<URL:-> <https://www.youtube.com/watch?v=m559BxR30ls>

**Q) Principle of REST API?**

--client-server architecture

--stateless

--can be cached

**Q) REST is stateless?**

statelessness refers to a communication method in which the server completes every client request independently of all previous requests. Clients can request resources in any order, and every request is stateless or isolated from other requests.

Stateless Protocol does not require the server to retain the server information or session details. Stateful Protocol require server to save the status and session information.

**Q) http and https?**

HTTPS uses TLS (SSL) to encrypt normal HTTP requests and responses, and to digitally sign those requests and responses. As a result, HTTPS is far more secure than HTTP.

**Q) Classification of status code?**

-- 1xx (Informational)

-- 2xx(successful)

-- 3xx (Redirection)

-- 4xx (client error)

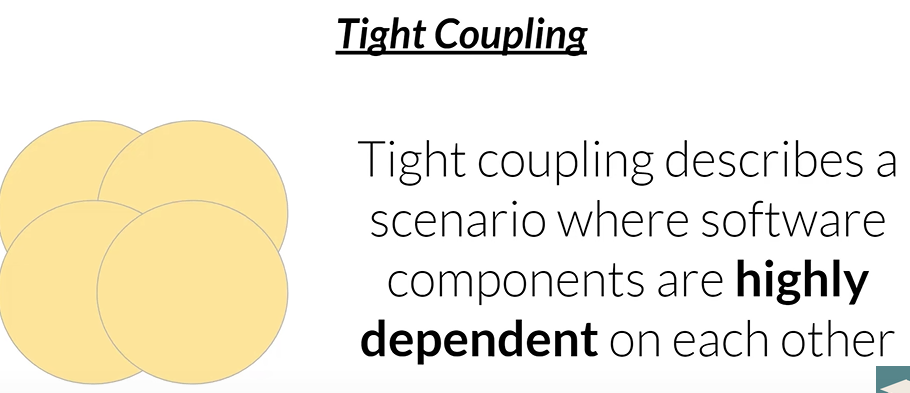
--5xx (server error)

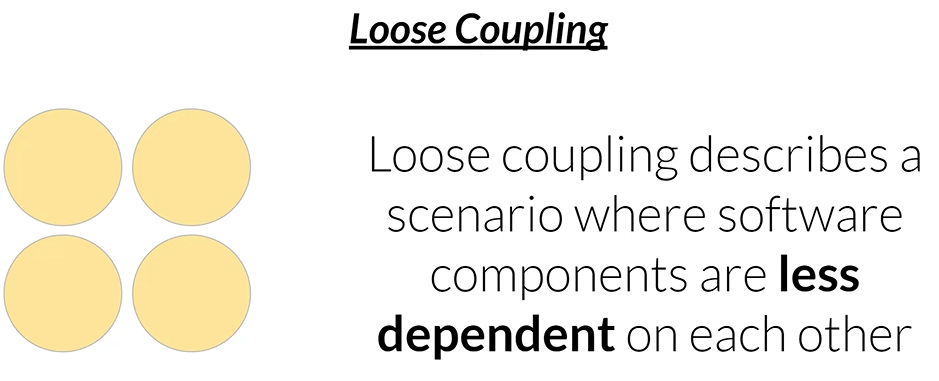
200 ok, 201 created, 204 no content, 301 move permanently, 400 bad request,401 unauthorized, 403 forbidden, 404 Not found,500 Internal server error

**Q) Intro to Spring Framework**

**Q) Intro to Tight Coupling and Loose**

In a tightly coupled application, two or more components that interact with one another are dependent and must be present and functioning at the same time. However, in a loosely coupled application, the components can function independently.

****

****

1. **Tight coupling example below: -**

Create package: com.tight.coupling

Create 3 classes: UserDatabase.java, UserManager.java and TightCouplingExample.java

**UserDatabase.java code:**

package com.tight.coupling

public class UserDatabase {

public String getUserDetails(){

Return “User details from database”;

}

}

**UserManager.java code:**

package com.tight.coupling

public class UserManager {

private UserDatabase userDatabase = new UserDatabase();

public String getUserInfo(){

Return userDatabase.getUserDetails();

}

}

**TightCouplingExample.java code (Main class):**

package com.tight.coupling

public class TightCouplingExample {

private UserManager userManager = new UserManager();

public static void main(String[] args){

Return userManager.getUserInfo();

}

}

1. **Loose coupling example below: -**

Create package: com.loose.coupling

Create 3 classes: UserDatabaseProvider.java, UserManager.java and LooseCouplingExample.java

Create Interface: UserDataProvider.java

**UserDataProvider.java interface code: -**

package com.loose.coupling

public inteface UserDataProvider {

String getUserDetails(); // it give the contract to implement this method

}

}

**UserDatabaseProvider.java code: -**

package com.loose.coupling

public class UserDatabaseProvider implements UserDataProvider {

@Override

public String getUserDetails(){

Return “User details from database”;

}

}

**UserManager.java code:**

package com.loose.coupling

public class UserManager {

private UserDataProvider userDataProvider;

//constructor

UserManager(UserDataProvider userDataProvider){

this.userDataProvider = userDataProvider;

}

public String getUserInfo(){

Return userDataProvider.getUserDetails();

}

}

**LooseCouplingExample.java code (Main class):**

package com.loose.coupling

public class LooseCouplingExample {

UserDataProvide databaseProvider = new UserDatabaseProvider();

UserManager userManagerWithDB = new UserManager(databaseProvider);

System.out.println(userManagerWithDB.getUserInfo());

//If I create on webserice, due to database change

UserDataProvide webServiceProvider = new WebServiceDataProvider();

UserManager userManagerWithWS = new UserManager(webServiceProvider);

System.out.println(userManagerWithWS.getUserInfo());

}

**Note: - If I create another web service due to database change**

WebServiceDataProvider**.java code: -**

package com.loose.coupling

public class WebServiceDataProvider implements UserDataProvider {

@Override

public String getUserDetails(){

Return “Fetch data from WebService”;

}

}

**Q) what is Inversion of control(IOC)?**

In practical terms, this often means that the control over the creation and management of objects, their dependencies, and their lifecycle is handled by an IoC container or framework. This approach helps in decoupling components and enhancing the modularity and testability of the application.

There are several key concepts associated with IoC:

1. **Dependency Injection (DI)**: This is a common implementation of IoC where dependencies are provided to a component from an external source rather than the component creating them itself. This can be done through constructor injection, setter injection, or interface injection.
2. **Service Locator Pattern**: Another way to achieve IoC is through a service locator, which is a registry where components can request their dependencies.
3. **Event-based or Callback Systems**: Some systems use events or callbacks to invert control, where components register to listen for certain events or actions and respond accordingly.

**Q) What is maven repository?**

A Maven Repository is **a location, generally on a filesystem (either remote or local), where maven artifacts are stored and managed**. Once artifacts have been stored in a maven repository, they are available for retrieval and inclusion in other maven projects.

**Q) What is Beans?**

Objects that are managed by frameworks are known as Beans.

**Q) Core Concepts of Spring**

**Q) Spring Container and Configuration**

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 life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application.

**Types of Spring container:**

1. **ApplicationContext**
2. **BeanFactory**

**Q) Setting Up a Spring Project**

**Q) Creating Your First Bean**

When you create a new project, you will find a file called pom.xml. In this file we need to provide the dependency of basic spring jars. Copy the below code and paste it in your pom.xml inside <dependencies> tag. You can choose to provide the latest stable version of your own choice.

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 <version>4.1.4.RELEASE</version>  
</dependency>

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>4.1.4.RELEASE</version>  
</dependency>

Once you save the pom.xml file, all required jars will get downloaded automatically.

**Step 1: - create class “car.example.bean.MyBean” , it will create class(MyBean) under the package(com.example.bean)**

package car.example.bean;

public class MyBean {

private String message;

public void setMessage(String message){

this.message = message;

}

public void showMessage(){

System.ou.println(“Message: “+ message);

}

@Override

Public String toString(){

Return “MyBean{“ +

“Message=’” + message +’\’’ +

‘}’;

}

}

**Step 2: - create a xml file inside resources folder “applicationBeanContext.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.xsd">

<!-- bean definitions here -->

<bean id=”myBean“ class=”car.example.bean.MyBean“>

<property name=”message” value=”I am a first bean” />

</bean>

</beans>

**Step 3: - create a class (App.java) under the package(com.example.bean)**

package car.example.bean;

public class App {

public static void main (String[] args){

ApplicationContext context =

new ClassPathXmlApplicationContext(**applicationBeanContext.xml);**

MyBean myBean = (MyBean) context.getBean(“myBean”);

System.out.println(myBean);

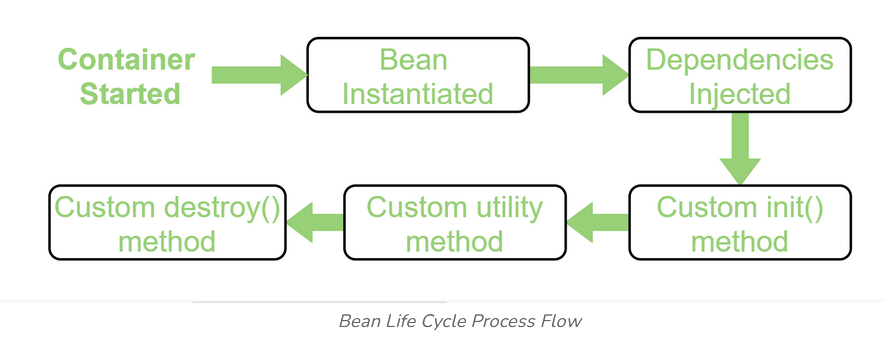
}

}

**Note: - If we run it output: MyBean{Message=’null’} OR MyBean{Message=’I am a first bean ‘}**

**Q) Lifecycle of Bean?**

When we talk about the life cycle of a living thing as a general term, It means when & how it is born, how & what it has done throughout its life, and when & how it has died. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed.



**What are Spring Bean Life Cycle Methods?**

When a Spring Container creates a bean, it provides two methods to every bean by default. These are:

1) public void init() 2) public void destroy()

However, we can change the names of methods, but method signature must be the same. If we remain the method names as it is, the code becomes easy to understand.

**When are the Bean Life Cycle Methods called?**

The init() method is called after bean construction and before requesting an object. destroy() method is called just before the destruction of a bean.

**What are the purpose of implementing Life Cycle Methods?**

init(): If we want to initialize anything such as loading some configurations, creating database connections, we can write that code in init() method.

destroy(): If we want to cleanup something such as closing database connections, we can write that code in destroy() method.

**Q) What are the different approaches to configure Spring Bean Life Cycle Methods?**

There are three approaches to configure Spring Bean Life Cycle methods:

1) Using XML (also called Declarative Approach)

2) Using Spring Interfaces (also called Programmatic Approach)

3) Using Annotations

**Q) Dependency Injection (DI)**

Dependency Injection (DI) is a design pattern used in software development to achieve loose coupling between classes by removing the direct dependency instantiating from the dependent class itself.

Types of DI: -

1. **Constructor Injection: -**
2. Dependencies are provided to the dependent class through its constructor.
3. Dependencies are passed as arguments to the constructor when the dependent class is intantiated.
4. Constructor injection ensures that the dependencies are available when the object is created.
5. **Setter Injection: -**

i) Setter injection involves providing dependencies using setter methods. Dependencies are declared as private instance variables and corresponding setter methods are defined. These setters are used to inject the dependencies into the class.

**Q) Introduction to Autowiring and its types**

@Autowired is an annotation in Spring Framework that **enables dependency injection for Java classes**. It allows Spring to automatically inject dependencies into the class, eliminating the need for manual configuration. This annotation can be used to inject dependencies into fields, methods, and constructors.

**Q) Autowiring by Name**

@Component

public class Bar {

@Autowired

@Qualifier("fooBeanName") // Explicit name-based injection

private Foo foo;

}

**Q) Autowiring by Type**

@Component

public class Bar {

@Autowired

private Foo foo; // Autowiring by type

}

**Q) Autowiring by Constructor**

public car (Specification specification) {

this. Specification = specification;

}

**Q) Introduction to Annotations**

Spring Boot Annotations is **a form of metadata that provides data about a program**. In other words, annotations are used to provide supplemental information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate.

**@Service,@Controller,@RequestMapping,@SpringBootApplication,@Autowired**

**Q) Understanding Components and ComponentScan**

 @**Component**: Used to mark a class as a Spring bean, which can then be injected into other parts of the application. component refers to a java class that is managed by the spring IoC container.

import org.springframework.stereotype.Component;

@Component //Marks the class as a spring component

public class MyService {

public void doSomething() {

// Business logic here

}

}

 @**ComponentScan**: Used to define which packages Spring should scan for components to automatically register as beans.

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = "com.example.myapp")

public class AppConfig {

// Configuration code here

}

In this example, @**ComponentScan** is used to tell Spring to scan the **com.example.myapp** package and its sub-packages for components. If you don't specify a base package, Spring will scan the package of the class where @**ComponentScan** is used.

**Q) @Value Annotation**

This annotation can be used for injecting values into fields in Spring-managed beans, and it can be applied at the field or constructor/method parameter level.

@Value ("string value")

private String stringValue;

@Value ("#{4000}")

private double salary;

@Value("${java.home}")

private String valueFromFile;

@Component

public class MovieRecommender {

private final Map<String, Integer> countPerCatalog;

public MovieRecommender(

@Value("#{{'Thriller': 100, 'Comedy': 300}}") Map<String, Integer> countPerCatalog) {

this.countPerCatalog = countPerCatalog;

}

}

**Q) Transition from XML to Annotations in Spring**

**Q) @Autowired Annotation**

Used for **automatic dependency injection**. In simpler terms, it allows Spring to automatically wire the required beans (dependencies) into your classes, eliminating the need for manual configuration.

The *@Autowired* annotation is used to inject the bean automatically.

The *@Autowired* annotation is used in [Constructor injection](https://www.javaguides.net/2023/01/spring-boot-constructor-injection.html), [Setter injection](https://www.javaguides.net/2023/01/spring-boot-setter-injection-example.html), and [Field injection](https://www.javaguides.net/2023/01/spring-boot-field-injection-example.html).

**Q) @Qualifier Annotation**

There may be a situation when we create more than one bean of the same type and want to wire only one of them with a property. In such cases, we can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired. Following is an example to show the use of @Qualifier annotation.

Let's take a Message Processing Example - a message can be sent in many ways like Email, SMS, Twitter, etc.  
Let's create a *MessageService* interface for multiple message service implementations - *EmailService*, *SMSService*, and *TwitterService* classes.

## **MessageService interface**

public interface MessageService {

public void sendMsg(String message);

}

Next, let's create implementations - *EmailService*, *SMSService,* and *TwitterService* classes.

## **EmailService Class**

public class EmailService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

## **SMSService Class**

public class TwitterService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

## **TwitterService Class**

public class SMSService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

## **MessageProcessor Interface and MessageProcessorImpl Class**

It's time to see the usage of @Qualifier annotation.

public interface MessageProcessor {

public void processMsg(String message);

}

public class MessageProcessorImpl implements MessageProcessor {

private MessageService messageService;

// setter based DI

@Autowired

@Qualifier("twitterService")

public void setMessageService(MessageService messageService) {

this.messageService = messageService;

}

// constructor based DI

@Autowired

public MessageProcessorImpl(@Qualifier("twitterService") MessageService messageService) {

this.messageService = messageService;

}

public void processMsg(String message) {

messageService.sendMsg(message);

}

}

In the above example, Dependency is injected by both setter and constructor so you can use either one of them.

We have used *@Qualifier* to inject *TwitterService* bean using constructor injection:

// setter based DI

@Autowired

@Qualifier("twitterService")

public void setMessageService(MessageService messageService) {

this.messageService = messageService;

}

We have used *@Qualifier* to inject the *TwitterService* bean using setter injection:

// constructor based DI

@Autowired

public MessageProcessorImpl(@Qualifier("twitterService") MessageService messageService) {

this.messageService = messageService;

}

If you want to inject EmailService bean instead of TwitterService bean then you can simply pass bean EmailService bean name. For example:

// constructor based DI

@Autowired

public MessageProcessorImpl(@Qualifier("emailService") MessageService messageService) {

this.messageService = messageService;

}

## **AppConfiguration**

Let's write the java based configuration.

@Configuration

@ComponentScan("com.javadevsguide.springframework.di")

public class AppConfiguration {

@Bean(name="emailService")

public MessageService emailService(){

return new EmailService();

}

@Bean(name="twitterService")

public MessageService twitterService(){

return new TwitterService();

}

@Bean(name="smsService")

public MessageService smsService(){

return new SMSService();

}

@Bean

public MessageProcessor messageProcessor(){

return new MessageProcessorImpl(twitterService());

}

}

## **Testing**

Let's test the example using the Spring IOC container which is an **ApplicationContext** object.

public class TestApplication {

public static void main(String[] args) {

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfiguration.class);

MessageProcessor processor = applicationContext.getBean(MessageProcessor.class);

processor.processMsg("twitter message sending ");

}

}

Output:

twitter message sending

**Q) Introduction to Spring Boot**

Open-source, java-based framework used to create stand-alone, production-grade spring-based application.

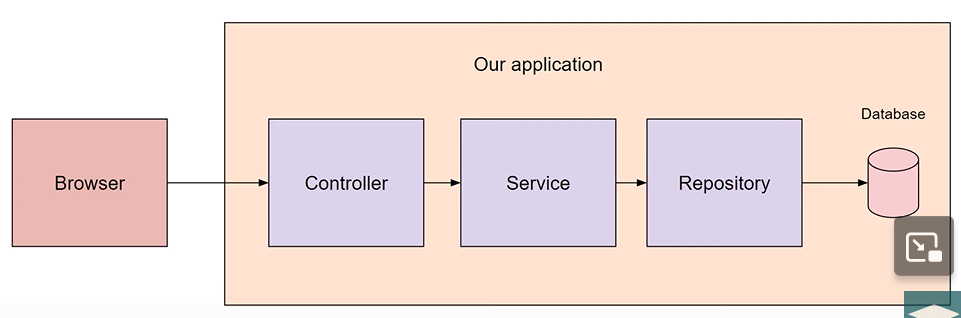
1. **Spring boot = Spring framework + Prebuilt configuration + Embedded server**
2. **Components of Spring boot: -**
3. Springboot starter
4. Auto configuration
5. Spring boot actuator
6. Embedded server
7. Spring boot DevTools
8. **Advantage of spring boot: -**
9. starter code
10. production ready
11. Less time and faster development
12. Less boilercode

**Q) How does Spring Boot work? – Architecture**

**Presentation Layer:** Handles the user interface and user interactions, presenting information to users and capturing user input.

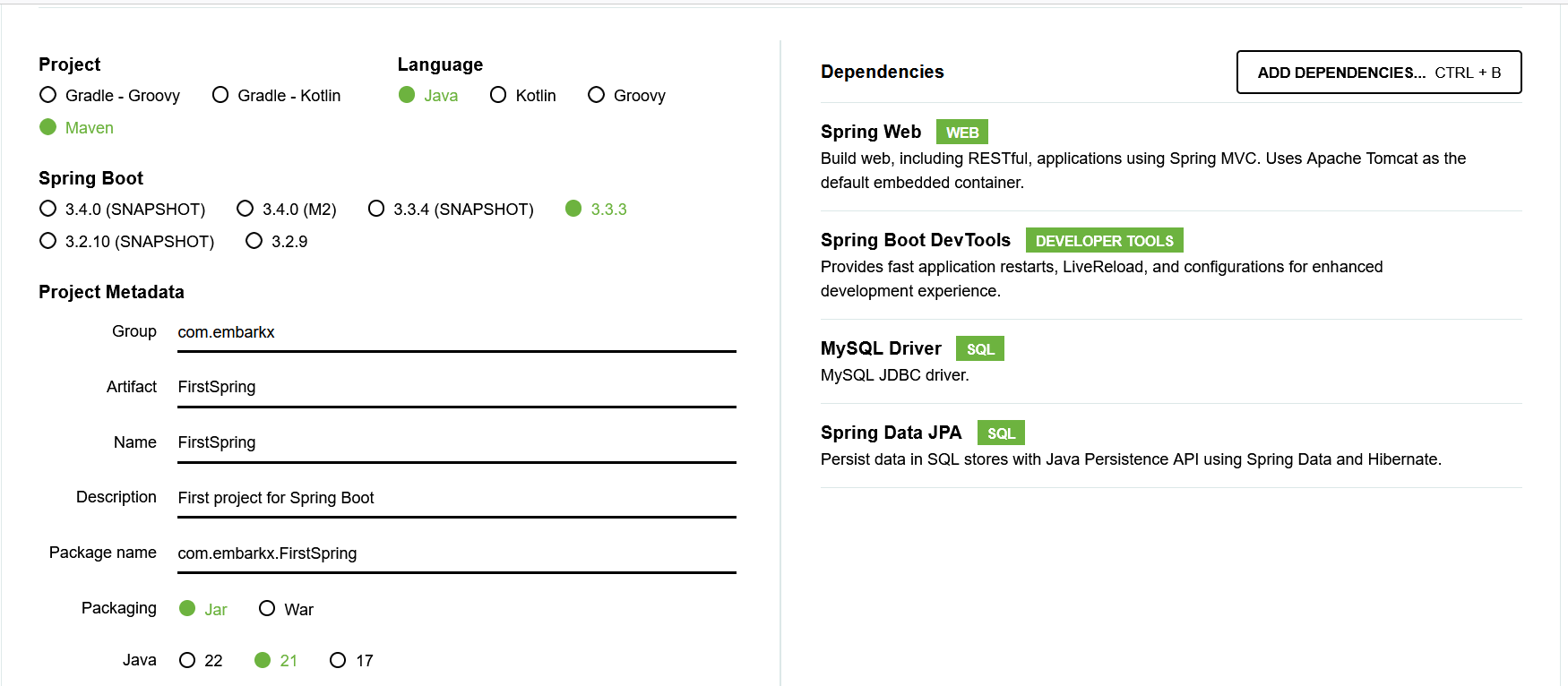
**Service Layer:** Implements the core functionality and rules that drive the business processes and operations.

**Data access layer:** Handles the storage & retrieval of data from databases.



**Q) Getting Started with Spring Initializer**

[**https://start.spring.io/**](https://start.spring.io/) **=> to create project or skeleton of project**

****

**Q) Maven Repository: -**

[**https://central.sonatype.com**](https://central.sonatype.com)

**Q) Designing our First Hello World API**

@RestController

**public** **class** HelloController {

@GetMapping("/hello")

**public** String hello() {

**return** "Hello world";

}

@PostMapping("/hello")

**public** String helloPost(@RequestBody String name) {

**return** "Hello "+ name +"!";

}

}

**Q) JSON Response First Hello World API**

**Step 1: - create class with name “HelloResponse”**

**package** com.embarkx.FirstSpring;

**public** **class** HelloResponse {

**private** String message;

**public** HelloResponse(String message) {

**super**();

**this**.message = message;

}

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

**Step 2: - write below code in rest controller**

@RestController

**public** **class** HelloController {

@GetMapping("/hello")

**public** HelloResponse hello() {

**return** **new** HelloResponse("Hello world");

}

@PostMapping("/hello")

**public** HelloResponse helloPost(@RequestBody String name) {

**return** **new** HelloResponse("Hello "+ name +"!");

}

}

**Output: -**

{

     "message": "Hello world"

}

**Q) Making use of @Pathvariable**

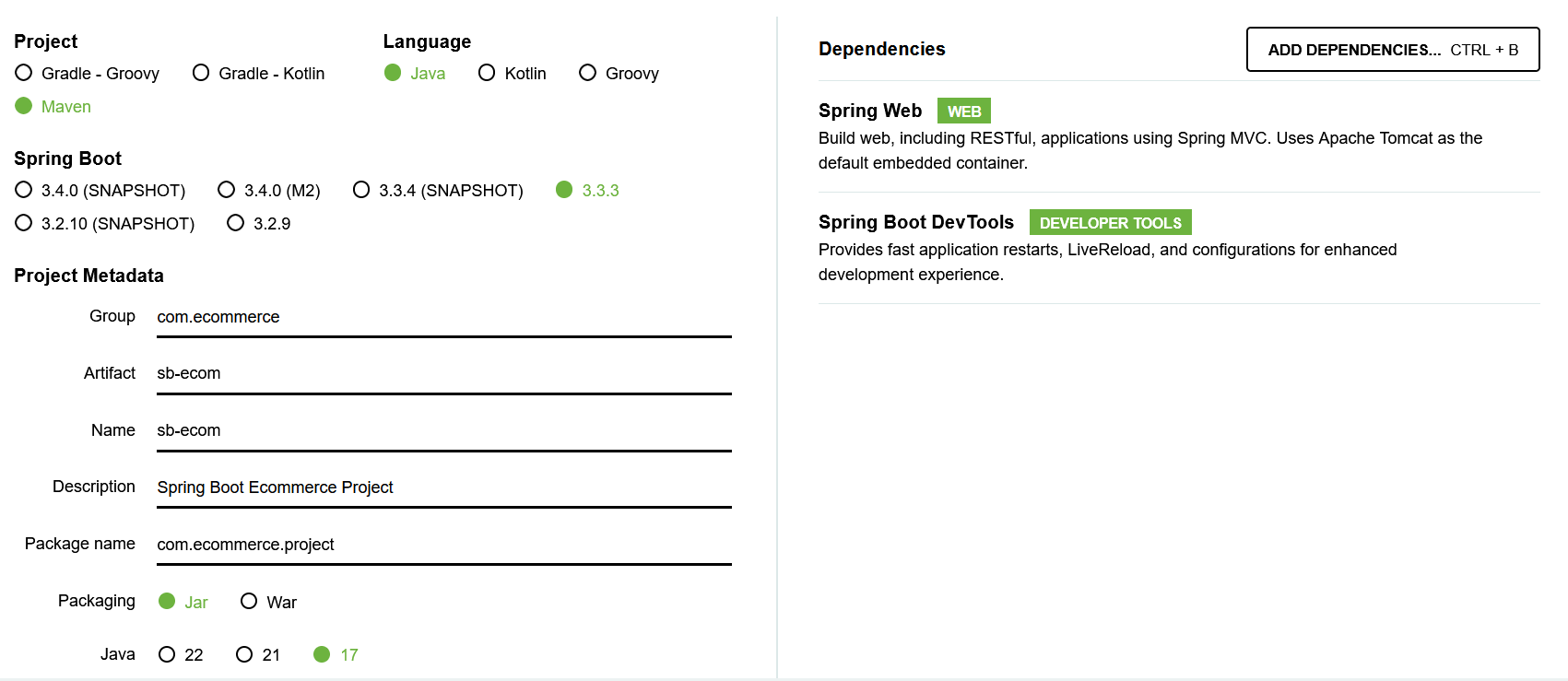
@GetMapping("/hello")

**public** HelloResponse hello() {

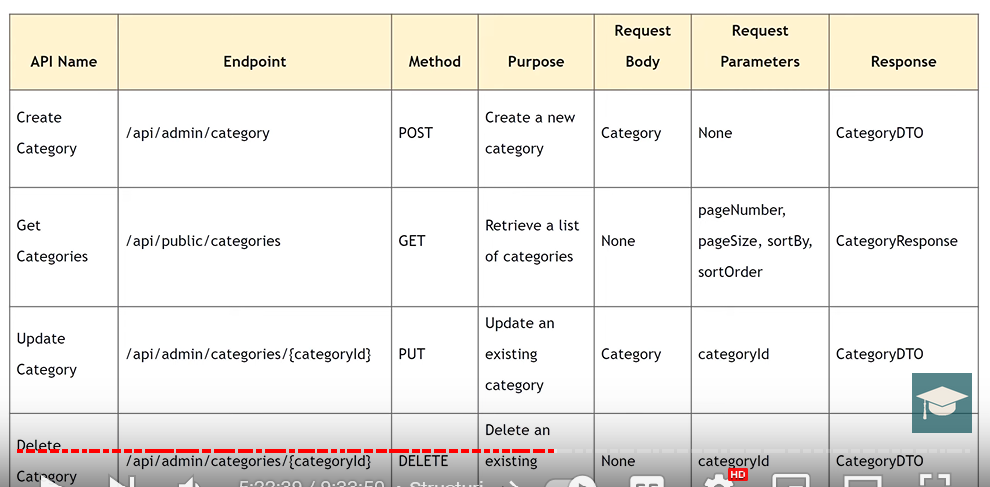
**return** **new** HelloResponse("Hello world");

}

**Q) Setting up Ecommerce Project**

****

**Q) Structuring Thoughts**

****

**Step 1: - create controller**

Create package name “controller” => com.ecommerce.project.controller

Then create a class “CategoryController” under controller package

**CategoryController.java code: -**

package com.ecommerce.project.controller;

import java.util.ArrayList;

import java.util.List;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.DeleteMapping;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.PutMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.server.ResponseStatusException;

import com.ecommerce.project.model.Category;

import com.ecommerce.project.service.CategoryService;

@RestController

@RequestMapping("/api")

public class CategoryController {

private CategoryService categoryService;

public CategoryController(CategoryService categoryService) {

this.categoryService = categoryService;

}

@GetMapping("/public/categories")

public ResponseEntity<List<Category>> getAllCategories(){

List<Category> categories = categoryService.getAllCategories();

return new ResponseEntity<>(categories, HttpStatus.OK);

}

@PostMapping("/admin/category")

public ResponseEntity<String> createCategory(@RequestBody Category category){

categoryService.createCategory(category);

return new ResponseEntity<>("Category added successfully",HttpStatus.CREATED);

}

@PutMapping("/admin/categories/{categoryId}")

public ResponseEntity<String> updateCategory(@RequestBody Category category,

@PathVariable Long categoryId){

try {

Category savedCategory = categoryService.updateCategory(category,categoryId);

return new ResponseEntity<>("Category updated "+categoryId,HttpStatus.OK);

}catch(ResponseStatusException e) {

return new ResponseEntity<>(e.getReason(),e.getStatusCode());

}

}

@DeleteMapping("/admin/categories/{categoryId}")

public ResponseEntity<String> deleteCategory(@PathVariable Long categoryId) {

try {

String status = categoryService.deleteCategory(categoryId);

//return new ResponseEntity<>(status,HttpStatus.OK);

//return ResponseEntity.ok(status);

return ResponseEntity.status(HttpStatus.OK).body(status);

}catch(ResponseStatusException e) {

return new ResponseEntity<>(e.getReason(),e.getStatusCode());

}

}

}

**Step 2: - create model**

Create package name “model” => com.ecommerce.project.model

Then create a class “Category” under model package

**Category.java code: -**

**package** com.ecommerce.project.model;

**import** jakarta.persistence.Entity;

**import** jakarta.persistence.GeneratedValue;

**import** jakarta.persistence.GenerationType;

**import** jakarta.persistence.Id;

@Entity(name="categories")

**public** **class** Category {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long categoryId;

**private** String categoryName;

**public** Category(Long categoryId, String categoryName) {

**super**();

**this**.categoryId = categoryId;

**this**.categoryName = categoryName;

}

**public** Category() {

}

**public** Long getCategoryId() {

**return** categoryId;

}

**public** **void** setCategoryId(Long categoryId) {

**this**.categoryId = categoryId;

}

**public** String getCategoryName() {

**return** categoryName;

}

**public** **void** setCategoryName(String categoryName) {

**this**.categoryName = categoryName;

}

}

**Step 3: - create service**

Create package name “service” => com.ecommerce.project.service

1. Then create an interface “CategoryService” under service package

**CategoryService.java code: -**

**package** com.ecommerce.project.service;

**import** java.util.List;

**import** com.ecommerce.project.model.Category;

**public** **interface** CategoryService {

List<Category> getAllCategories();

**void** createCategory(Category category);

String deleteCategory(Long categoryId);

Category updateCategory(Category category, Long categoryId);

}

1. Then create a class “CategoryServiceImpl” under service package

**CategoryServiceImpl.java code: -**

package com.ecommerce.project.service;

import java.util.ArrayList;

import java.util.List;

import java.util.Optional;

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

import org.springframework.http.HttpStatus;

import org.springframework.stereotype.Service;

import org.springframework.web.server.ResponseStatusException;

import com.ecommerce.project.model.Category;

import com.ecommerce.project.repositories.CategoryRepository;

@Service

public class CategoryServiceImpl implements CategoryService{

//private List<Category> categories = new ArrayList<>();

private Long nextId = 1L;

@Autowired

private CategoryRepository categoryRepository;

@Override

public List<Category> getAllCategories(){

return categoryRepository.findAll();

}

@Override

public void createCategory(Category category) {

category.setCategoryId(nextId++);

categoryRepository.save(category);

}

@Override

public String deleteCategory(Long categoryId) {

Category category = categoryRepository.findById(categoryId)

.orElseThrow(()-> new ResponseStatusException(HttpStatus.NOT\_FOUND,"Resource not found"));

categoryRepository.delete(category);

return "Category with Id: "+ categoryId + " deleted successfully";

}

@Override

public Category updateCategory(Category category, Long categoryId) {

Category savedCategory = categoryRepository.findById(categoryId)

.orElseThrow(()-> new ResponseStatusException(HttpStatus.NOT\_FOUND,"Resource not found"));

category.setCategoryId(categoryId);

savedCategory = categoryRepository.save(category);

return savedCategory;

}

}

**Step 4: - create repositories**

Create package name “repositories” => com.ecommerce.project.repositories

1. Then create an interface “CategoryRepository” under repositories package

**CategoryRepository.java code: -**

package com.ecommerce.project.repositories;

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

import com.ecommerce.project.model.Category;

public interface CategoryRepository extends JpaRepository<Category, Long>{

}

**Q) What is DBMS?**

Database Management Systems (DBMS) are **software systems used to store, retrieve, and run queries on data**. A DBMS serves as an interface between an end-user and a database, allowing users to create, read, update, and delete data in the database.

**Q) Introduction to Relational Databases Concepts**

Relational databases store data in rows and columns like a spreadsheet while non-relational databases (or NoSQL databases) use a variety of data models for accessing and managing data.

**Q) Overview of SQL**

**Q) Installing MySQL On Windows**

Download mysql software and install in local system

1. MySQL server
2. MySQL Workbench
3. MySQL Shell

**Q) What is ORM?**

An ORM (Object Relational Mapper) is a piece/layer of software that helps map your code Objects to your database.

-- Whenever there is a class, that class can be automatically converted to a table with its attributes being converted to columns.

-- So now the developer does not have to write queries for table creation, it’s created automatically.

-- Whenever an object is created, its data can be saved in the database as row in table, this is automatically handled by ORM.

-- Lets developer focus on application rather than SQL queries.

Here’s a breakdown of how ORM works:

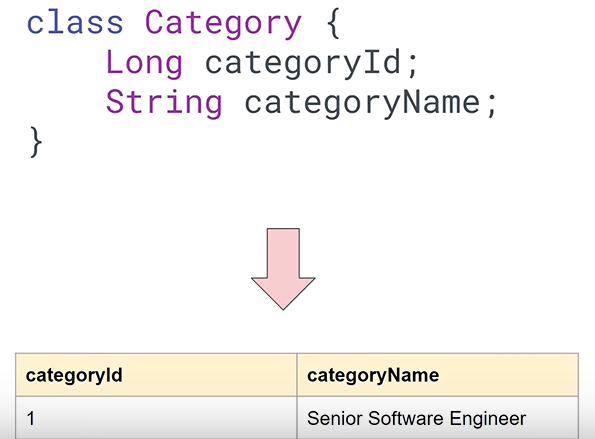
1. **Mapping**: ORM tools map database tables to classes in your code, and rows in those tables to instances of those classes. For example, if you have a User table in your database, you might have a User class in your code.
2. **CRUD Operations**: With ORM, you can perform Create, Read, Update, and Delete (CRUD) operations on database records using methods on these objects, rather than writing SQL queries. For instance, you might use user.save() to insert or update a user record, or User.find(id) to retrieve a record by its ID.
3. **Abstraction**: ORM abstracts away much of the raw SQL and database interactions, making it easier to work with data in a more intuitive and object-oriented manner.
4. **Consistency**: By using ORM, you can ensure that your data access code is consistent and easier to maintain. It also helps in managing database schema changes and migrations.

Popular ORM libraries and frameworks include:

* **Hibernate** (Java)
* **Entity Framework** (C#)
* **Django ORM** (Python)
* **SQLAlchemy** (Python)
* **ActiveRecord** (Ruby on Rails)

ORM is useful for speeding up development and reducing the amount of boilerplate code needed for database interactions, though it’s important to understand its limitations and performance considerations.

**Q) What is JPA?**

****

JPA stands for Java Persistence API. It's a specification in Java that provides a standard way to manage relational data in Java applications. Here’s a quick overview:

1. **Purpose**: JPA simplifies database interactions by allowing developers to work with Java objects rather than directly with database tables and SQL. It manages the mapping between Java objects and database tables.

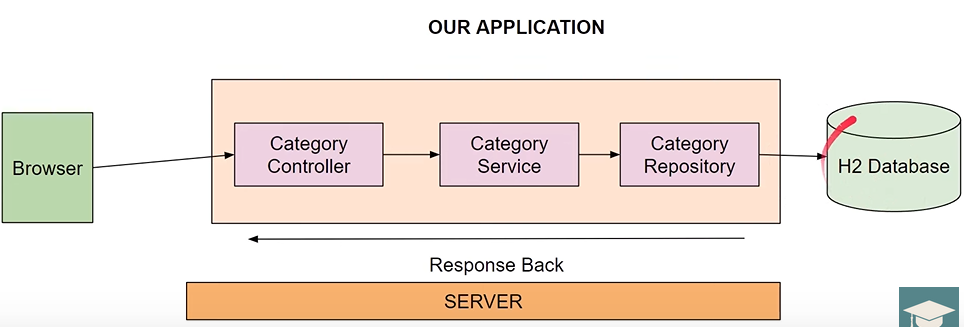
-- Easy and simple

-- Makes querying easier

-- Allow to save and update table

**Q) Let’s understand Data Layer**

Data access layer is the layer where all the repository classes resides.



**Q) H2 database**

Welcome to H2, the Java SQL database. The main features of H2 are:

* Very fast, open source, JDBC API
* Embedded and server modes; in-memory databases
* Browser based Console application
* Small footprint: around 2.5 MB jar file size

## Features

* Very fast, open source, JDBC API
* Embedded and server modes; disk-based or in-memory databases
* Transaction support, multi-version concurrency
* Browser based Console application
* Encrypted databases
* Fulltext search
* Pure Java with small footprint: around 2.5 MB jar file size
* ODBC driver

**Q) Configuring our Project using H2 database**

**Add below dependency in pom.xml (Spring Data JPA and H2 Database)**

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

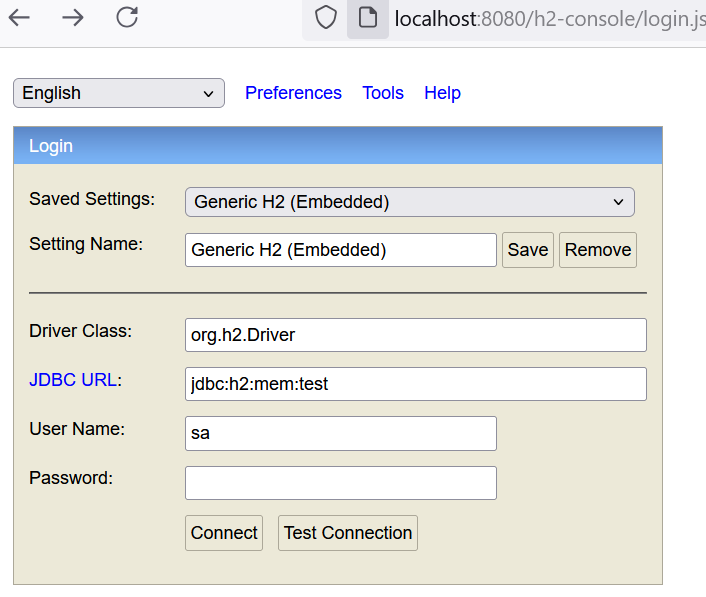
</dependency>

* 1. **Add below code in application properties file**

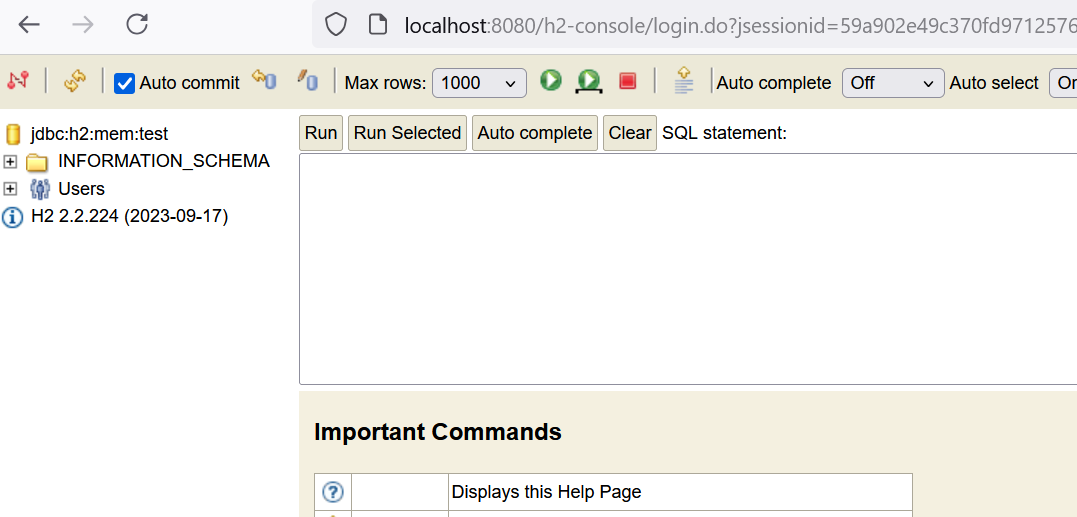
spring.h2.console.enabled=true

spring.datasource.url=jdbc:h2:mem:test

* 1. **Open url in browser: -** [**http://localhost:8080/h2-console**](http://localhost:8080/h2-console) **and copy and paste JDBC URL**

****

* 1. **If you connect h2 database**

****

**Q) Testing changes**

**Q) What is Lombok?**

Java is the most popular object-oriented programming language but it has some drawbacks. The major drawback is to write lots of **boilerplate** code. To overcome this drawback, project **Lombok** comes into existence. It is a tool that spices up our Java application. In this section, we will discuss the **project Lombok, features, Lombok package**.

a) Add below dependency in pom.xml

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

**b) update below code in model entity**

@Entity(name="categories")

@Data

@NoArgsConstructor

@AllArgsConstructor

**public** **class** Category {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long categoryId;

**private** String categoryName;

**}**

**Spring Security using JWT token**

**URL: -** [**https://www.youtube.com/watch?v=GH7L4D8Q\_ak**](https://www.youtube.com/watch?v=GH7L4D8Q_ak)

**Q) Role of spring security within the spring ecosystem:**

-- Sprong framework

-- Spring boot

-- Spring Data

-- Spring Security (Authentication and Authorization)

**Q) Authentication and Authorization?**

**Authentication: - Authentication** is proving Who you are?

**Authorization: -** Authentication is about what you are allowed to do after you have proven who you are.

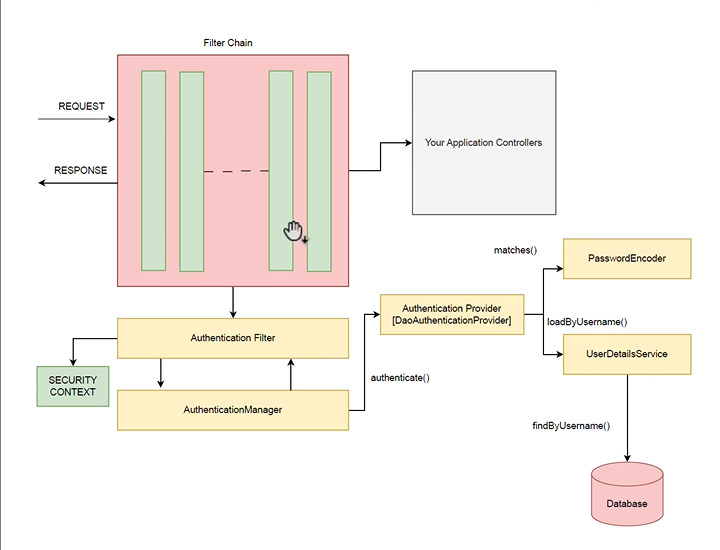
**Q) Spring security principle?**

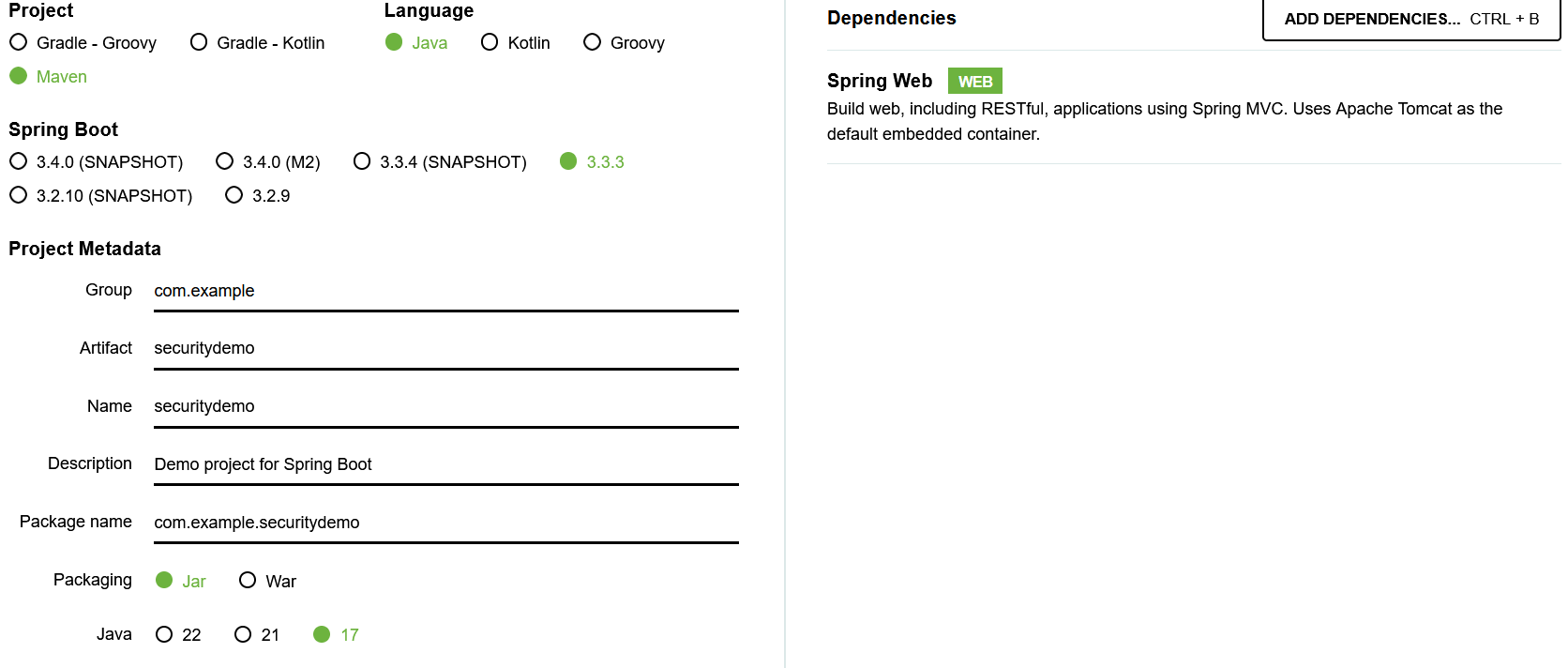
-- Input validation

-- Auditing and Logging

-- Regular updates and Patch Management

**Q) How Spring security works?**



****

**Step 1: - Add spring security dependency in pom.xml**

<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>

**Note: - If we want assign login username and password, put below code in application properties**

spring.security.user.name=admin

[spring.security.user.password=demo@123](mailto:spring.security.user.password=demo@123)

**Step 2: - Create controller “**GreetingsController” => under this package com.example.securitydemo

package com.example.securitydemo;

import org.springframework.security.access.prepost.PreAuthorize;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class GreetingsController {

@GetMapping("/hello")

public String sayHello() {

return "Hello";

}

@PreAuthorize("hasRole('USER')")

@GetMapping("/user")

public String userEndpoint() {

return "Hello User!";

}

@PreAuthorize("hasRole('ADMIN')")

@GetMapping("/admin")

public String adminEndpoint() {

return "Hello Admin!";

}

}

**Step 3: - Creation a basic authentication**

Create class name “SecurityConfig” => under this package com.example.securitydemo

**SecurityConfig.java code: -**

**package** com.example.securitydemo;

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

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

**import** org.springframework.security.config.Customizer;

**import** org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;

**import** org.springframework.security.config.annotation.web.builders.HttpSecurity;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

@EnableMethodSecurity

**public** **class** SecurityConfig {

@Bean

SecurityFilterChain defaultSecurityFilterChain (HttpSecurity http) **throws** Exception{

http.authorizeHttpRequests((requests) -> requests.anyRequest().authenticated());

//http.formLogin(Customizer.withDefaults());

http.httpBasic(Customizer.*withDefaults*());

**return** http.build();

}

}

**Step 4: - Creation a In-memory authentication and roles based**

@Bean

**public** UserDetailsService userDetailsService() {

UserDetails user1 = User.*withUsername*("user1")

.password("{noop}password1")

.roles("USER")

.build();

UserDetails admin = User.*withUsername*("admin")

.password("{noop}adminPass")

.roles("ADMIN")

.build();

**return** **new** InMemoryUserDetailsManager(user1,admin);

}

**Q) Creation a JWT authentication VVIP:**

* 1. **Add dependency H2 Database and Spring Data JPA in pom.xml**

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

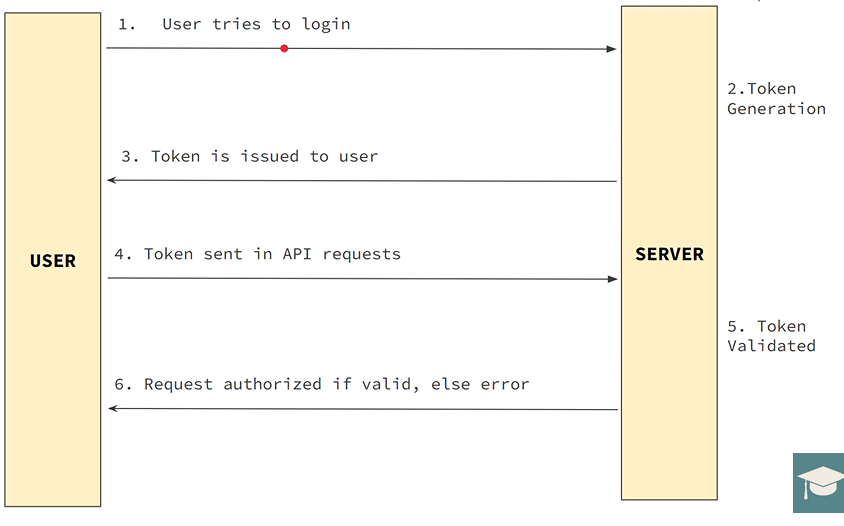
* 1. **Add below code for H2 database configuration in application.properties file**

spring.h2.console.enabled=true

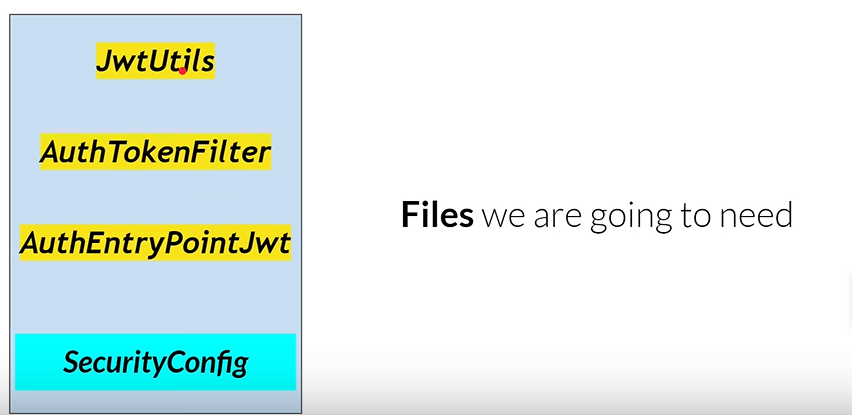
spring.datasource.url=jdbc:h2:mem:test

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

****

**JWT encode = Header + Payload + Signature**

****

1. **JWTUtils**: - contains utility methods for generating, parsing and validating JWTs. Include generating a token from a username,validating a JWT, and extracting the usename from a token.
2. **AuthTokenFilter**: - Filters incoming requests to check for a valid JWT in the header, setting the authentication context if the token is valid.
3. **AuthEntryPoingJwt**: - Provides custom handling for unauthorized requests, typically when authentication is required but not supplied or valid.When an unauthorized request is detected, its logs error and return a JSON response with an error message and status code, and the path attempted.
4. **SecurityConfig**: - Configures spring security filters and rules for the application.Set upthe security filter chain, permitting or denying access based on paths and roles.

**URL: -** [**https://github.com/jwtk/jjwt?tab=readme-ov-file#jdk-projects**](https://github.com/jwtk/jjwt?tab=readme-ov-file#jdk-projects)

**Step 1: - Add below dependency in pom.xml**

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.12.6</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.12.6</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->

<version>0.12.6</version>

<scope>runtime</scope>

</dependency>

**Step 2: - Create package name “jwt” => com.example.securitydemo.jwt**

1. Then create a class “JwtUtils” under jwt package

**JwtUtils.java code: -**

**package** com.example.securitydemo.jwt;

**import** java.security.Key;

**import** java.util.Date;

**import** javax.crypto.SecretKey;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.security.core.Authentication;

**import** org.springframework.security.core.userdetails.UserDetails;

**import** org.springframework.stereotype.Component;

**import** org.springframework.stereotype.Component;

**import** io.jsonwebtoken.\*;

**import** io.jsonwebtoken.io.Decoders;

**import** io.jsonwebtoken.security.Keys;

**import** jakarta.servlet.http.HttpServletRequest;

@Component

**public** **class** JwtUtils {

**private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(JwtUtils.**class**);

@Value("${spring.app.jwtSecret}")

**private** String jwtSecret;

@Value("${spring.app.jwtExpirationMs}")

**private** **int** jwtExpirationMs;

**public** String getJwtFromHeader(HttpServletRequest request) {

String bearerToken = request.getHeader("Authorization");

***logger***.debug("Authorization Header: {}", bearerToken);

**if**(bearerToken !=**null** && bearerToken.startsWith("Bearer ")) {

**return** bearerToken.substring(7);

}

**return** **null**;

}

**public** String generateTokenFromUsername(UserDetails userDetails) {

String username = userDetails.getUsername();

**return** Jwts.*builder*()

.subject(username)

.issuedAt(**new** Date())

.expiration(**new** Date((**new** Date()).getTime() + jwtExpirationMs ))

.signWith(key())

.compact();

}

**public** String getUserNameFromJwtToken(String token) {

//return Jwts.parserBuilder().setSigningKey(key()).build()

//.parseClaimsJws(token).getBody().getSubject();

**return** Jwts.*parser*()

.verifyWith((SecretKey) key())

.build().parseSignedClaims(token)

.getPayload().getSubject();

}

**private** Key key() {

**return** Keys.*hmacShaKeyFor*(Decoders.***BASE64***.decode(jwtSecret));

}

**public** **boolean** validateJwtToken(String authToken) {

**try** {

System.***out***.println("Validated");

Jwts.*parser*().verifyWith((SecretKey) key()).build().parseSignedClaims(authToken);

**return** **true**;

} **catch** (MalformedJwtException e) {

***logger***.error("Invalid JWT token: {}", e.getMessage());

} **catch** (ExpiredJwtException e) {

***logger***.error("JWT token is expired: {}", e.getMessage());

} **catch** (UnsupportedJwtException e) {

***logger***.error("JWT token is unsupported: {}", e.getMessage());

} **catch** (IllegalArgumentException e) {

***logger***.error("JWT claims string is empty: {}", e.getMessage());

}

**return** **false**;

}

}

1. **Add below code in application properties file**

spring.app.jwtSecret=mySecretKet123

spring.app.jwtExpirationMs=30000000000

1. **Then create a class “AuthTokenFilter” under jwt package**

**AuthTokenFilter.java code: -**

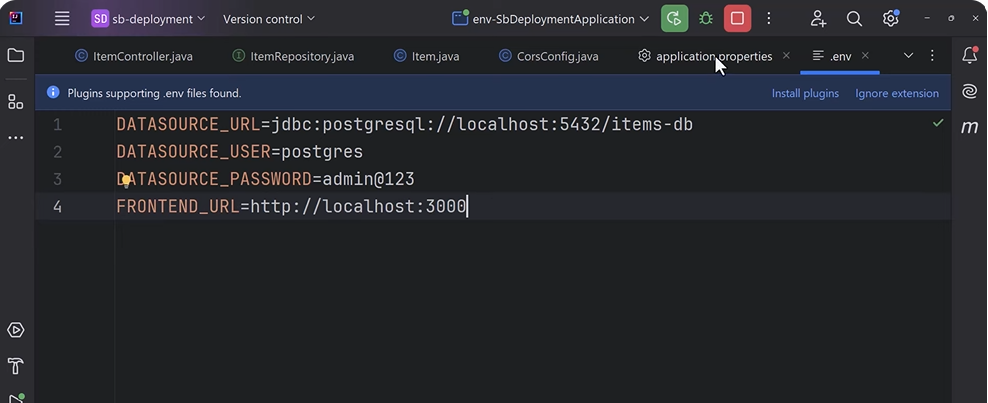
1. **Then create a class “**AuthEntryPointJwt**” under jwt package**

**AuthEntryPointJwt.java code: -**

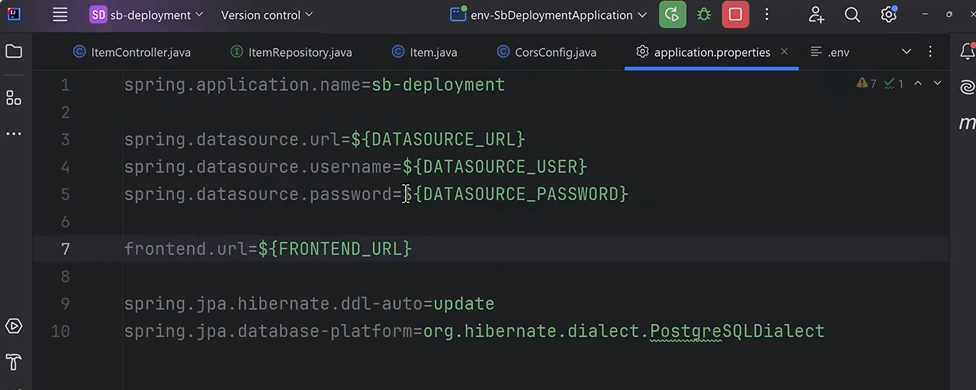
# Deploy a Full Stack Spring Boot & React App with Database for FREE | Step-by-Step Guide

**URL: -** [**https://www.youtube.com/watch?v=jBDTsf8jsEs**](https://www.youtube.com/watch?v=jBDTsf8jsEs)

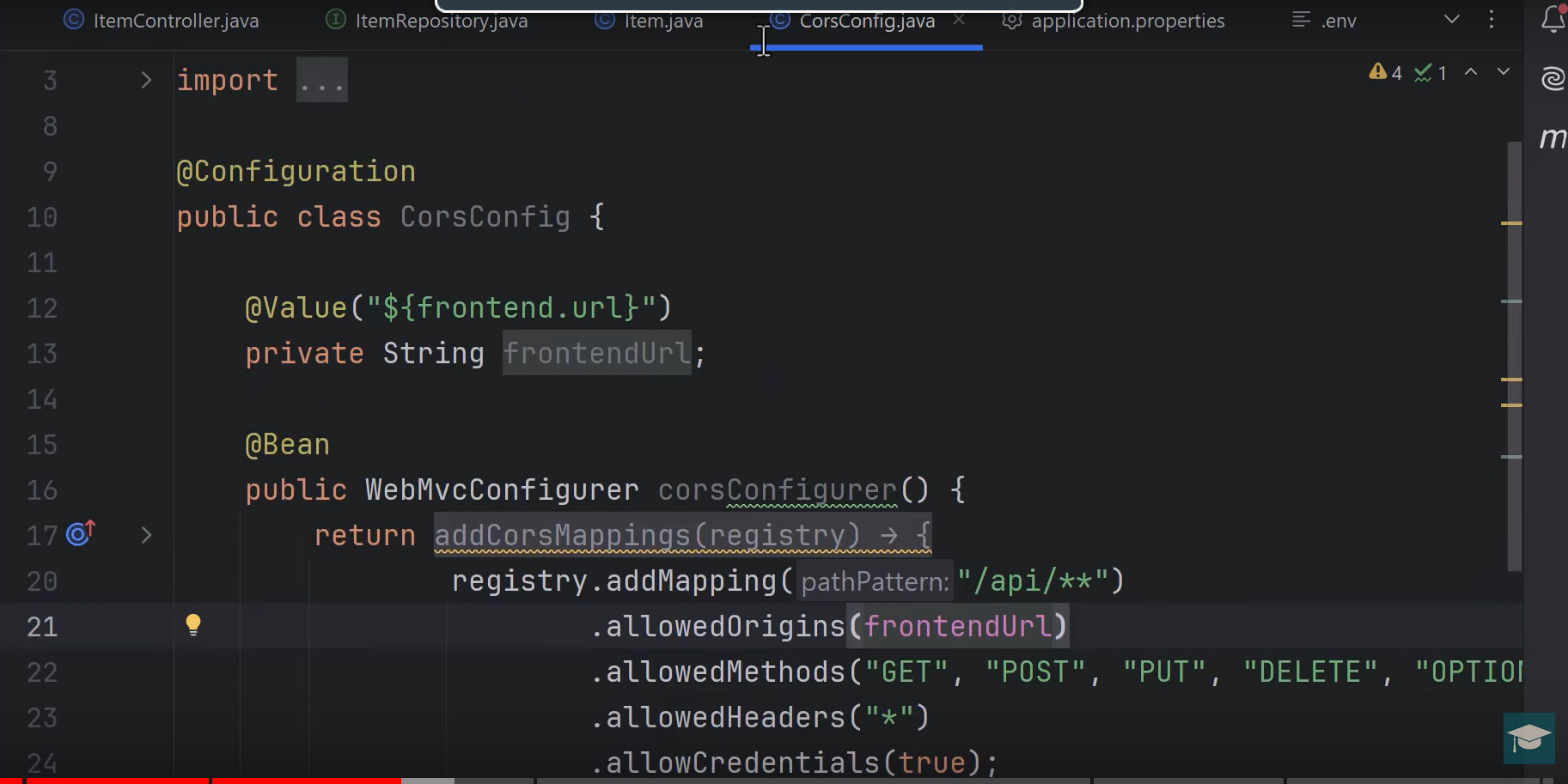
**Step 1: - We have create .env file to use environment variable in root springboot project folder**

****

**Step 2: - We have set all the env variables in application.properties file**

****

**Step 3: - Then we have to set frontend url in CorsConfig.java**

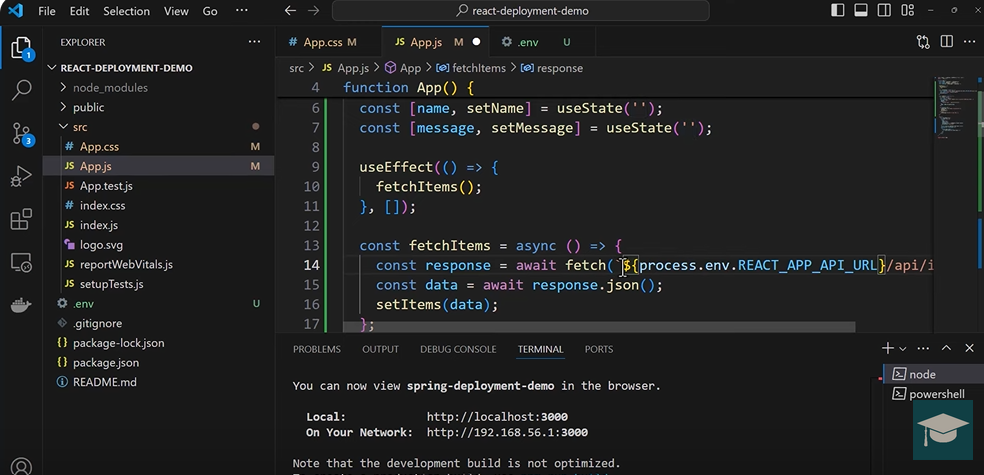
****

**Step 4:- create a environment(.env) file in the root of React project**

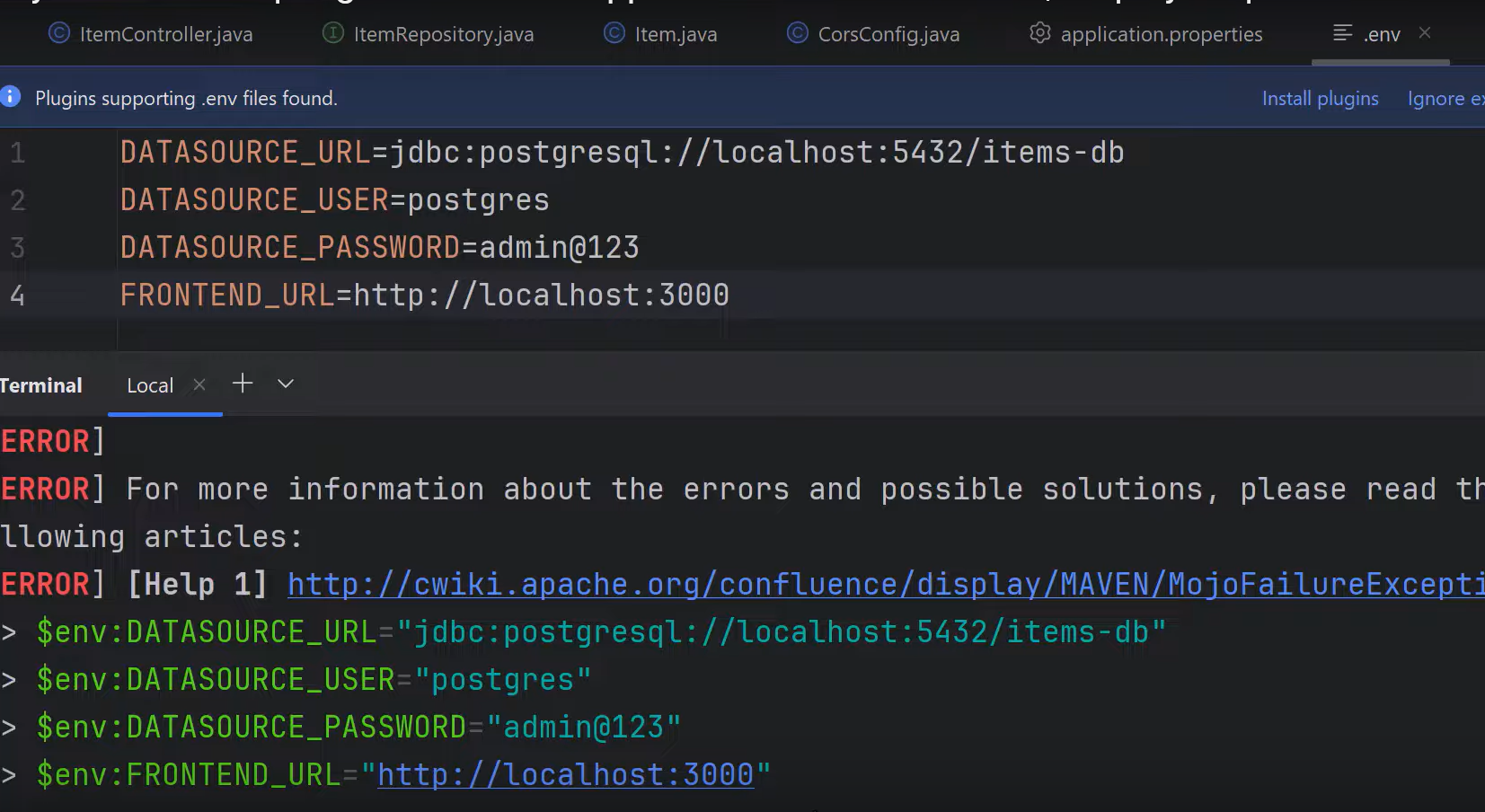
REACT\_APP\_API\_URL=http://localhost:8080

**Step 5: - Now used the variable in React project**

Const response = await fetch(`${ process.env. REACT\_APP\_API\_URL }/api/items`);

****

**NOTE: - Setup environment variable in different OS and then run command** > ./mvnw package

****

**Step 6:- Install docker desktop in local system**

**Step 7: - Build JAR of springboot project from terminal**

1. > ./mvnw package
2. To build jar From Eclipse, Right click on your maven project name then

Run as > Maven install

1. Right click maven project

choose Run As-> Maven Build ....

Type **package** in the **Goals** box.

Click Run.

**Step 8: - Create a Dockerfile in Springboot project root folder**

#use official maven image to build the spring boot app

FROM maven:3.8.4-openjdk-17 AS build

#set the working directory

WORKDIR /app

#copy pom.xml and install dependency

COPY pom.xml .

RUN mvn dependency:go-offline

#copy the source code and build the application

COPY src ./src

RUN mvn clean package -DskipTests

#use an official OpenJdk image to run the application

FROM openjdk:17-jdk-slim

#set the working directory

WORKDIR /app

#copy the built JAR file from the build stage

COPY --from=build /app/target/note-0.0.1-SNAPSHOT.jar

EXPOSE PORT 8080

ENTRYPOINT [“java”,”-jar”,”/app/note-0.0.1-SNAPSHOT.jar”]

**Step 9: - Now build the docker image step by step in CLI**

> docker build -t demo-deployment .

> docker tag demo-deployment binaysamanta/demo-deployment:latest

> docker push binaysamanta/demo-deployment:latest

**Step 10: - Now build React project and change .env file setting for production**

> npm run build

Then build folder will create automatically

**Final Step: -**  Deploy docker hub image in render.com and signup in the site.