..))")
  public void k(){}//pointcut name

  @Before("k()")//applying pointcut on before advice
  public void myadvice(JoinPoint jp)//it is advice (before advice)
  {
    System.out.println("additional concern");
    //System.out.println("Method Signature: "  + jp.getSignature());
  }
}
```

File: applicationContext.xml

3

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">


  <bean id="opBean" class="com.javatpoint.Operation">  </bean>
  <bean id="trackMyBean" class="com.javatpoint.TrackOperation"></bean>

  <bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"></bean>

</beans>
```

```
public class Test{
  public static void main(String[] args){
    ApplicationContext context = new ClassPathXmlApplic
    Operation e = (Operation) context.getBean("opBean");
    System.out.println("calling msg...");
    e.msg();
    System.out.println("calling m...");
    e.m();
    System.out.println("calling k...");
    e.k();
  }
}
```

4

Output

```
calling msg...
additional concern
msg() method invoked
calling m...
additional concern
m() method invoked
calling k...
additional concern
k() method invoked
```

Change pointcut to

5

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

```
calling msg...
msg() method invoked
calling m...
additional concern
m() method invoked
calling k...
k() method invoked
```

Now you can see additional concern is not printed before k() & msg() method invoked.

AOP: Implementations with Spring boot

https://www.javatpoint.com/spring-boot-aop-before-advice

Step 8: Open the **pom.xml** file and add the following **AOP** dependency. It is a starter for aspect-oriented programming with **Spring AOP** and **AspectJ**.

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

Step 9: Open **Application.java** file and add an annotation **@EnableAspectJAutoProxy.**

```
package com.javatpoint;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.EnableAspectJAutoProxy
@SpringBootApplication
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class AopBeforeAdviceExampleApplication
{
public static void main(String[] args) {
SpringApplication.run(AopBeforeAdviceExampleApplication.class, args);
}
}
```

EmployeeController.java

```
package com.javatpoint.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.javatpoint.model.Employee;
import com.javatpoint.service.EmployeeService;
@RestController
public class EmployeeController
{
@Autowired
private EmployeeService employeeService;
@RequestMapping(value = "/add/employee", method = RequestMethod.GET)
public com.javatpoint.model.Employee addEmployee(@RequestParam("empId") St
{
return employeeService.createEmployee(empId, firstName, secondName);
}
@RequestMapping(value = "/remove/employee", method = RequestMethod.GET)
public String removeEmployee( @RequestParam("empId") String empId)
{
employeeService.deleteEmployee(empId);
return "Employee removed";
}
}
```

EmployeeService.java

```
package com.javatpoint.service;
import org.springframework.stereotype.Service;
import com.javatpoint.model.Employee;
@Service
public class EmployeeService
{
public Employee createEmployee( String empId, String fname, String sname)
{
Employee emp = new Employee();
emp.setEmpId(empId);
emp.setFirstName(fname);
emp.setSecondName(sname);
return emp;
}
public void deleteEmployee(String empId)
{
}
}
```

It enables support for handling components marked with AspectJ's @Aspect annotation. It is used with @Configuration annotation. We can control the type of proxy by using the **proxyTargetClass** attribute. Its default value is **false**. NO XML needed because of this.

AOP: Implementations with Spring boot

EmployeeServiceAspect.java

```
package com.javatpoint.aspect;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;
@Aspect
@Component
public class EmployeeServiceAspect
{
@Before(value = "execution(* com.javatpoint.service.EmployeeService.*(..)) and args(empId, fname, sname)")
public void beforeAdvice(JoinPoint joinPoint, String empId, String fname, String sname) {
System.out.println("Before method:" + joinPoint.getSignature());
System.out.println("Creating Employee with first name - " + fname + ", second name - " + sname + " and id - " + empId);
}
}
```

In the above class:

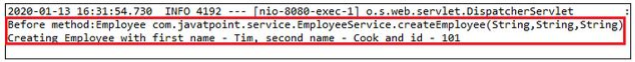
- **execution(expression):** The expression is a method on which advice is to be applied.
- **@Before:** It marks a function as an advice to be executed before method that covered by PointCut.

Step 19: Open the browser and invoke the following URL : *http://localhost:8080/add/employee?empId={id}&firstName={fname}&secondName={sname}*

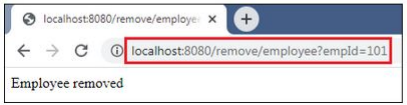
In the above URL, **/add/employee** is the mapping that we have created in the Controller class. We have used two separators **(?)** and **(&)** for separating two values.

In the above output, we have assigned **emId 101, firstName=Tim,** and **secondName=cook.**

Let's have a look at the console. We see that before invoking the **createEmployee**() method of **EmployeeService** class, the method **beforeAdvice()** of **EmployeeServiceAspect** class invokes, as shown below.



Similarly, we can also remove an employee by invoking the URL http://localhost:8080/remove/employee?empId=101. It returns a message **Employee removed**, as shown in the following figure.



In console this will also print what is mentioned by Aspect

Thanks

Frameworks vs Libraries

Library	Framework
Library is the collection of frequently used, pre-compiled classes.	Framework is the collection of libraries.
It is a set of reusable functions used by computer programs.	It is a piece of code that dictates the architecture of your project and aids in programs.
You are in full control when you call a method from a library and the control is then returned.	The code never calls into a framework, instead the framework calls you.
It is in corporate seamlessly into existing projects to add functionality that you can access using an API.	It cannot be seamlessly incorporated into an existing project. Instead it can be used when a new project is started.
They are important in program for linking and binding process.	They provide a standard way to build and deploy applications.
Libraries do no employ an inverted flow of control between itself and its clients.	Framework employs an inverted flow of control between itself and its clients.
Example: jQuery is a JavaScript library that simplifies DOM manipulation.	**Example:** Angular JS is a JavaScript-based framework for dynamic web applications.

Frameworks vs Platform

A platform is a set of hardware and software components that provide a space for developers to build and run applications. Examples of platforms include WordPress and Microsoft Azure. A framework is a software-only app skeleton that includes preset tools, libraries, software development kits, and other components

Principle vs Pattern

Design Principle

Design principles provide high level guidelines to design better software applications. They do not provide implementation guidelines and are not bound to any programming language. The SOLID (SRP, OCP, LSP, ISP, DIP) principles are one of the most popular sets of design principles.

For example, the Single Responsibility Principle (SRP) suggests that a class should have only one reason to change. This is a high-level statement which we can keep in mind while designing or creating classes for our application. SRP does not provide specific implementation steps but it's up to you how you implement SRP in your application.

Design Pattern

Design Pattern provides low-level solutions related to implementation, of commonly occurring object-oriented problems. In other words, design pattern suggests a specific implementation for the specific object-oriented programming problem. For example, if you want to create a class that can only have one object at a time, then you can use the Singleton design pattern which suggests the best way to create a class that can only have one object.

Design patterns are tested by others and are safe to follow, e.g. Gang of Four patterns: Abstract Factory, Factory, Singleton, Command, etc.

Spring	Spring Boot
Spring Framework is a widely used Java EE framework for building applications.	**Spring Boot Framework** is widely used to develop **REST APIs**.
It aims to simplify Java EE development that makes developers more productive.	It aims to shorten the code length and provide the easiest way to develop **Web Applications**.
The primary feature of the Spring Framework is **dependency injection**.	The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement.
It helps to make things simpler by allowing us to develop **loosely coupled** applications.	It helps to create a **stand-alone** application with less configuration.
The developer writes a lot of code (**boilerplate code**) to do the minimal task.	It **reduces** boilerplate code.
To test the Spring project, we need to set up the sever explicitly.	Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc.
It does not provide support for an in-memory database.	It offers several plugins for working with an embedded and **in-memory** database such as **H2**.
Developers manually define dependencies for the Spring project in **pom.xml**.	Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement.

Spring Architecture

The Spring Framework contains a lot of features, which are well-organized in six modules shown in the diagram below.