CS3215: Web Technology TY Div C n D AY 2022-23

Study Material for Section-II-Part-I- Spring Boot

Spring Boot: Overview of Spring Framework, Introduction to Spring Boot, Installing Spring Boot, Build Tool Maven/Gradle/Ant, Core Features, Spring Security, Web Applications, JPA for database connectivity, Working with SQL and NoSQL, Messaging, Testing, Deploying Spring Boot Applications, Monitoring

----

Software Required: One ID: Eclipse/Intellij/Netbeans/**STS**/Visual Code : STS Download : Spring Tools 4 for eclipse for Windows 64-bit

----

JAVA applications in 2000 EJB JAVA API for Server Side

Apache **Struts** 1 is an open-source web application framework for developing Java EE web applications. It uses and extends the Java Servlet API to encourage developers to adopt a model–view–controller architecture. It was originally created by Craig McClanahan and donated to the Apache Foundation in May 2000

[Red Hat 2001 -**Hibernate** ORM is an object–relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database.]

--typically consist of objects that collaborate to form the application proper. **The objects in an application have dependencies on each other.**

--Although the Java platform provides a wealth of application development functionality, **it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers.**

--Although you can use design patterns such as Factory, Abstract Factory, Builder, Decorator, and Service Locator to compose the various classes and object instances that make up an application, these patterns are simply that: best practices given a name, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth.

--Patterns are formalized best practices that you must implement yourself in your application.

--therefore **concern is to provide a formalized means of composing disparate components into a fully working application ready for use.**

--**Need a framework which codifies formalized design patterns using first-hand class objects that you can integrate into your own application(s).**

What is Spring?

--Spring is an open-source lightweight framework (Java platform) that provides comprehensive programming infrastructure and configuration model support for developing Java-based reliable, and scalable enterprise applications by extending your plain old java objects (POJO).

--Spring handles the infrastructure so you can focus on your application.

--It is an extended java platform for J2EE developers.

-- Dependency Injection (DI) is a programming technique that makes a class independent of its dependencies. In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. A 'dependency' is an object that can be used, for example as a service

----

Spring allows you:

--Make a Java method execute in a database transaction without dealing with transaction APIs.

--Make a local Java method a remote procedure without having to deal with remote APIs.

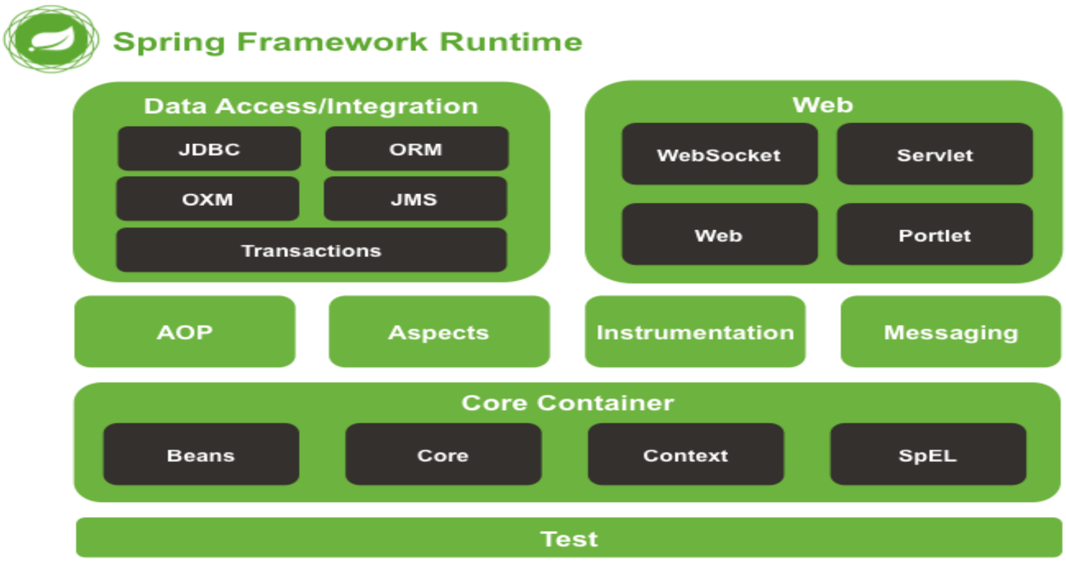
--Make a local Java method a management operation without having to deal with JMX APIs.

--Make a local Java method a message handler without having to deal with JMS APIs.

-- **The Spring Framework *Inversion of Control* (IoC) component provides a formalized means of composing disparate components into a fully working application ready for use.**

The Spring framework can be considered as a collection of sub-frameworks, also called layers, such as Spring AOP. Spring Object-Relational Mapping (Spring ORM). Spring Web Flow, and Spring Web MVC. You can use any of these modules **separately** while constructing a Web application. The modules may also be grouped **together** to provide better functionalities in a Web application**.**

Spring Framework Runtime:



Core Container: consists of the spring-core, spring-beans, spring-ontext, spring-context-support, and spring-expression (Spring Expression Language) modules.

The spring-core and spring-beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features.

The BeanFactory is a sophisticated implementation of the factory pattern.

Context module inherits its features from the Beans module and adds support for internationalization, event propagation, resource loading, a Servlet container,. Java EE features such as EJB, JMX, and basic remoting, Application Context interface, integrating common third-party libraries into a Spring application context

Spring-expression module provides a powerful Expression Language for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the content of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method interceptors and point cuts to cleanly decouple code that implements functionality that should be separated.

Aspects module provides integration with AspectJ.

Instrument module provides class instrumentation support and class loader implementations to be used in certain application servers.

Messaging module with key abstractions from the Spring Integration project such as Message, Message Channel, Message Handler, and others to serve as a foundation for messaging-based applications. The module also includes a set of annotations for mapping messages to methods, similar to the Spring MVC annotation based programming model.

Data Access/Integration Layer: consists of the JDBC, ORM, OXM, JMS, and Transaction modules.

Spring-JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

Spring-TX module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs (Plain Old Java Objects).

Spring-ORM module provides integration layers for popular object-relational mapping APIs, including JPA and Hibernate.

Spring-OXM module provides an abstraction layer that supports Object/XML mapping implementations such as JAXB, Castor, JiBX and XStream.

Spring-jms module (Java Messaging Service) contains features for producing and consuming messages.

Web Layer: consists of the spring-web, spring-webmvc and spring-websocket modules.

Spring-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. It also contains an HTTP client and the web-related parts of Spring's remoting support.

Spring-webmvc module (also known as the Web-Servlet module) contains Spring's model-view-controller (MVC) and REST Web Services implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms and integrates with all of the other features of the Spring Framework.

The spring-test module supports the unit testing and integration testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

What is Spring Boot

--Spring Boot is an open source Java-based framework

-- is built on top of the conventional spring framework, so, it provides all the features of spring and is yet easier to use than spring

-- Spring Boot is a micro service-based framework and making a production-ready application in very less time.

-- In Spring Boot everything is auto-configured. We just need to use proper configuration for utilizing a particular functionality.

-- Spring Boot is very useful if we want to develop REST API

--developed by Pivotal Team

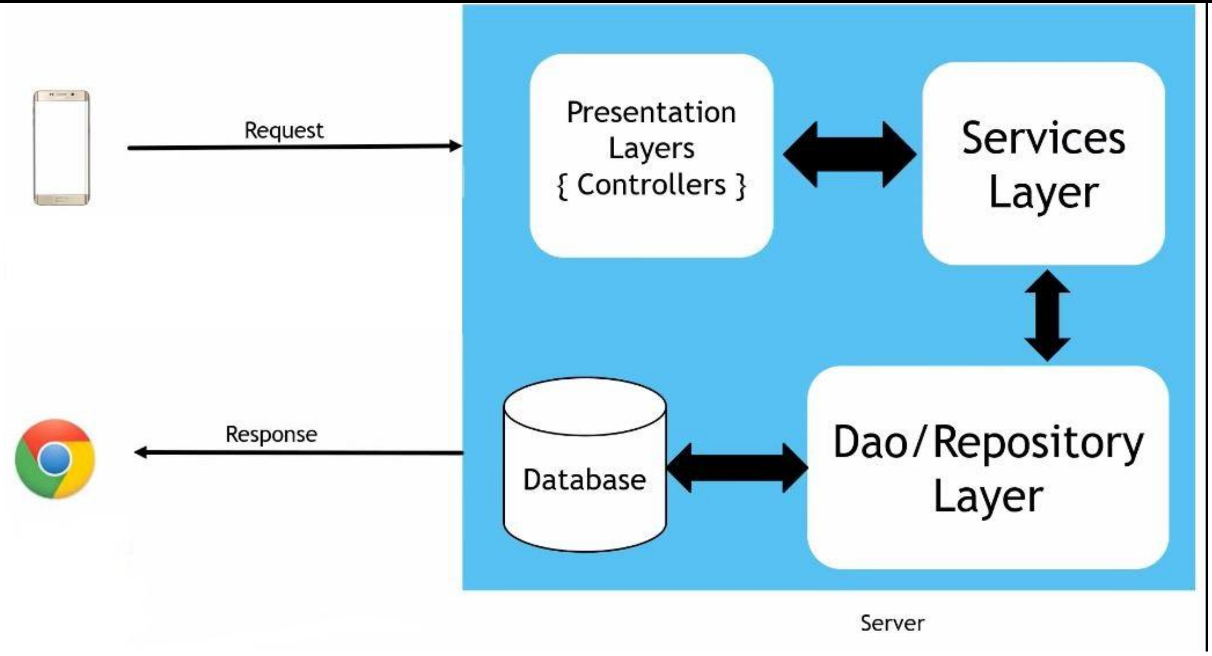
Difference between Spring and Spring Boot :

Spring	Spring Boot
Spring is an open-source lightweight framework widely used to develop enterprise applications.	Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.
The most important feature of the Spring Framework is dependency injection.	The most important feature of the Spring Boot is Auto-configuration.
It helps to create a loosely coupled application.	It helps to create a stand-alone application.

To run the Spring application, we need to set the server explicitly.	Spring Boot provides embedded servers such as Tomcat and Jetty etc.
To run the Spring application, a deployment descriptor is required.	There is no requirement for a deployment descriptor.
To create a Spring application, the developers write lots of code.	It reduces the lines of code.
It doesn't provide support for the in-memory database.	It provides support for the in-memory database such as H2.

Understanding J2EE / Spring Boot Communication Client Server Architecture (Horizontal Slicing)

Client		**Server**				**Database**
Browser - Chrome, Mozilla, Mobile Phone, Laptop, Desktop **Postman**	**----->** Request **<-----** Reply	Presentation layer **Controller uses**	Service Layer	DAO Layer	**----->** Request **<-----** Reply	Repository layer
		To accept Request (What client wants)	**Business Logic** (Classess, Methods)	Database Connectivity		Database Oracle, MySQL



Request ---> Controller --> Controller uses Services from Service layer -->DAO---> Database

----

From hands on perspective...how to send or fire a request.....

Use the software Tool Postman - To fire request

What is POSTMAN Tool?

Postman is an interactive and automatic tool for verifying the APIs of your project. Postman is a Google Chrome app for interacting with HTTP APIs. It presents you with a friendly GUI for constructing requests and reading responses.

Request Method	Response
GET	Get one or multiple records from database
POST	Add one or more records in database
PUT	Update one or more records in database
DELETE	Delete one or more records from database

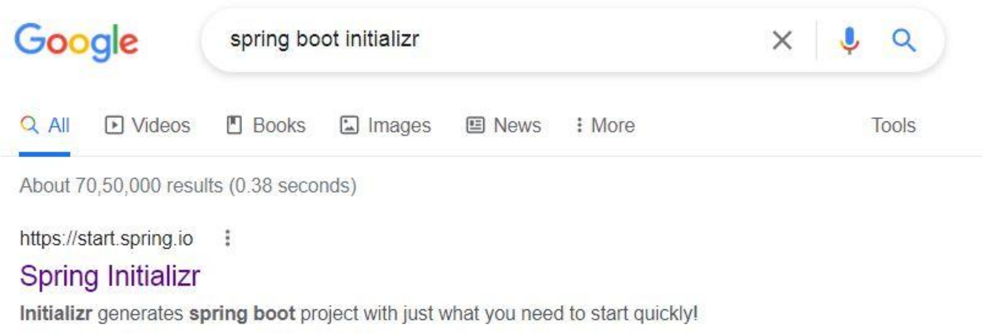
Moving towards writing API for our CV project

Request Method	API URLs	Response
GET	/biodata	Get all biodatas from database
GET	/ biodata /grno	Get only one biodata from database
POST	/ biodata	Add new biodata to database
PUT	/ biodata	Update the biodata in database
DELETE	/ biodata/grno	Delete a biodata from a database

How to start coding?

How to start spring boot project?

Type "Spring boot Initializer" in google search: You will get



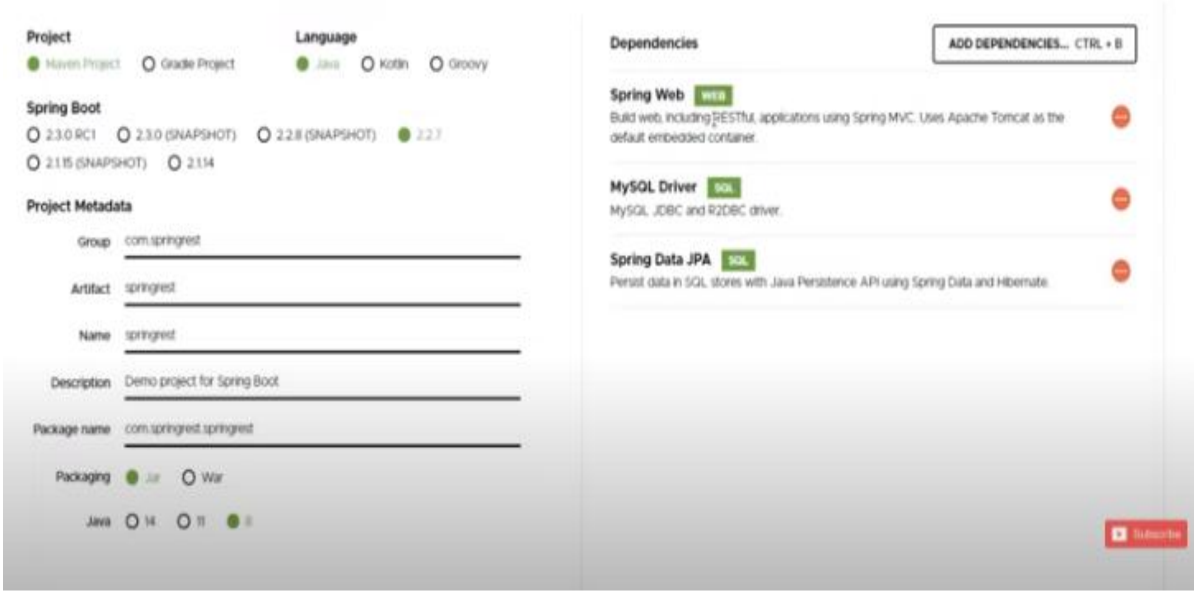
About 70,50,000 results (0.38 seconds)

https://start.spring.io

Spring Initializr

Initializr generates **spring boot** project with just what you need to start quickly!

Next - Click the first link : https://start.spring.io/



Project

● Maven Project ○ Gradle Project

Language

● Java ○ Kotlin ○ Groovy

Spring Boot

○ 2.3.0 RC1 ○ 2.3.0 (SNAPSHOT) ○ 2.2.8 (SNAPSHOT) ● 2.2.7
○ 2.1.15 (SNAPSHOT) ○ 2.1.14

Project Metadata

Group: com.springrest

Artifact: springrest

Name: springrest

Description: Demo project for Spring Boot

Package name: com.springrest.springrest

Packaging: ● Jar ○ War

Java: ○ 14 ○ 11 ● 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

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

Select: Mavan Java and spring boot Version 2.2.7

Project Metadata: comp.springrest

Artifact: springrest ------------This is your project name

Packaging ------ jar

Add Dependencies: Spring Web.......MySQL Driver......Spring Data JPA

Click on: Generate ....it will generate jar file in download folder

---

Extract it

Open it in ID

Import - as a maven project

Tick on ------ pom.xml

It will update

-------Postman --- to send request as well as data

Click on your project

springrest

-- src/main/java ---- consists of all java Classess

within it

com.springrest.springrest

within it

SpringrestApplication.java ----this is our main spring boot application.

------- right click and run as app

----

src/main/resources ---- application properties --- configuration

pom.xml ---- heart file : it consists of all dependencies'- **web, jpa and my-sql**-----

use build path -----to add and remove dependencies

java version--updation --- project-maven---update

Built-in server in Maven / gradle -- Tomcat - 8080

Postman: GET localhost 8080

Let us build controller : Controller for REST API

(REST - Representational State Transfer)

Note...main package is: com.springrest.springrest

Create a new subpackage controller within mainpackage : com.springrest.springrest.controller

and within this package, create a class : MyController

```
@RestController
public class MyController {
   @GetMapping("/home")
   public String home() {
       return "This is my home page home";
   }
```

Annotations: Syntactic Metadata (data about data)

Next we want following operations via controllor:

- get all biodatas

- getbiodata by id

- addbiodata etc

// get the all biodata's

public List<Biodata> getBiodata();

```
public List<Biodata> getBiodata();
```

Create a new subpackage entities within mainpackage

com.springrest.springrest.entities

right click on <Biodata> and create entities class Biodata

public class Biodata{

private long id;

private String title;

private String description;

```
public class Biodata {
    private long id;
    private String fname;
    private String lname;
    private int age;

    }
```

source ---Generate constructor using fields

```
public Biodata(long id, String fname, String lname, int age) {
    super();
    this.id = id;
    this.fname = fname;
    this.lname = lname;
    this.age = age;
}
```

Default constructor from superclass

```
public Biodata() {
    super();
}
```

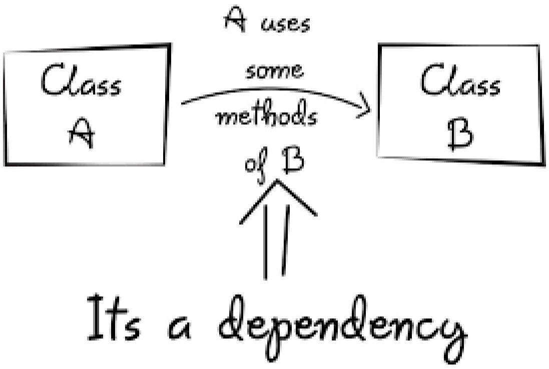
Generate getters and setters

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

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

public String getFname() {
    return fname;
}

public void setFname(String fname) {
    this.fname = fname;
}

public String getLname() {
    return lname;
}

public void setLname(String lname) {
    this.lname = lname;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}
```

Generate toStrings

```
@Override
public String toString() {
    return "Biodata{" +
            "id=" + id +
            ", fname='" + fname + '\'' +
            ", lname='" + lname + '\'' +
            ", age=" + age +
            '}';
}
```

Revisiting Dependency Injection (DJ)



When class A uses some functionality of class B, then its said that class A has a dependency of class B.

In Java, before we can use methods of other classes, we first need to create the object of that class (i.e. class A needs to create an instance of class B).

So, transferring the task of creating the object to someone else and directly using the dependency is called dependency injection.

Why should I use dependency injection?

Let’s say we have a car class which contains various objects such as wheels, engine, etc.

Here the car class is responsible for creating all the dependency objects. Now, what if we decide to ditch MRFWheels in the future and want to use Yokohama Wheels?

We will need to recreate the car object with a new Yokohama dependency. But when using dependency injection (DI), we can change the Wheels at runtime (because dependencies can be injected at runtime rather than at compile time).

You can think of DI as the middleman in our code who does all the work of creating the preferred wheels object and providing it to the Car class.

It makes our Car class independent from creating the objects of Wheels, Battery, etc.

There are basically three types of dependency injection:

constructor injection: the dependencies are provided through a class constructor.

setter injection: the client exposes a setter method that the injector uses to inject the dependency.

interface injection: the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.

So now its the dependency injection’s responsibility to:

Create the objects

Know which classes require those objects

And provide them all those objects

If there is any change in objects, then DI looks into it and it should not concern the class using those objects. This way if the objects change in the future, then its DI's responsibility to provide the appropriate objects to the class.

Benefits of using DI

Helps in Unit testing.

Boiler plate code is reduced, as initializing of dependencies is done by the injector component.

Extending the application becomes easier.

Helps to enable loose coupling, which is important in application programming.

Disadvantages of DI

It's a bit complex to learn, and if overused can lead to management issues and other problems.

Many compile time errors are pushed to run-time.

Dependency injection frameworks are implemented with reflection or dynamic programming. This can hinder use of IDE automation, such as "find references", "show call hierarchy" and safe refactoring.

@GetMapping("/Biodata")

public List<Biodata> getBiodatas()

{ **service**

}

Then create sub package- com.springrest.springrest.services

Creating service---- one more class

First create interface -- BiodataService

```
@GetMapping("/Biodata")
    public List<Biodata> getbiodata()
    {
        return bioservice.getbiodata();
    }
```

@GetMapping("/Biodata")

public List<Biodata> getbioData()

```
  {

      return bioservice.getbiodata();

  }
```

drag your courseservice to services by dragging

```
@Autowired
private BioService bioService;
```

```
// get all biodatas
@GetMapping("/Biodata")
public List<Biodata> getBiodata()
{
    return this.bioService.getBiodata();
}
```

```
public interface BioService {
    public List<Biodata> getBiodata();

    public Biodata getBiodatabyId(int id);

    public Biodata addBiodata(Biodata biodata);
}
```

```
@Service
public class BioServiceImpl implements BioService {

    List<Biodata> list;

    public BioServiceImpl()
    {
        list = new ArrayList<>();
        list.add(new Biodata(1,"Manik","Dhore",55));
        list.add(new Biodata(2,"Anil","Mhaske",53));
    }

    @Override
    public List<Biodata> getBiodata() {
        return list;
    }

    @Override
    public Biodata getBiodatabyId(int id) {
        return null;
    }

    @Override
    public Biodata addBiodata(Biodata biodata) {
        return null;
    }
}
```

EJB - 2000

Java EE features - EJB - messaging and heavy entities

- Difficult to manage

POJOS

Spring Boot- POJI, DI, MVC, REST, Security. Batch, Data, AOP

----- It integrates with Hibernate and Struts

----- For big project....enterprise application ....you need many java files and lots of configuration

----- Need more attention on configuration but we need to focus on coding

Developer
Spring Boot
Spring

Spring -- lot of configuration

--you want develop web application using spring then you need to install server

virtual server...cloud server...Linux......web server------Tomcat

Spring Boot Says ------ I will give you dependencies and configuration

------- all basic configuration s done by spring boot

------- you can add as many as features with little configuration

- application configuration file

------- when we create project it will embed Tomcat....n...you can run on any JVM

----- will give you starter web......starter jdbc......starter auto configuration

pom.xml

------ therefore you can develop production ready application

Advanced Project

- ---Dependency Injection ------JAVA....C#.......PHP

---- using from long time

---- objects------dependendent on other objects

------ Laptop.......low cost.......high cost .....dependent on RAM n Hard drive

----- Laptop is having Hitachi Hard drive....you want to replace it Samsung drive

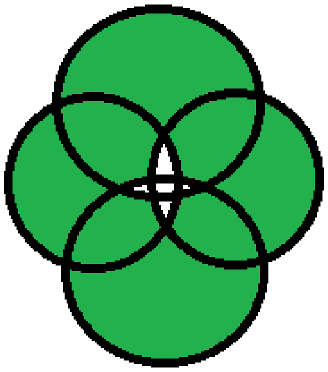
----- we create object by using new....which is tight coupling

------ Software Engineering -- Tight Coupling and loose coupling

----- loose coupling ---one object not totally dependent on other object

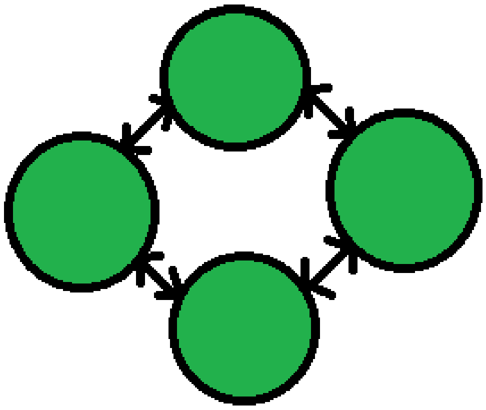
------ hence we create.....abstract class----which is interface

```
public interface Topic
{
    void understand();
}

class Topic1 implements Topic {
public void understand()
    {
        System.out.println("Got it");
    }
}

class Topic2 implements Topic {
public void understand()
    {
        System.out.println("understand");
    }
}

public class Subject {
public static void main(String[] args)
    {
        Topic t = new Topic1();
        t.understand();
    }
```



Tight coupling:

1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:

1. Less Interdependency
2. Less coordination
3. Less information flow

```
class Volume
            {
                 public static void main(String args[])
                 {
                     Box b = new Box(5,5,5);
                     System.out.println(b.getVolume());
                 }
            }
            final class Box
            {
                 private int volume;
                 Box(int length, int width, int height)
                 {
                     this.volume = length * width * height;
                 }
                 public int getVolume()
                 {
                     return volume;

                 }
            }
```

------- Singleton design Pattern

------ DI -- external party does this for you without using new

------ @ Component

------@AutoWired   -----they are getting connected

------ Testing ------ Class and Database......You can create Mock class

------- it is possible because of loose coupling.

Complete Implementation:

```
@RestController
public class MyControler {
    @Autowired
    private BioService bioService;

    @GetMapping("/Biodata")
    public List<Biodata> getBiodata(){
        return bioService.getBiodata();
    }

    @GetMapping("/Biodata/{id}")
    public Biodata getBiodatabyId(@PathVariable int id)
    {
        return this.bioService.getBiodatabyId(id);
    }

    @PostMapping("/Biodata")
    public Biodata addBiodata(@RequestBody Biodata biodata)
    {
        return this.bioService.addBiodata(biodata);
    }
}
```

BioService

```
public interface BioService {
    public List<Biodata> getBiodata();
    public Biodata getBiodatabyId(int id);
    public Biodata addBiodata(Biodata biodata);
}
```

Implementation

```
@Service
public class BioServiceImpl implements BioService {

     public List<Biodata> list;

    public BioServiceImpl() {
        list = new ArrayList<>();
        list.add(new Biodata(1,"Manik","Dhore",55));
        list.add(new Biodata(2,"Anil","Mhaske",53));
    }

    @Override
    public List<Biodata> getBiodata() {
        return list;
    }

    @Override
    public Biodata getBiodatabyId(int id) {
        // TODO Auto-generated method stub
        Biodata biodata = null;
        for (Biodata data : list) {
            if (data.getId() == id) {
                biodata = data;
            }
        }
        return biodata;
    }

    @Override
    public Biodata addBiodata(Biodata biodata) {
        list.add(biodata);
        return biodata;
    }
}
```

Spring Boot Dependency and Application Configuration

Spring Boot Security

----- Very Simple......Just add dependency related with security

----Two ways

---while creating spring boot project

--- or in existing spring boot project

During creating spring boot project add following two dependencies from spring Initializer

add --- Spring Security

and for existing project......add following dependencies in pom.xml file

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

In browser...it will automatically display login window:

Enter login as :

login: user and **password**: generated runtime by spring boot

In Postman:

goto **authorization**: select **basic auth**

Provide credentials: in user and password

Setting user specific login and password:

Goto **application.config file**

```
spring.security.user.name=manik
spring.security.user.password=Tcd22#12dsan
```

-----------Following code is automatically added in your project--------------------

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <meta name="description" content="">
    <meta name="author" content="">
    <title>Please sign in</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css" rel="stylesheet"
        integrity="sha384-/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M" crossorigin="anonymous">
    <link href="https://getbootstrap.com/docs/4.0/examples/signin/signin.css" rel="stylesheet"
        crossorigin="anonymous" />
</head>

<body>
    <div class="container">
        <form class="form-signin" method="post" action="/login">
            <h2 class="form-signin-heading">Please sign in</h2>
            <p>
                <label for="username" class="sr-only">Username</label>
                <input type="text" id="username" name="username" class="form-control" placeholder="Username" required autofocus>
        </p>
                <p>
                    <label for="password" class="sr-only">Password</label>
                    <input type="password" id="password" name="password" class="form-control" placeholder="Password" required>
        </p>
                    <input name="_csrf" type="hidden" value="b227cf5c-081b-4a6b-bc7f-459ded104111" />
                    <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
        </form>
    </div>
</body>

</html>
```

Annotation for HTTP methods:

@GetMapping("/Biodata")

@PostMapping("/Biodata")

@PutMapping("/Biodata")

is actually:

@RequestMapping(path="/Biodata", Method=**RequestMethod.GET**)

@RequestMapping(path="/Biodata", Method=RequestMethod.**POST**)

@RequestMapping(path="/Biodata", Method=RequestMethod.**PUT**)

Understanding Controller:

public Biodata getBiodatabyId(int id)

public Biodata getBiodatabyId**(@PathVariable** int id)   // used for individual attribute

but for

public Biodata addBiodata(Biodata biodata)

public Biodata addBiodata(**@RequestBody** Biodata biodata)   // used for class implementation body

----

Database Connectivity: using JPA

Need one interface: BiodataDao

```
public interface BiodataDao extends {
}
```

Extends it to JPA Repository: will provide all query operations:

```
public interface BiodataDao extends JpaRepository<Biodata, Long> {
}
```

Service Implementation:

```
private BiodataDao biodataDao;
```

Record Insertion: POST Method

```
biodataDao.save(biodata);
```

Record Updation: PUT Method

```
biodataDao.save(biodata);
```

To diplay all records: GET Method

```
biodataDao.findAll();
```

```
# Maria DB database configuration
spring.datasource.url=jdbc:mariadb://localhost:3306/studentscvs
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MariaDB103Dialect
```

Understanding Spring Boot

Build Systems:

It is strongly recommended that you choose a build system that supports ***dependency management*** and that can consume artifacts published to the “Maven Central” repository. We would recommend that you choose Maven or Gradle.

What is gradle?

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle. It **allows you to package executable jar or war archives, run Spring Boot applications**, and use the **dependency management** provided by spring-boot-dependencies . Spring Boot's Gradle plugin requires Gradle 6.8, 6.9, or 7

What is Maven?

The Spring Boot Maven Plugin provides Spring Boot support in Apache Maven. It allows you to package executable jar or war archives, run Spring Boot applications, generate build information and start your Spring Boot application prior to running integration

What is build in spring boot?

Both the Maven plugin and the Gradle plugin allow generating build information containing the coordinates, name, and version of the project. The plugins can also be configured to add additional properties through configuration. When such a file is present, Spring Boot auto-configures a Build Properties bean.

Spring Boot team provides a **list of dependencies** to support the Spring Boot version for its every release. You do not need to provide a version for dependencies in the build configuration file. Spring Boot automatically configures the dependencies version based on the release.

Example of build Systems:

Starters

Starters are a set of convenient dependency descriptors that you can include in your application. You get a one-stop shop for all the Spring and related technologies that you need without having to hunt through sample code and copy-paste loads of dependency descriptors. For example, if you want to get started using Spring and JPA for database access, include the spring-boot-starter-data-jpa dependency in your project.

The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

What is in a name?

All official starters follow a similar naming pattern;

spring-boot-starter-*,

where * is a particular type of application.

Application starters are provided by Spring Boot

under the

org.springframework.boot group:

Spring Boot application starters	
Name	**Description**
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML
spring-boot-starter-activemq	Starter for JMS messaging using Apache ActiveMQ
spring-boot-starter-amqp	Starter for using Spring AMQP and Rabbit MQ
spring-boot-starter-aop	Starter for aspect-oriented programming with Spring AOP and

Spring Boot application starters	
Name	**Description**
	AspectJ
spring-boot-starter-artemis	Starter for JMS messaging using Apache Artemis
spring-boot-starter-batch	Starter for using Spring Batch
spring-boot-starter-cache	Starter for using Spring Framework’s caching support
spring-boot-starter-data-cassandra	Starter for using Cassandra distributed database and Spring Data Cassandra
spring-boot-starter-data-cassandra-reactive	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive
spring-boot-starter-data-couchbase	Starter for using Couchbase document-oriented database and Spring Data Couchbase
spring-boot-starter-data-couchbase-reactive	Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive
spring-boot-starter-data-elasticsearch	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch
spring-boot-starter-data-jdbc	Starter for using Spring Data JDBC
spring-boot-starter-data-jpa	Starter for using Spring Data JPA with Hibernate
spring-boot-starter-data-ldap	Starter for using Spring Data LDAP
spring-boot-starter-data-mongodb	Starter for using MongoDB document-oriented database and Spring Data MongoDB
spring-boot-starter-data-mongodb-reactive	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive
spring-boot-starter-data-neo4j	Starter for using Neo4j graph database and Spring Data Neo4j
spring-boot-starter-data-r2dbc	Starter for using Spring Data R2DBC
spring-boot-starter-data-redis	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client
spring-boot-starter-data-redis-reactive	Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client
spring-boot-starter-data-rest	Starter for exposing Spring Data repositories over REST using Spring Data REST
spring-boot-starter-freemarker	Starter for building MVC web applications using FreeMarker views

Spring Boot application starters	
Name	**Description**
spring-boot-starter-groovy-templates	Starter for building MVC web applications using Groovy Templates views
spring-boot-starter-hateoas	Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS
spring-boot-starter-integration	Starter for using Spring Integration
spring-boot-starter-jdbc	Starter for using JDBC with the HikariCP connection pool
spring-boot-starter-jersey	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web
spring-boot-starter-jooq	Starter for using jOOQ to access SQL databases with JDBC. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc
spring-boot-starter-json	Starter for reading and writing json
spring-boot-starter-jta-atomikos	Starter for JTA transactions using Atomikos
spring-boot-starter-mail	Starter for using Java Mail and Spring Framework’s email sending support
spring-boot-starter-mustache	Starter for building web applications using Mustache views
spring-boot-starter-oauth2-client	Starter for using Spring Security’s OAuth2/OpenID Connect client features
spring-boot-starter-oauth2-resource-server	Starter for using Spring Security’s OAuth2 resource server features
spring-boot-starter-quartz	Starter for using the Quartz scheduler
spring-boot-starter-rsocket	Starter for building RSocket clients and servers
spring-boot-starter-security	Starter for using Spring Security
spring-boot-starter-test	Starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito
spring-boot-starter-thymeleaf	Starter for building MVC web applications using Thymeleaf views
spring-boot-starter-validation	Starter for using Java Bean Validation with Hibernate Validator
spring-boot-starter-web	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
spring-boot-starter-web-services	Starter for using Spring Web Services

Spring Boot application starters	
Name	**Description**
spring-boot-starter-webflux	Starter for building WebFlux applications using Spring Framework’s Reactive Web support
spring-boot-starter-websocket	Starter for building WebSocket applications using Spring Framework’s WebSocket support

Spring Boot technical starters	
Name	**Description**
spring-boot-starter-jetty	Starter for using Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat
spring-boot-starter-log4j2	Starter for using Log4j2 for logging. An alternative to spring-boot-starter-logging
spring-boot-starter-logging	Starter for logging using Logback. Default logging starter
spring-boot-starter-reactor-netty	Starter for using Reactor Netty as the embedded reactive HTTP server.
spring-boot-starter-tomcat	Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web
spring-boot-starter-undertow	Starter for using Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat

In addition to the application starters, the following starters can be used to add *production ready* features:

Spring Boot production starters	
Name	**Description**
spring-boot-starter-actuator	Starter for using Spring Boot’s Actuator which provides production ready features to help you monitor and manage your application

Locating the Main Application Class:

The @SpringBootApplication annotation is often placed on your main class, and it implicitly defines a base “search package” for certain items. For example, if you are writing a JPA application, the package of the @SpringBootApplication annotated class is used to search for @Entity items. Using a root package also allows component scan to apply only on your project.

If you do not want to use @SpringBootApplication,

the **@EnableAutoConfiguration** and

@ComponentScan annotations

that it imports defines that behavior so you can also use those instead.

What is the difference between @configuration and @component in Spring?

The main difference between these annotations is that
@ComponentScan scans for Spring components
while
@EnableAutoConfiguration
is used for auto-configuring beans present in the classpath in Spring Boot applications.

The MyApplication.java file would declare the main method, along with the basic @SpringBootApplication, as follows:

```
@SpringBootApplication
public class MyApplication {

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

Typical Layout of Spring Boot Application:

```
com
 +- example
     +- myapplication
         +- MyApplication.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

Configuration Classes:

Spring Boot favors Java-based configuration. Although it is possible to use SpringApplication with XML sources, we generally recommend that your primary source be a single @Configuration class. Usually the class that defines the main method is a good candidate as the primary @Configuration.

Spring Boot lets you **externalize your configuration** so that you can work with the same application code in different environments. You can use properties files, YAML files, environment variables, and command-line arguments to externalize configuration.

Importing Additional Configuration Classes and XML configurations :

@Configuration

@Import

and

@Configuration

@ImportResource

Disabling Specific Auto-configuration Classes

```
@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
public class MyApplication {

}
```

Spring Beans and Dependency Injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies.

We generally recommend using constructor injection to wire up dependencies and @ComponentScan to find beans.

If you structure your code as suggested above (locating your application class in a top package), you can add

@ComponentScan without any arguments or use the @SpringBootApplication annotation which implicitly includes it.

All of your application components

@Component,

@ Entity

@Service,

@Repository,

@Controller

and others are automatically registered as Spring Beans.

@Service Bean uses constructor injection to obtain a required bean:

If a bean has more than one constructor, you will need to mark the one you want Spring to use with

@Autowired

Using the @SpringBootApplication Annotation

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single @SpringBootApplication annotation can be used to enable those three features, that is:

- @EnableAutoConfiguration: enable Spring Boot’s auto-configuration mechanism
-
- @ComponentScan: enable @Component scan on the package where the application is located
-
- @SpringBootConfiguration: enable registration of extra beans in the context or the import of additional configuration classes. An alternative to Spring’s standard @Configuration that aids configuration detection in your integration tests.

```
@SpringBootApplication // same as @SpringBootConfiguration @EnableAutoConfiguration
                    // @ComponentScan
public class MyApplication {

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

None of these features are mandatory and you may choose to replace this single annotation by any of the features that it enables. For instance, you may not want to use component scan or configuration properties scan in your application:

```
@SpringBootConfiguration(proxyBeanMethods = false)
@EnableAutoConfiguration
```

```
@Import({ SomeConfiguration.class, AnotherConfiguration.class })
public class MyApplication {

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

Developer Tools

Spring Boot includes an additional set of tools that can make the application development experience a little more pleasant. The spring-boot-devtools module can be included in any project to provide additional development-time features. To include devtools support, add the module dependency to your build, as shown in the following listings for Maven and Gradle:

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

spring.devtools.restart.enabled

```
-Dspring.devtools.restart.enabled=true

-Dspring.devtools.restart.enabled=false
```

What is use of Devtools in spring boot?

Spring-boot-Devtools module includes an embedded Live Reload server that is used **to trigger a browser refresh when a resource is changed**.

How do I enable dev tools in spring boot?

To enable dev tools in spring boot application is very easy. **Just add the spring-boot-devtools dependency in your build file**.

Property Defaults:

Several of the libraries supported by Spring Boot use caches to improve performance.

For example, template engines cache compiled templates to avoid repeatedly parsing template files.

Also, Spring MVC can add HTTP caching headers to responses when serving static resources.

While caching is very beneficial in production, it can be counter-productive during development, preventing you from seeing the changes you just made in your application. For this reason, spring-boot-devtools disables the caching options by default.

Cache options are usually configured by settings in your application.properties file.

For example, Thymeleaf offers the spring.thymeleaf.cache property.

Rather than needing to set these properties manually, the spring-boot-devtools module automatically applies sensible development-time configuration.

Restart vs Reload

The restart technology provided by Spring Boot works by using two classloaders. :

Classes that do not change (for example, those from third-party jars) are loaded into a *base* classloader.

Classes that you are actively developing are loaded into a *restart* classloader.

When the application is restarted, the *restart* classloader is thrown away and a new one is created.

This approach means that application restarts are typically much faster than “cold starts”, since the *base* classloader is already available and populated.

Global Settings

You can configure global devtools settings by adding any of the following files to the $HOME/.config/spring-boot directory:

1. spring-boot-devtools.properties
2. spring-boot-devtools.yaml
3. spring-boot-devtools.yml

Any properties added to these files apply to *all* Spring Boot applications on your machine that use devtools.

Remote Applications

The Spring Boot developer tools are not limited to local development. You can also use several features when running applications remotely. Remote support is opt-in as enabling it can be a security risk. It should only be enabled when running on a trusted network or when secured with SSL. If neither of these options is available to you, you should not use DevTools' remote support. You should never enable support on a production deployment.

To enable it, you need to make sure that devtools is included in the repackaged archive, as shown in the following listing:

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

Running the Remote Client Application

The remote client application is designed to be run from within your IDE. You need to run

```
org.springframework.boot.devtools.RemoteSpringApplication
```

with the same classpath as the remote project that you connect to. The application's single required argument is the remote URL to which it connects.