**Spring Boot Framework:**

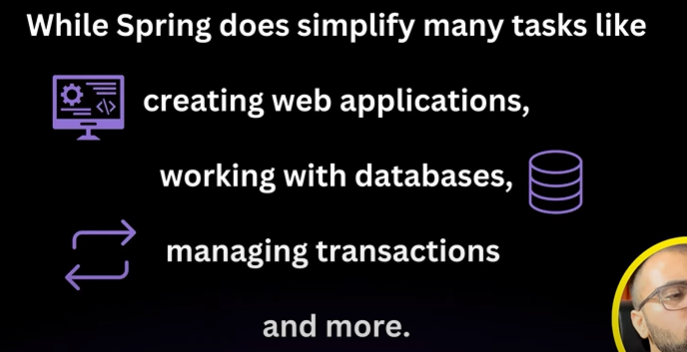
Spring Boot is a framework for building application in java programming language.

Spring Boot makes it easy to create stand alone, production-grade Spring based Applications that you can just run.

It provides some tools so that we can develop some applications.

The core Spring framework already reduces boilerplates code and provides a lot of helpful features for java applications.

However, Spring boot takes this convenience to the next level by focusing specifically on reducing the effort required to set up and configure a Spring application.



But setting up a Spring project can still involve quite a bit of manual configuration.

This is where Spring Boot Steps in.  
Spring Boot provides Auto-Configuration, Standalone Applications, Embedded Tom cat server.

Context path is called http://localhost:8080/contextpathname/api

//without Spring Boot, setting up a basic RESTful API requires manual configuration. In spring

//Controller

@RestController

@Requestmapping(“/api”)

public class MyController {

@GetMapping(“/hello”)

public String sayHello() {

return “Hello from Spring!”;

}

}

//Application Main

public class SpringApp{

public static void main(String[] args) {

ApplicationContext context = new AnnotationConfigApplicationContext(MyController.class);

}

}

We need to manually set up the Spring application context in the main method.

import org.springframework.boot.SpringApplication;

import org.spirngframework.boot.autoconfigure.SpringBootApplcation;

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

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

//With Spring Boot, the setup process is greatly simplified.

//Controller

@RestController

@SpringBootApplication

public class SpringBootApplicationExample{

@GetMapping(“/api/hello”)

public String sayHello() {

return “Hello from Spring Boot!”;

}

public static void main(String[] args){

SpringApplication.run(SpringBootApplicationExample.class, args);

}

}

We use the **@SpringBootApplication** annotation on the main class. This single annotation replaces the need for setting up a manual Spring application context.

We no longer need to explicitly create an application context using **AnnotationConfigApplicationContext** as Spring Boot handles that behind the scenes.

We use **SpringApplication.run()** to start the application, and Spring Boot takes care of configuring the embedded web server and other necessary components.

The **@SpringBootApplication** annotation alone brings in a lot of pre-configured features, including **automatic component scanning** and **embedded server configuration,** which would have required more steps in a traditional Spring setup.

In spring or Spring boot we they need bean only. In spring ApplicationContext and in Spring Boot @SpringBootApplication it will be created.

We can create the spring in two ways: 1> spring initializr 2> in sts, spring starter project

What is Maven?

Mave is a build automation tool. It is used to make the build

It is build automation tool, simplify the build, manage the dependencies,

Write mvn package in IntelliJ idea console it will validate, compile, test, jar will be created.

pom,.xml stands for **Project Object Model**.xml file. It contains information of our project, like how to build the project, and what external libraries we have,

if any project is having the pom.xml file then definitely that project is maven project. (build tool is maven).

Plugin helps to packaging your code into jar or war.

Previously we were creating the object like Car car = new Car();

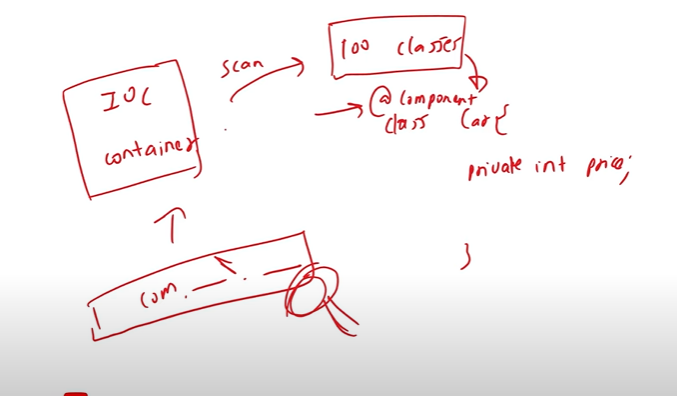
But here we will tell the spring to give the object of this one.

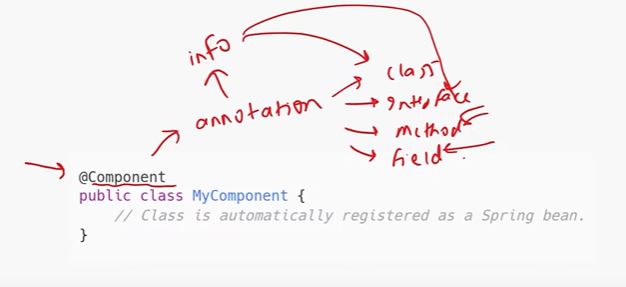
Object 🡪 externalize (Now we will create the object by him) (Inversion of Control)

Now Spring(IOC) will give the object of asked class.

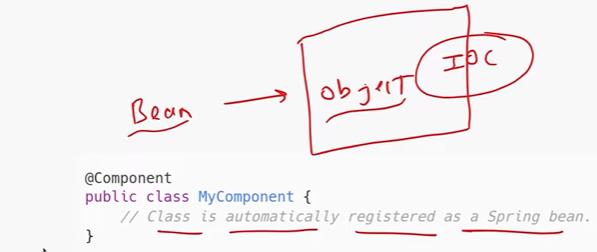
ApplicationContext is way to implement the IOC container.

ApplicationContext is way to achieve the IOC container. (IOC/ApplicationContext)





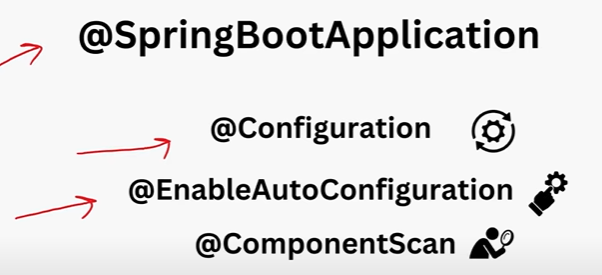




Annotation provides the information related to classes or methods, or interfaces or fields depending on the Annotation.

In A complete spring boot project only one @SpringBootApplication annotation will be there in the main class that is which are having main method.

@SpringBootApplication annotation works three annotations works like:



@CompnonentScan: This annotation scans the beans Which are having @Componenet it will scans the beans from the IOC/ApplicationContext.

@RestController: It is used to create the bean like @Compononet but @Componenet + something else, specialized version of componenet Annotation.

All the below annotations are used to create the APIs, used to create the end-point.

@GetMapping(“/..”)

@PostMapping(“/..”)

@PutMapping(“/..”)

@DeleteMapping(“/..”)

@PathMapping(“/..”)

@Autowire wherever we will use that class will be dependent on the particular object.(annotated object).

@EnableAutoConfiguration will provide the configuration automatically.(we have to give the dependency in the pom.xml file and application properties we have to pass the dabase name password and all that’s it.)

@Configuration annotation if we will use on the class then we will know that, that class will provide some configuration.

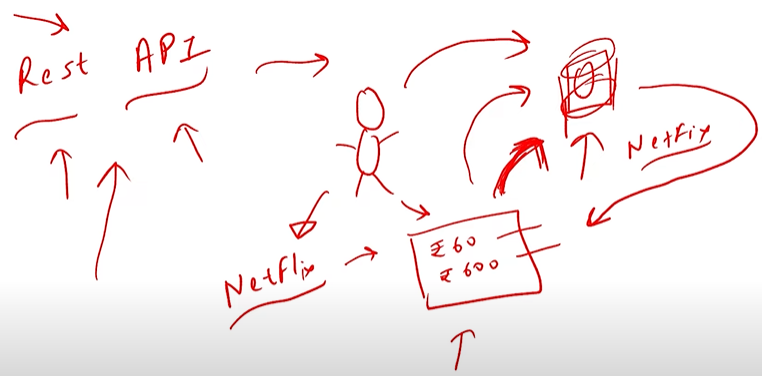
Most frequently @Configuration annotation are used with another annotation ie, @Bean annotation

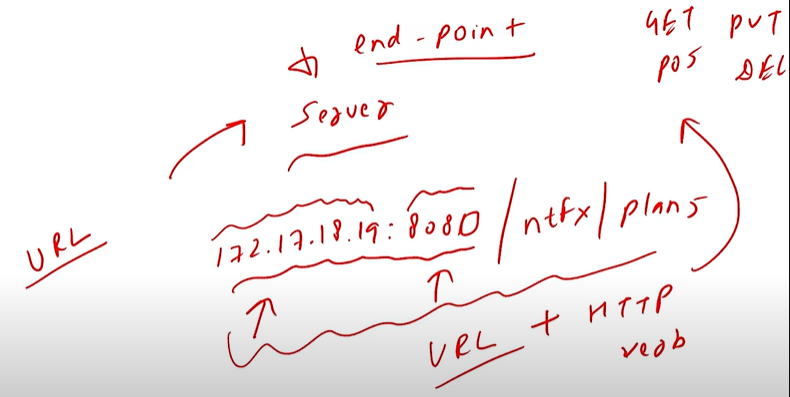
@Bean annotation is also used to create the bean(object), But we can’t use on the class. We will use on the function.

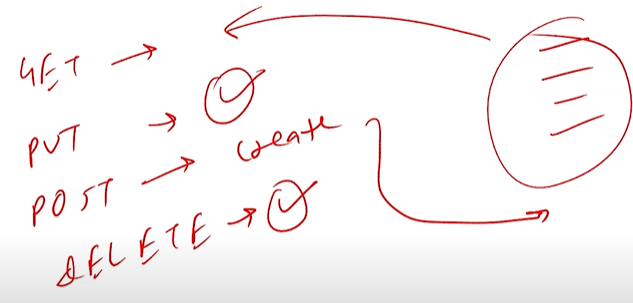
If we will use @Configuration on the class then we can create the bean inside that class with the help of function.

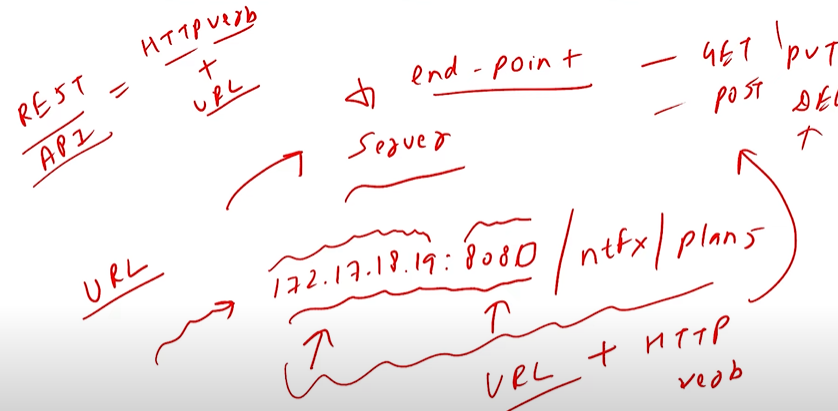
REST API’s

REST Representational Estate Transfer, API Application Programming Interfaces

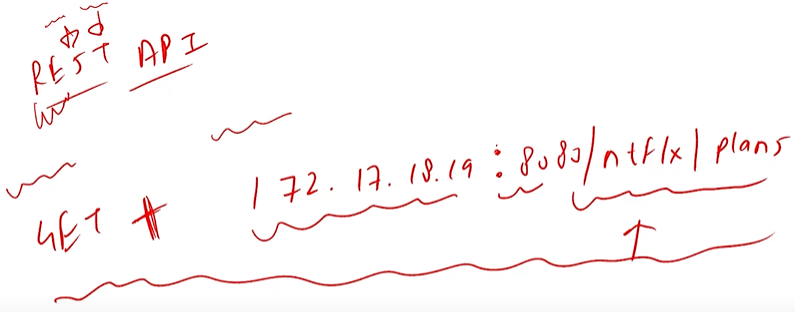








REST API = Http Verb + URL



Http verb having mainly four types:

Get, Post, Put, Delete

@RestController: A special type of classes/ components + additional functionality, which will handle our https request.

How many ends points will be there our url path will be there it will automatically return/converted into the json.

POJO class means Plain Old Java Object.

@RequestMapping is used to add the mapping on the entire controller class and endpoint will be started from their url

@PathVariable & @RequestParam

**Mongodb database:**

MongoDB is a database like in mysql tables are there like that here inside MongoDB collections will be there inside mysql inside table row and columns was there here inside mongodb collections will be there inside collections fields will be there, and row we called documents.

Commands:

C:\Windows\System32>mongosh

test> show dbs

test> use school (it will directly create the school database)

school> show collections

db.students.insertOne({"name": "Ram", "age":20}) (To insert the data into the databse)

db.students.insertOne({"name": "Shyam", "age": 25})

show collections (To see the database)

db.students.find() (To see the table data)

db.students.find().pretty() (To see in a beautiful way)

db.students.find({"name": "Ram"})

db.students.find({Name : "Ramkumar"})

db.students.deleteOne({name:"Ram"}) (To delete the data from the database.)

Collection is similar to table.

Documents is similar to row

Column is similar to field

**What is ORM?**

ORM stands for Object-Relational Mapping.

ORM is a technique used to map Java objects to database tables.

It allows developers to work with databases using object-oriented programming concepts, making it easier to interact with relational databases.

Consider a Java class *User* and a database table *users*.

ORM frameworks like Hibernate can map the fields in the *User* class to columns in the *users* table, making it easier to insert, update, retrieve and delete records.

**What is JPA?**

JPA stands for Java Persistence (permanently store the data) API (Set of rules).

JPA is a way to achieve ORM, includes interfaces and annotations that you use in your Java classes, requires a persistence provider (ORM tools) for implementation.

**Persistence Provider/ORM tools**

*To use JPA, you need a persistence provider.* A persistence provider is a specific implementation of the JPA specification. Example of JPA persistence providers include Hibernate, EclipseLink, and **OpenJPA**. These providers implement the JPA interfaces and provide the underlying functionality to interact with databases.

**Spring Data JPA**

*Spring Data JPA* is built on top of the JPA (Java Persistence API) specification, but it is not a JPA implementation itself. Instead, it *simplifies working with JPA* by providing higher-level abstractions and utilities. However, *to use Spring Data JPA effectively, you still need a JPA implementation*, such as Hibernate, EclipseLink, or another JPA-compliant provider, to handle the actual database interactions.

JPA is primarily designed for working with relational databases, where data is stored in tables with a predefined schema. MongoDB, on the other hand, is a NoSQL database that uses a different data model, typically based on collections of documents, which are schema-less or have flexible schemas. This fundamental difference in data models and storage structures is why *JPA is not used with MongoDB.*

In the case of MongoDB, you don’t have a traditional JPA persistence provider. MongoDB is a NoSQL database, and *Spring Data MongoDB* serves as the “persistence provider” for MongoDB.

*It provides the necessary abstractions and implementations to work with MongoDB in a Spring application.*

Spring Data MongoDB

<dependency>

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

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

</dependency>

*Query Method DSL* and *Criteria API* are two different ways to interact with a database when using Spring Data JPA for relational databases and Spring Data MongoDB for MongoDB databases.

Spring Data JPA is a part of the Spring Framework that simplifies data access in Java applications, while Spring Data MongoDB provides similar functionality for MongoDB.

*Query Method DSL* is a simple and convenient way to create queries based on method naming conventions, while the *Criteria API* offers a more dynamic and programmatic approach for building complex and custom queries.

Controller layer(Only create the end point) will call to the Service layer(Business logic will write)

Controller 🡪 Service 🡪 Repository

Post localhost:8080/journal

{

    "title": "Morning",

    "content": "Went to gym!",

    "id": 2

}

In open cmd with administrator and type

Mongosh <-|

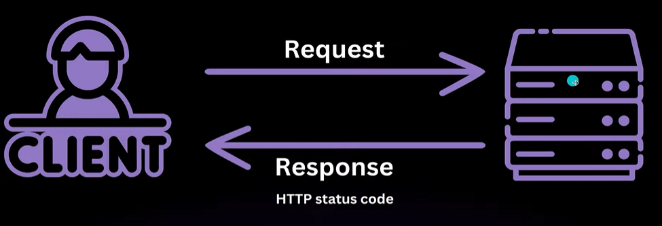
Show dbs

Use journaldb

Show collections

db.journal\_entries.find()

To delete all type db.journal\_entries.deleteMany({})



HTTP Status Code:

An HTTP status code is a three-digit numeric code returned by a web server as part of the response to an HTTP request made by a client. These status codes are used to convey information about the result or status of the requested operation.

HTTP status codes are grouped into five categories based on their first digit.

1xx (Informational): These status codes indicate that the request was received and understood, and the server is continuing to process it. These are typically used for informational purposes and rarely seen in practice.

2xx (Successful): These status codes indicate that the request was successfully received, understood and processed by the server.

3xx (Redirection): These status code indicate that further action is needed to complete the request. They are used when the client needs to take additional steps to access the requested resource.

4xx (Client Error): These status codes indicate that there was an error on the client’s part, such as a malformed request or authentication issues.

5xx (Server Error): These status code indicates that there was an error on the server’s part while trying to fulfill the request.

Examples:

**200 OK**: The request has been successfully processed, and the server is returning the requested resource.

**201 Created**: The request has been fulfilled, and a new resource has been created as a result.

**204 No Content**: The request was successful, but there is no response body (typically used for operations that don’t return data, like a successful deletion).

**301 Moved Permanently**: The requested resource has been permanently moved to a different URL.

**302 Found**: The HTTP status code 302 indicates that the requested resource has been temporarily moved to a different URL. When a server sends a response with a 302 status code, it typically includes a Location header field that specifies the new temporary URL where the client should redirect to.

**304 Not Modified**: The client’s cached version of the requested resource is still valid, so the server sends this status code to indicate that the client can use its cached copy.

**400 Bad Request**: The server cannot understand or process the client’s request due to invalid syntax or other client-side issues.

**401 Unauthorized**: The client needs to provide authentication credentials to access the requested resource.

**403 Forbidden**: The client is authenticated, but it does not have permission to access the requested resource.

**500 Internal Server Error**: A generic error message indicating that something went wrong on the server, and server could not handle the request.

**502 Bad Gateway**: The server acting as a gateway or proxy received an invalid response from an upstream server.

**503 Service Unavailable**: The server is currently unable to handle the request due to temporary overloading or maintenance.



**ResponseEntity**: The ResponseEntity class is part of the Spring Framework and is commonly used in Spring Boot applications to customize the HTTP response.

It provides methods for setting the response status, headers, and body. You can use it to return different types of data in your controller methods, such as JSON, XML, or even HTML.

You can use generics with ResponseEntity to specify the type of data you are returning.

**Project Lombok**: Lombok is a popular library in the Java ecosystem, often used in Spring Boot applications.

It aims to reduce the boilerplate code that developers have to write, such as getters, setters, constructors, and more.

Lombok achieves this by generating this code automatically during compilation, based on annotations you add to your Java classes.

Lombok generates bytecode for methods like getters, setters, constructors, equals(), hashCode(), and toString(), as specified by the annotations used in your code. This generated code is added to the compiled class files(.class files).

The Java Compiler compiles your classes, including the generated code. This means that the generated methods become part of your compiled class files.

When you run your application, the generated methods are available for use, just like any other methods in your classes.

To delete the data from the database

db.journal\_entries.deleteMany({id:{$in:[ObjectId(“88737373737”), ObjectId(“88737373734”)]}})

MongoDatabaseFactory is used to help us to make the connection with database.

**Transaction:** If we are saving any data it should be saved all the data in a single transaction if any one of them code is getting failed then entire transaction will be rolled back.

<https://cloud.mongodb.com/v2/684d8780da06de032ff464b5#/metrics/replicaSet/684d89238bc3ea3cc9a47bd1/explorer/journaldb/users/find>

**Spring Security:**

Spring Security is a powerful and highly customizable security framework that is often used in Spring Boot applications to handle authentication and authorization.

**Authentication:** The process of verifying a user’s identity (e.g, username and password).

**Authorization:** The process of granting or denying access to specific resources or actions based on the authenticated user’s roles and permissions.

**Configuration:**

<dependency>

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

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

Once the dependency is added, Spring Boot’s auto-configuration feature will automatically apply security to the application.

By default, Spring Security uses HTTP Basic Authentication.

The client sends an Authorization header Authorization: Basic <encoded-string> The server decodes the string, extracts the username and password, and verifies them. If they are correct, access is granted. Otherwise, an “Unauthorized” response is sent back.

Encoding

Credentials are combined into a strike like

Username:password

which is then encoded using Base64

By default, all endpoints will be secured. Spring Security will generate a default user with a random password that is printed in the console logs on startup.

You can also configure username & password in your application.properties

spring.security.user.name=name

spring.security.user.password=password

**Customize Authentication:**

@Configuraion //to define it’s a configuration class and create the bean for this one.

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests( )

.antMatchers(“/hello”).permitAll( )

.anyRequest( ).authenticated( )

.and( )

.formLogin( );

}

}

**@EnableWebSecurity:** This annotation signals Spring to enable its web security support. This is what makes your application secured. It’s used in conjunction with @Configuration.

**SecurityConfig extends WebSecurityConfigurerAdapter:** WebSecurityConfigurerAdapter is a utility class in the Spring framework that provides default configurations and allows customization of certain features. By extending it, you can configure and customize Spring Security for your application needs.

**@Override**

**protected void configure(HttpSecurity http) throws Exception{**

**super.configure(http);**

**}**

**Configure method:** This method provides a way to configure how requests are secured. It defines how request matching should be done and what security actions should be applied.

**http:** By using this HttpSecurity instance we can apply the authentication or authorization like on which end point we want to applied.

**http.authorizeRequest( ):** This tells Spring Security to start authorizing the request.

**.antMatchers(“/hello”).permitAll( ):** This part specifies that HTTP request matching the path /hello should be permitted (allowed) for all users, whether they are authenticated or not.

**.anyRequest( ).authenticated( ):** This is a more general matcher that specifies any request (not already matched by previous matchers) should be authenticated, meaning users have to provide valid credentials to access theses endpoints.

**.and():** This is a method to join several configurations. It helps to continue the configuration from the root (HttpSecurity).

**.formLogin( ):** This enables from-based authentication. By default, it will provide a form for the user to enter their username and password. If the user is not authenticated and they try to access a secured endpoint, they’ll be redirected to the default login form.





You can access /hello without any authentication. However, if you try to access another endpoint, you’ll be redirected to a login form.

When we use the .formLogin() method in our security configuration without specifying .loginPage(“/custome-path”), the default login page becomes active.

Spring Security provides an in-built controller that handles the /login path. This controller is responsible for rendering the default form when a GET request is made to /login.

By default, Spring Security also provides logout functionality. When .logout() is configured, a POST request to /logout will log the user out and invalidate their session.

Stateless means: another request doesn’t know what was the previous request was sent.

**Basic Authentication, by its design, is stateless.**

Some applications do mix Basic Authentication with session management for various reasons. This isn’t standard behavior and requires additional setup and logic. In such scenarios, once the user’s credentials are verified via Basic Authentication, a session might be established, and the client is provided a session cookie. This way, the client won’t need to send the Authorization header with every request, and the server can rely on the session cookie to identify the authenticated user.

When you log in with Spring Security, it manages your authentication across multiple requests, despite HTTP being stateless.

1. Session Creation: After successful authentication, an HTTP session is formed. Your authentication details are stored in this session.

2. Session Cookie: A JSESSIONID cookie is sent to your browser, which gets sent back with subsequent requests, helping the server recognize your session.

3. SecurityContext: Using the JSESSIONID, Spring Security fetches your authentication details for each request.

4. Session Timeout: Sessions have a limited life. If you’re inactive past this limit, you’re logged out.

5. Logout: When logging out, your session ends, and the related cookie is removed.

6. Remember-Me: Spring Security can remember you even after the session ends using a different persistent cookie (typically have a longer lifespan).

In essence, Spring Security leverages session and cookies, mainly JSESSIONID, to ensure you remain authenticated across requests.

We want our Spring Boot application to authenticate users based on their credentials stored in a MongoDB database.

This means that our users and their passwords(hashed) will be stored in MongoDB, and when a user tries to log in, the system should check the provided credentials against what’s stored in the database.

There are four steps that we have to follow:

1. A User entity to represent the user data model.
2. A repository UserRepository to interact with MongoDB.
3. UserDetailsService implementation to fetch user details.
4. A configuration SecurityConfig to integrate everything with Spring Security.

1.

@Document(collection = “users”)

public class User {

@Id

Private String id;

private String username;

private String password;

private List<String> roles;

//getters , setter and other methods.

}

2.

public interface UserRepository extends MongoRepository<User, String> {

Optional<User> findByUsername(String username);

}

3.

@Service

public class CustomUserDetailsService implements UserDetailsService {

@Autowired

private UserRepository userRepository;

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException{

User user = userRepository.findByUsername(username)

.orElseThrow(() -> new UsernameNotFoundException(“Username not found: “ + username));

return User.withUsername(user.getUsername())

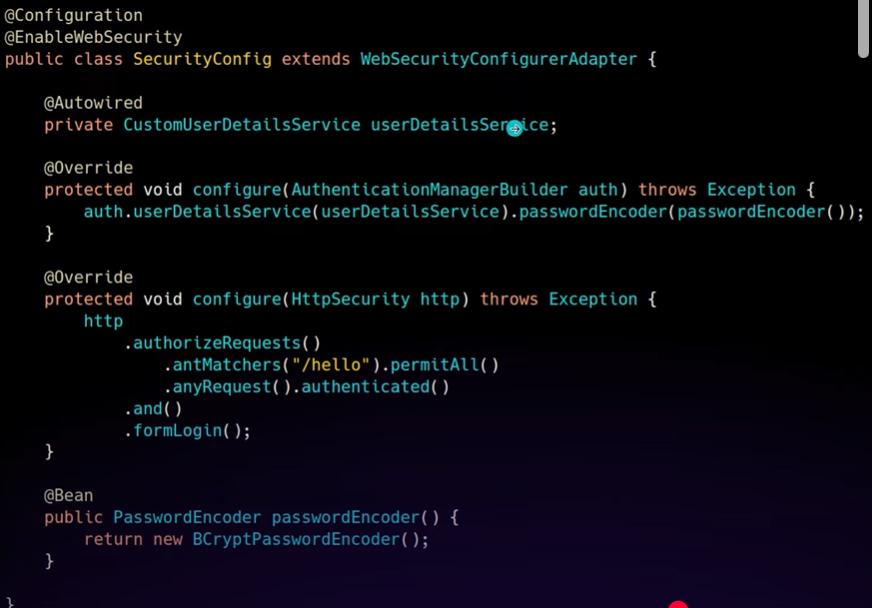
.password(user.getPassword())

.roles(user.getRoles().toArray(new String[0])).build();

}

}

4.



http.csrf().disable: csrf(Cross Sight Request Forgery) By default in spring security csrf is enabled.

Csrf protection enabled by default. Then spring security expects that you will send one token in request csrf token.

**CSRF:** csrf is a cyber attack where a malicious website/program can trick you into submitting a request that you don’t want to send.

Cluster: A group of nodes.

**Spring Security Customization:**

package net.engineeringdigest.journalApp.config;

import net.engineeringdigest.journalApp.service.UserDetailsServiceImpl;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.Customizer;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

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

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

import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

public class SpringSecurity {

    @Autowired

    private UserDetailsServiceImpl userDetailsService;

    @Bean

    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        return http.authorizeHttpRequests(request -> request

                        .requestMatchers("/public/\*\*").permitAll()

                        .requestMatchers("/journal/\*\*", "/user/\*\*").authenticated()

                        .requestMatchers("/admin/\*\*").hasRole("ADMIN")

                        .anyRequest().authenticated())

                .httpBasic(Customizer.withDefaults())

                .csrf(AbstractHttpConfigurer::disable)

                .build();

    }

    @Autowired

    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

        auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());

    }

    @Bean

    public PasswordEncoder passwordEncoder() {

        return new BCryptPasswordEncoder();

    }

}

<https://cloud.mongodb.com/v2/684d8780da06de032ff464b5#/metrics/replicaSet/684d89238bc3ea3cc9a47bd1/explorer/journaldb/users/find>

<https://cloud.mongodb.com/v2/684d8780da06de032ff464b5#/security/network/accessList>

Till now we were using the application.properties file for our configurations. Now we will se Spring Boot application how it reads the application.properties file or is there any other way to write these properties. If multiple way are there then which way we should use. If there are two ways then spring Boot application in which order it will read for our configurations.

1.classPath: A classPath is a just of list of jar’s and directories which is used by JVM. JVM needs bytecode only and bytecode will be available in the classpath. classPath is way to tell jvm like these things are here you can use it.

Inside classPath .class file, jar’s, configurations file will be available.

Q. How Spring Boot application find the applications.properties file?

Since it is written inside src/main/resources/application.properties

Inside springboot src/main/resource’s contents is added in the classPath so we don’t need to do anything, SpringBoot will automatically detects it. If we will put it in another folder structure then we need to setup the path of it. Also static & templates will be available inside the resources, And whatever dependencies you have used inside pom.xml it is also packaged in the classPath jar’s.

application.properties syntax are: key = value;

So Configuration we can provide in second way like YAML(YML).

Like in application.properties. format are .properties but here we will write application.yml.

It is called YAML. It’s a recursive acronym. In it’s full comes YAML. It means, YAML is not like a traditional markup language like XML in this we use tags, but in YAML is human readable unlike XML. In this we use indentation to show him what properties we are setting. It’s full form is YAML ain’t markup language. And File extension is .yml

**Precedence Order of configuration:**

Higher priority will be application.properties file over application.yml in case same property you have written.

**So, There are three ways to provide the configuration in SpringBoot:**

**1> application.properties**

spring.application.name=Journal App  
  
spring.data.mongodb.uri=mongodb+srv://rr7691435:Nu2mDuCOVF7abc9b@cluster0.tcnqqjo.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0  
spring.data.mongodb.database=journaldb  
spring.data.mongodb.auto-index-creation=true  
  
server.port = 8081  
server.servlet.context-path=/abcd

**2> application.yml**

spring:  
 data:  
 mongodb:  
 uri: mongodb+srv://rr7691435:Nu2mDuCOVF7abc9b@cluster0.tcnqqjo.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0  
 database: journaldb  
 auto-index-creation: true  
  
server:  
 port: 8081  
 servlet:  
 context-path: /journal

**3> Command Line:**

.\mvnw clean

.\mvnw package

dir target

java -jar target/journalApp-0.0.1-SNAPSHOT.jar

ctrl + c to stop the server then,

We can also give the property here (inside terminal) as a command line argument. Like:

java -jar target/journalApp-0.0.1-SNAPSHOT.jar –property=value like,

java -jar target/journalApp-0.0.1-SNAPSHOT.jar –server.port=9090

So, first priority goes to **Command Line argument**, then **application.properties** then **application.yml** file

and inside Command Line Argument we have to use the application.properties syntax

if you some press ctrl + c then server will be stop then we can manually set it by using like click on the JournalApplication above dropdown then go to the Edit Configurations.. then select the project and then inside program arguments box then type the –server.port=9090

Here we are also doing the same thing as done the command line.

**Context-path:** like localhost:8080/admin/all-users this is my endpoint but if we have mentioned in our application.properties file like server.servlete.context-path=/journal

Then we have to use localhost:8080/journal/admin/all-users this will be our url.

**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -Testing’s- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

Whenever we are making any API whatever the code we write then it is recommended to you to test it.

There are two types of testing are there:

1. Unit Testing : Suppose we have made any api in that multiple components are there For ex: In that written four functions, So four functions will be components, so testing of multiple individual components is called Unit Testing.

So how we will do the Unit testing like: we have one functions, I will unit test of this functions, that whatever I want the result like that it is working or not? If it is working then we will move another method, then we will write them, then it called them Test Driven Development. (however we are developing parallelly we are testing them.).

Inside the Java One framework is there called JUnit. We use this for Unit testing.

In java one framework is there **JUnit**. We will use this for Unit Testing. It’s full form is Java Unit.

Setup:

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

This dependency we have to add only during Testing phase.(means Whenever we will create the Jar that time we don’t have to use). As it is needed only during testing.

Inside this dependency one more dependency JUnit jupiter dependency will be available.

<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter</artifactId>  
 <version>5.10.5</version>  
 <scope>compile</scope>  
</dependency>

It is already included. We don’t need to do anything.So, Initially they told this JUnit, But when version 5 came and our Fifth planate is Jupiter that’s why their team named it JUnit Jupiter.

Because when you integrate it in old application, It may be you use Junit4

Inside src/test/JournalAppApplicationTests file will be there by default. Maven can be used to test also.

Example of code:

package com.jspiders.journalApp.service;  
  
import com.jspiders.journalApp.entity.User;  
import com.jspiders.journalApp.repository.UserRepository;  
import org.junit.jupiter.api.\*;  
import org.junit.jupiter.params.ParameterizedTest;  
import org.junit.jupiter.params.provider.ArgumentsSource;  
import org.junit.jupiter.params.provider.CsvSource;  
import org.junit.jupiter.params.provider.ValueSource;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
/\*without this annotation spring application context will not be injected and other @component or @Autowired annotation will be null(component will not be created). This annotation tells that we have to start the application context or just like i am starting my application.\*/

@SpringBootTest  
public class UserServiceTests {  
  
 @Autowired  
 private UserRepository userRepository;  
  
 @Autowired  
 private UserService userService;  
  
 @Test //Any method if we want to run as test then we have to annotate it by using @Test annotation.  
 public void testAdd()  
 {  
 assertEquals(4, 2+2);  
 }  
  
 @Test  
 public void testFindByUserName() {  
  
  *assertEquals(4, 2+2);* assertNotNull(userRepository.findByUserName("ram"));  
 } //This complete method block is called one Test.  
  
 @Disabled //If we don't want to run this test then we'll use this @Disabled annotations.  
 @Test  
 public void testFindByUserName() {  
 User user = userRepository.findByUserName("ram");  
 assertTrue(!user.getJournalEntries().isEmpty());  
 }

@BeforeEach //it’s used if you wanna initialize anything and before each test then we can use this  
 public void setUp() {  
  
 }  
  
 @BeforeAll //it will run if there any thounds of test cases are there. Then this will execute first.  
 public void toRead() {  
  
 }  
  
 @AfterEach //this will execute after executing all the tests all and the tests  
 public void setUpp()  
 {  
  
 }  
  
 @AfterAll //It will run after ending the execution of all the test.  
 public void setupe()  
 {  
  
 }  
  
 @Disabled  
 @ParameterizedTest  
 @CsvSource({  
 "1, 1, 2", //1st test case  
 "2, 10, 12",  
 "3, 3, 9"  
 })  
 public void test(int a, int b, int expected)  
 {  
 *assertEquals*(expected, a+b);

/\*I want to give multiple values then will use @ParameterizedTest annotations(its value like working loops will rise a, b then expected, a b then expected) so values will rise from @CsvSource()\*/  
 }  
  
 //we can also use this test to check for parametrized test  
 @ParameterizedTest  
 @ValueSource(strings = { // if we check integer value then will use ints //Here we can use EnumSource even we can create CustomSource  
 "ram",  
 "shyam",  
 "vipul"  
 })  
 public void testFindByUserName(String name){  
 *assertNotNull*(userRepository.findByUserName(name), "failed for: "+ name);

/\*Above we are writing some message but it is giving in console like which component has not worked.\*/  
 }  
  
 //We can also create one custom source  
 @ParameterizedTest  
 @ArgumentsSource(UserArgumentsProvider.class)  
 public void testSaveNewUser(User user) {  
 *assertTrue*(userService.saveNewUser(user));  
 }  
}

package com.jspiders.journalApp.service;  
  
import com.jspiders.journalApp.entity.User;  
import org.junit.jupiter.api.extension.ExtensionContext;  
import org.junit.jupiter.params.provider.Arguments;  
import org.junit.jupiter.params.provider.ArgumentsProvider;  
  
import java.util.stream.Stream;  
  
public class UserArgumentsProvider implements ArgumentsProvider {  
  
 @Override  
 public Stream<? extends Arguments> provideArguments(ExtensionContext extensionContext) throws Exception {  
 return Stream.*of*(  
 Arguments.*of*(User.*builder*().userName("shyam").password("shyam").build()),  
 Arguments.*of*(User.*builder*().userName("suraj").password("").build())  
 );  
 }  
}

1. Integration Testing:

**Mockito:**

Mockito is a mocking framework for Java applications that is used for effective unit testing. It plays a vital role in developing testable applications. In the context of Spring Boot, Mockito is **often used alongside JUnit to create mock objects, define their behavior, and verify interactions**. This allows developers to test their code without relying on external dependencies, making the tests faster and more reliable. The spring-boot-starter-test dependency includes JUnit, Mockito, and other useful testing libraries, making it easy to set up and use Mockito in Spring Boot projects.

---

If we used @Autowired anything then spring ApplicationContext will start completely. Then it takes time to start, it connects with database and components will start and initialized the dependency.

If you want make things getting fast. And you don’t want to do @Autowired of userService any dependency then you can **Mock** it.

It means actual service/actual dependency without using you can Mock it.

Suppose that we have UserDegailsServiceImpl we have to test it, we have to test loadUsreByUserName in this method we are using userRepository that means it will go the database and it will fetch the username then it return. Suppose we have big application so it will take time to load.

So we don’t want to use the @Autowired dependency we have to Mock it. It means we will make the fake repository. Whenever we call findByUserName then it will return the dummy username to test the rest methods. We don’t want to fetch from the actual database, bcz it will take time. So, this is called Mocking.

We can achieve this using Mockito.

@Mock annotation not interconnected with ApplicationContext. We have to use the @MockBean

Suppose that we have one scenario like we have to Mock one thing but one thing we don’t have to one thing then we have to use @SpringBootTest means we have to use SpringContext.

**1st way by using SpringContext.**

package com.jspiders.journalApp.service;  
import com.jspiders.journalApp.entity.User;  
import com.jspiders.journalApp.repository.UserRepository;  
import org.junit.jupiter.api.Assertions;  
import org.junit.jupiter.api.Test;  
import org.mockito.ArgumentMatchers;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.mockito.Mockito.\*;  
import java.util.ArrayList;  
import static org.mockito.Mockito.*when*;  
  
@SpringBootTest //Here we were playing with ApplicationContext means beans was creating. that's why we were using @Autowired and @MockBean  
public class UserDetailsServiceImplTests {  
  
 @Autowired  
 private UserDetailsServiceImpl userDetailsService;  
  
  
 @MockBean //@MockBean we will use when we are using ApplicationContext means Without using @SpingBootTest if we have to use @Mock bcz we will play with dummy actual repository.  
 private UserRepository userRepository;  
  
 @Test  
 void loadUserByUsernameTest()  
 {  
 *when*(userRepository.findByUserName(ArgumentMatchers.*anyString*())).thenReturn(User.*builder*().userName("ram").password("inrkiinrik").roles(new ArrayList<>()).build());  
 UserDetails user = userDetailsService.loadUserByUsername("ram");  
 Assertions.*assertNotNull*(user);  
 }  
}

**2nd way to Achieve Mockito(Without using SpringContext)**

package com.jspiders.journalApp.service;  
  
import com.jspiders.journalApp.entity.User;  
import com.jspiders.journalApp.repository.UserRepository;  
import org.junit.jupiter.api.Assertions;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.ArgumentMatchers;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.MockitoAnnotations;

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

import java.util.ArrayList;  
import static org.mockito.Mockito.*when*;

//Here we are not using SpringContext  
public class UserDetailsServiceImplTests{  
  
 @InjectMocks //It is used to inject all the dependencies and it will search for Mock annotated dependency.  
 private UserDetailsServiceImpl userDetailsService;  
  
 @Mock  
 private UserRepository userRepository; //It will be inject to @InjectMocks annotated dependency.  
  
 @BeforeEach  
 void setUp()  
 {  
 MockitoAnnotations.*initMocks*(this); //Here we are initializing all the mocks before executing any Test  
 }  
  
 @Test  
 void loadUserByUsernameTest() {  
 *when*(userRepository.findByUserName(ArgumentMatchers.*anyString*())).thenReturn(User.*builder*().userName("ram").password("inrkiinrik").roles(new ArrayList<>()).build());  
 UserDetails user = userDetailsService.loadUserByUsername("ram");  
 Assertions.*assertNotNull*(user);  
 }  
}

If we use @Mock annotation then it will not work incase of SpringContext. If we are using ApplicationContext then we have to use the @MockBean annotations to get the bean.

If we will use @InjectMocks then it will inject all the beans. According to this one it will ask the @Mock annotated dependency to inject itself.

And one more thing if we will use the @Mock then we have to initialize the dependency by using this one.(method name can be anything)

@BeforeEach  
void setUp()   
{  
 MockitoAnnotations.*initMocks*(this); //Here we are initializing all the mocks before executing any Test and it will initialize all the mocks inside this class.  
}

**Spring Profiles:**

If we will develop any application then there will be two environment One will be Development and Second-one will be Production.

In Development Environment we can do anything it doesn’t matter but real user data will not be there.

In Production Environment where we will deploy our application and actual user will use that.

Ex: Our Journal Entry application is in the development environment so that we can do any changes.

But when production it and it will go live then actual user will use them.

So basically our project is working in two environments.

So basically, difference will come in configuration(application.properties) so generally we use one MongoDB server in configuration but it can be in production we will use another MongoDB server, where actual user data will be there, it can be there port will be different, context-path will be different. So basically, here two profiles will be there, Development & Production Profile.

Spring Boot provide these facilities that we make easily configuration based on environment and we can change it. By default, one Profile will be there that is Default Profile.

We can add new Profile, inside src/resource/application.properties so this is default profie’s application property. We can put any name of profile. We can create multiple profiles like

application.yml -> will be by-default configuration profile.

application-dev.yml

application-prod.yml

So basically our default profile that application.yml is mandatory in this profile we will set like this:

**application.yml**spring:  
 profiles:  
 active: dev

So, It will check my active profile is dev so it will jump to application-dev.yml And our application-dev.yml file will be:

**application-dev.yml**

spring:  
 data:  
 mongodb:  
 uri: mongodb+srv://rr7691435:Nu2mDuCOVF7abc9b@cluster0.tcnqqjo.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0  
 database: journaldb  
 auto-index-creation: true  
  
server:  
 port: 8080  
 servlet:  
 context-path: /journal

This will be same as production profile but In real-world scenario it will be different because it will be live working and user will have been using this application. Suppose I have changed only port=8081

**application-prod.yml**

spring:  
 data:  
 mongodb:  
 uri: mongodb+srv://rr7691435:Nu2mDuCOVF7abc9b@cluster0.tcnqqjo.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0  
 database: journaldb  
 auto-index-creation: true  
  
server:  
 port: 8081  
 servlet:  
 context-path: /journal

And if we will write application.yml profile like this then it will jump to the production profile.

**application.yml**spring:  
 profiles:  
 active: prod

In production environment our

server:  
 port: 8080  
 servlet:  
 context-path: /journal

will be same so we will use in the application.yml profile itself but it will be override and work only production profile, but MongoDB server username and password will be there in the different profile, so we can’t write in the application.yml profile.

We can set this application.yml property using command line argument inside intellij, we can break down application.yml property in two profile and we can set them using intellij and using command line argument. So now we will be having only two profiles like

application-dev.yml and application-prod.yml so our app can’t be run because I didn’t specify that which profile is active. But suppose that we have all these things I made the jar using command line then it works on local, but I went to staging in staging I have set dev profile inside application.yml so it will work, but when I will go to the production then again I will change the application.yml like active: prod then we make the jar then we will push No we don’t do this we delete the application.yml and their code inside our intellij Edit Configuration option then inside **Environment variables** will write **spring.profiles.active=dev** which is basically application.properties format. Then it works. But when I will go to the server then I don’t use intellij then, In production we will use the Jar. By using jar -jar to clean jar will use, **.\mvnw clean pacakage** But in our case it won’t work because we have written the test and inside this test we must have use the **@SpringBootTest** annotations. So, SpringContext will be created and it will connect with MongoDB server and all. So we have to use the

**.\mvnw clean package -D spring.profiles.active=dev**

If we would not have used the Test then by **using .\mvnw clean pacakage** it will work

.\mvnw clean package -D spring.profiles.active=dev

ls

cd .\target\

ls

java -jar .\journalApp-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev

java -jar .\journalApp-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev

In this command we are running SpringBoot application and --spring.profiles.active=dev we are passing the arguments to that application. But below we are setting the JVM property which maven will run.

.\mvnw clean package -D spring.profiles.active=dev

-D flag is standard way to set java system properties inside command line argument.

By using our local machine we can’t able to access the production’s DB server. So companies are whitelisting production’s MongoDB server only works with only these two servers. So these two server will be whitelisted to work only these two servers. But when we run this jar on production then we pass what it profile. It may be your these works will do Jenkins. Jenkins will automate these things we don’t have to do anything. Jenkins is UI so it will set **.\mvnw clean package -D spring.profiles.active=dev** by clicking one button this will run and by other this will run **java -jar .\journalApp-0.0.1-SNAPSHOT.jar --spring.profiles.active=prod**

In Spring Config We can set which bean we have to load or not load based on profile.

So based on profile we can create the Bean.

package com.jspiders.journalApp.config;  
  
import com.jspiders.journalApp.service.UserDetailsServiceImpl;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Profile;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.config.Customizer;  
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;  
import org.springframework.security.config.http.SessionCreationPolicy;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.security.web.SecurityFilterChain;  
  
@Configuration  
@EnableWebSecurity  
@Profile("prod")  
public class SpringSecurityProd {  
  
 private final UserDetailsServiceImpl userDetailsService;  
  
 public SpringSecurityProd(UserDetailsServiceImpl userDetailsService) {  
 this.userDetailsService = userDetailsService;  
 }  
  
// @Autowired //here doing the same whatever we have done the above  
// private UserDetailsServiceImpl userDetailsService;  
  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 return http  
 .authorizeHttpRequests(request -> request  
 .anyRequest().authenticated())  
 .httpBasic(Customizer.*withDefaults*())  
 .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.*STATELESS*))  
 .csrf(AbstractHttpConfigurer::disable)  
 .build();  
 }  
  
 @Bean  
 public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {  
 return config.getAuthenticationManager();  
 }  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
}

package com.jspiders.journalApp.config;  
  
import com.jspiders.journalApp.service.UserDetailsServiceImpl;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Lazy;  
import org.springframework.context.annotation.Profile;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.config.Customizer;  
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;  
import org.springframework.security.config.http.SessionCreationPolicy;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.security.web.SecurityFilterChain;  
  
@Configuration  
@EnableWebSecurity  
@Profile("dev")  
public class SpringSecurity {  
  
 private final UserDetailsServiceImpl userDetailsService;  
  
 public SpringSecurity(@Lazy UserDetailsServiceImpl userDetailsService) {  
 this.userDetailsService = userDetailsService;  
 }  
  
// @Autowired //here doing the same whatever we have done the above  
// private UserDetailsServiceImpl userDetailsService;  
  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 return http  
 .authorizeHttpRequests(request -> request  
// .requestMatchers("/public/\*\*").permitAll()  
 .requestMatchers("/journal/\*\*", "/user/\*\*").authenticated()  
 .requestMatchers("/admin/\*\*").hasRole("ADMIN") //Role Based Authentication  
// .anyRequest().authenticated())  
 .anyRequest().permitAll())  
 .httpBasic(Customizer.*withDefaults*())  
 .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.*STATELESS*))  
 .csrf(AbstractHttpConfigurer::disable)  
 .build();  
 }  
  
 @Bean  
 public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {  
 return config.getAuthenticationManager();  
 }  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
}

**We can run the Test class according to the ActiveProfiles(“dev”) like that.**

package com.jspiders.journalApp.service;  
  
import com.jspiders.journalApp.entity.User;  
import com.jspiders.journalApp.repository.UserRepository;  
import org.junit.jupiter.api.Assertions;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.ArgumentMatchers;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.MockitoAnnotations;

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

import org.springframework.test.context.ActiveProfiles;  
import java.util.ArrayList;  
import static org.mockito.Mockito.*when*;

@ActiveProfiles("dev") //so developement profile will be used if we write prod then production will.  
public class UserDetailsServiceImplTests{  
  
 @InjectMocks //It is used to inject all the dependencies and it will search for Mock annotated dependency.  
 private UserDetailsServiceImpl userDetailsService;  
  
 @Mock  
 private UserRepository userRepository; //It will be inject to @InjectMocks annotated dependency.  
  
 @BeforeEach  
 void setUp()  
 {  
 MockitoAnnotations.*initMocks*(this); //Here we are initializing all the mocks before executing any Test  
 }  
  
 @Test  
 void loadUserByUsernameTest() {  
 *when*(userRepository.findByUserName(ArgumentMatchers.*anyString*())).thenReturn(User.*builder*().userName("ram").password("inrkiinrik").roles(new ArrayList<>()).build());  
 UserDetails user = userDetailsService.loadUserByUsername("ram");  
 Assertions.*assertNotNull*(user);  
 }  
}

**Sometime we need to check whether which Profile Environment is running we will do like this:**

package com.jspiders.journalApp;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.ConfigurableApplicationContext;  
import org.springframework.context.annotation.Bean;  
import org.springframework.core.env.ConfigurableEnvironment;  
import org.springframework.data.mongodb.MongoDatabaseFactory;  
import org.springframework.data.mongodb.MongoTransactionManager;  
import org.springframework.transaction.PlatformTransactionManager;  
import org.springframework.transaction.annotation.EnableTransactionManagement;  
  
@SpringBootApplication  
@EnableTransactionManagement  
public class JournalApplication {  
  
 public static void main(String[] args) {  
 ConfigurableApplicationContext context = SpringApplication.*run*(JournalApplication.class, args);  
// System.out.println(context.getEnvironment());  
 ConfigurableEnvironment environment = context.getEnvironment();  
 System.*out*.println(environment.getActiveProfiles()[0]);  
 }  
  
 @Bean  
 public PlatformTransactionManager add(MongoDatabaseFactory dbFactory){  
 return new MongoTransactionManager(dbFactory);  
 }  
}

In application.yml file we are writing, spring: profiles: active: dev means we are writing profiles it is plural it contains multiple profile at a time by using comma separated. So we can do like this:

**application.yml**

spring:  
 profiles:  
 active: dev, prod

**application-prod.yml**

server:  
 port: 8081  
 servlet:  
 context-path: /journal

**application-dev.yml**

spring:  
 data:  
 mongodb:  
 uri: mongodb+srv://rr7691435:Nu2mDuCOVF7abc9b@cluster0.tcnqqjo.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0  
 database: journaldb  
 auto-index-creation: true