What is Spring Boot?

Spring Boot makes developers work simple and we can develop production ready projects.

Spring Boot Advantages:

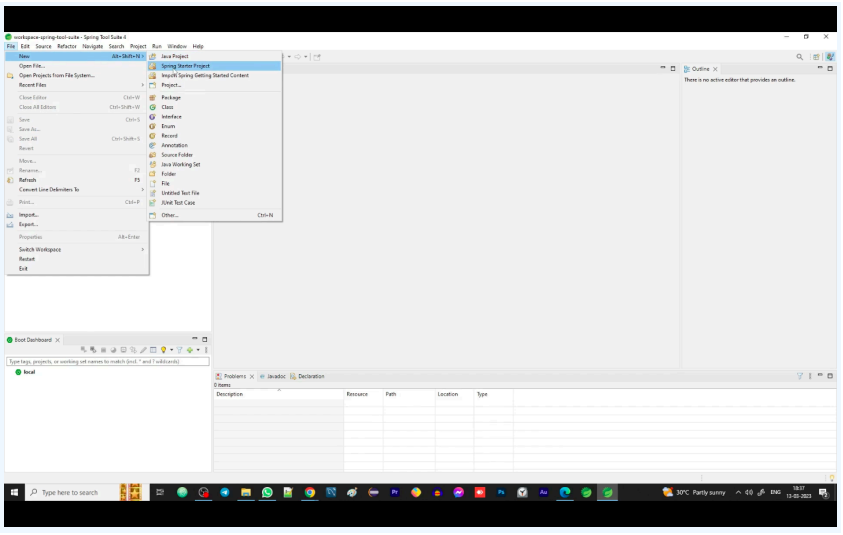
1) Auto Configurations : Spring automatically perform configurations
2) Spring Boot Starters : Spring boot provides different starter dependencies. Each starter dependency will automatically adds all the required dependencies that are related to a module or functionality to the project
3) Spring boot allows us to embed required server dependency with which we will get a server to run our project
4) CLI : Spring boot supports CLI (Command Line Interface) which allows us to run our application in command line interface
5) Actuators : We can check application metrics with actuators that are provided by spring boot

Software Setup :

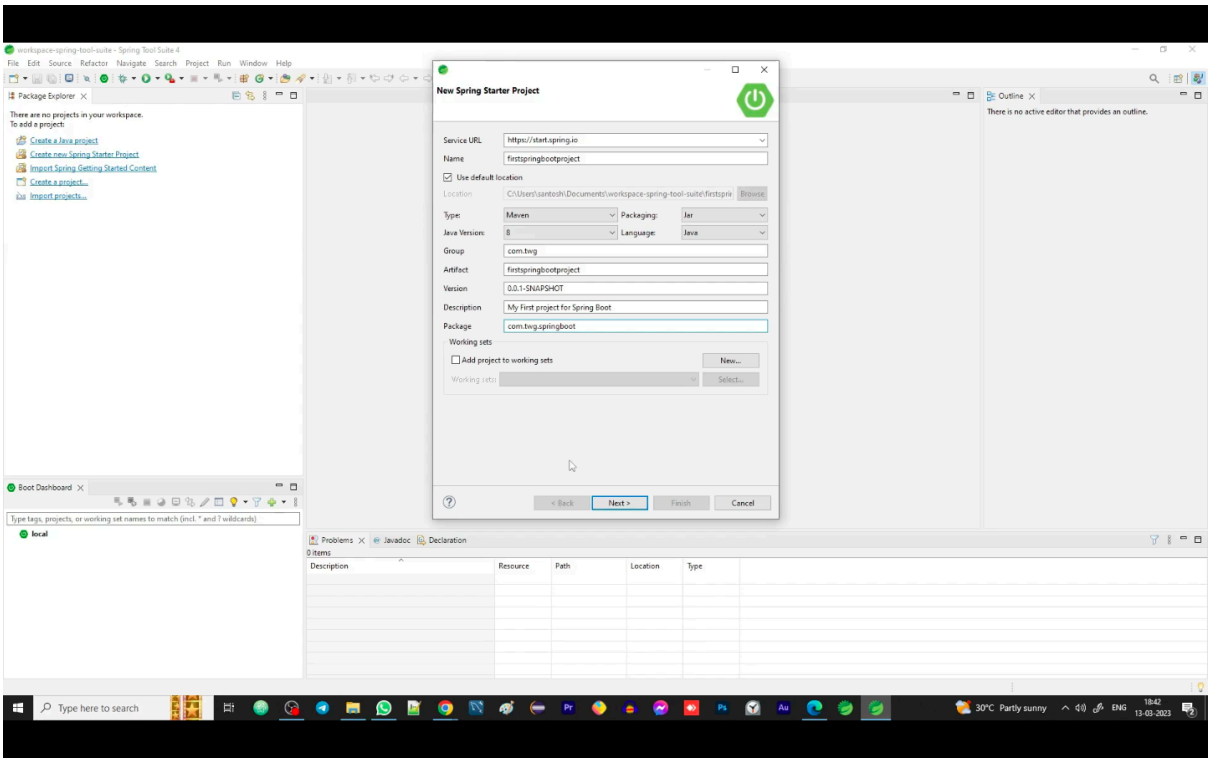
1) Please install jdk software
2) Install spring tool suite (STS) which is a dedicated IDE used to develop spring boot projects
3) Install required database software like oracle / mysql etc.,
4) Install Postman tool that is used to test rest api

Creating our First Spring Boot Project :

Open STS IDE and Create our first project by clicking on File -> New -> Spring Starter Project

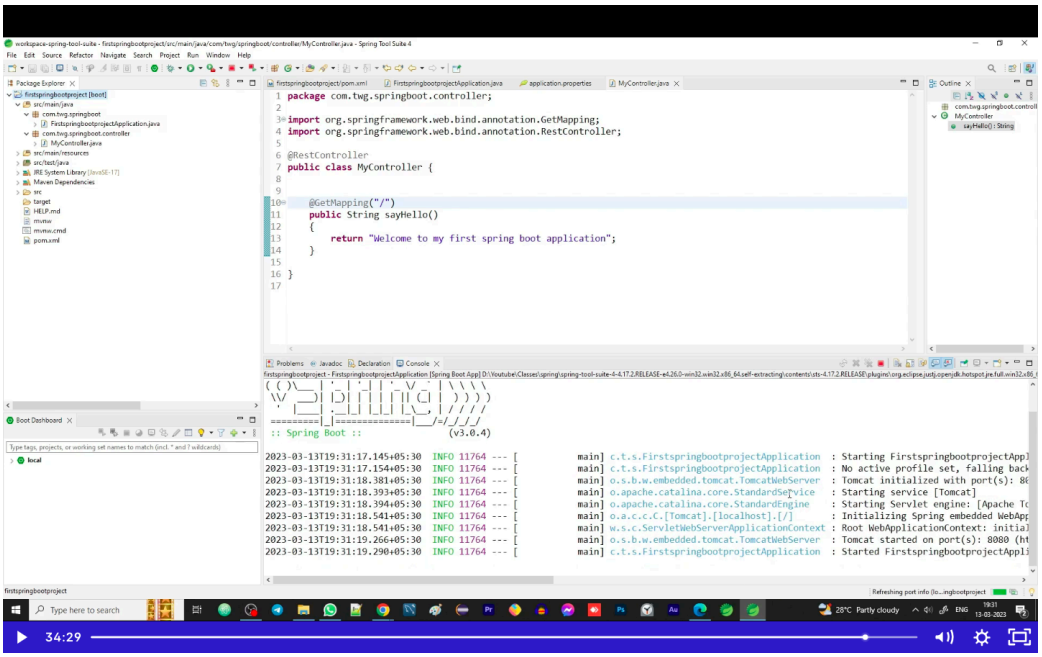


Fill all the required details like package, artifact and select required versions of jdk and click next

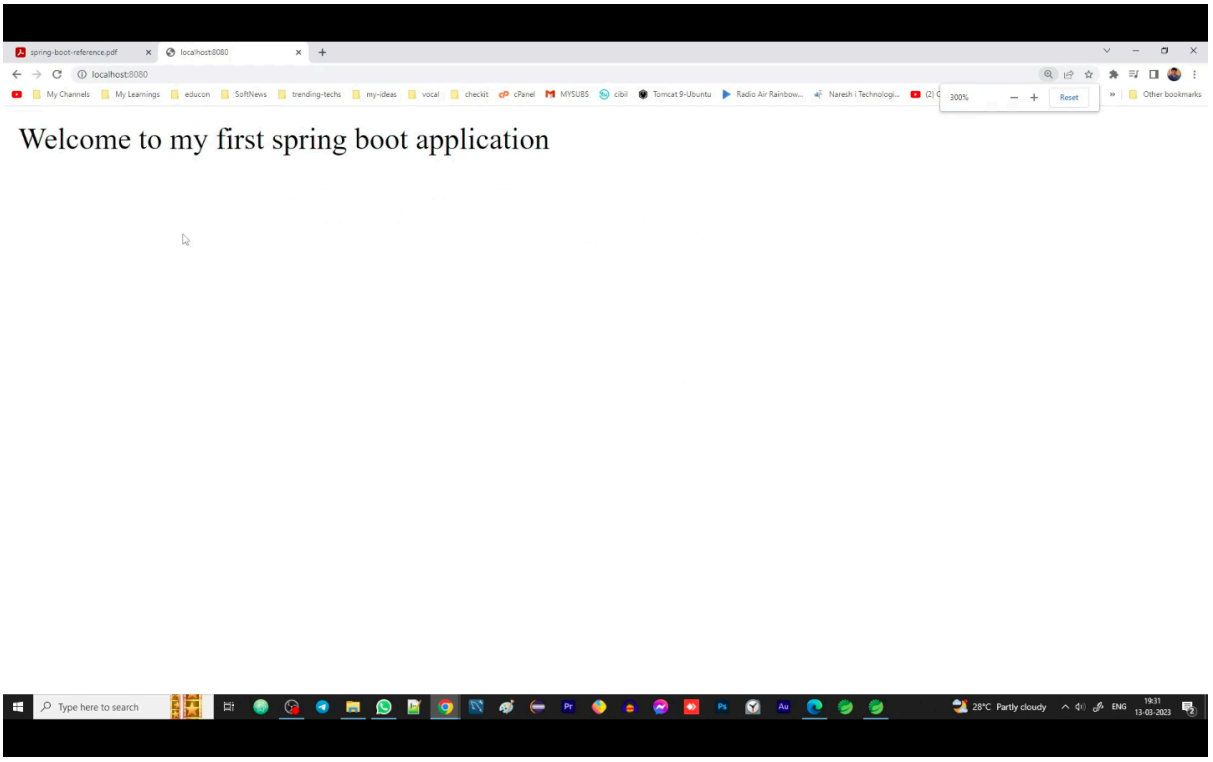


Select required dependencies in the next screen and create the project.

Create a Controller class under controllers sub package as shown below and run project



Once we run our project, we will get the output as shown below.



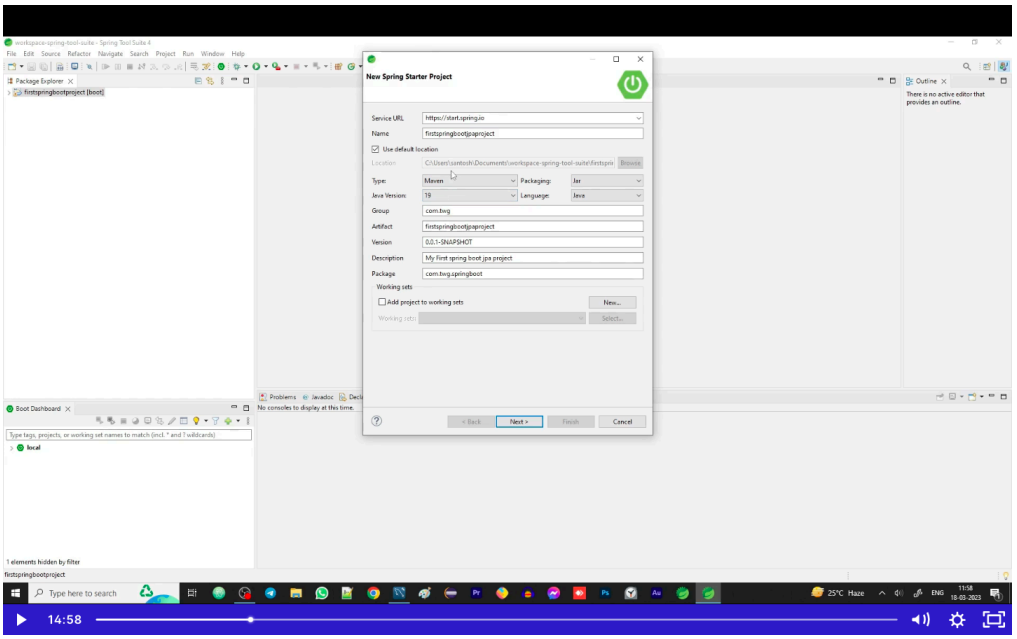
Developing Web Applications using Spring Boot:

The following are the steps to develop web applications using spring boot

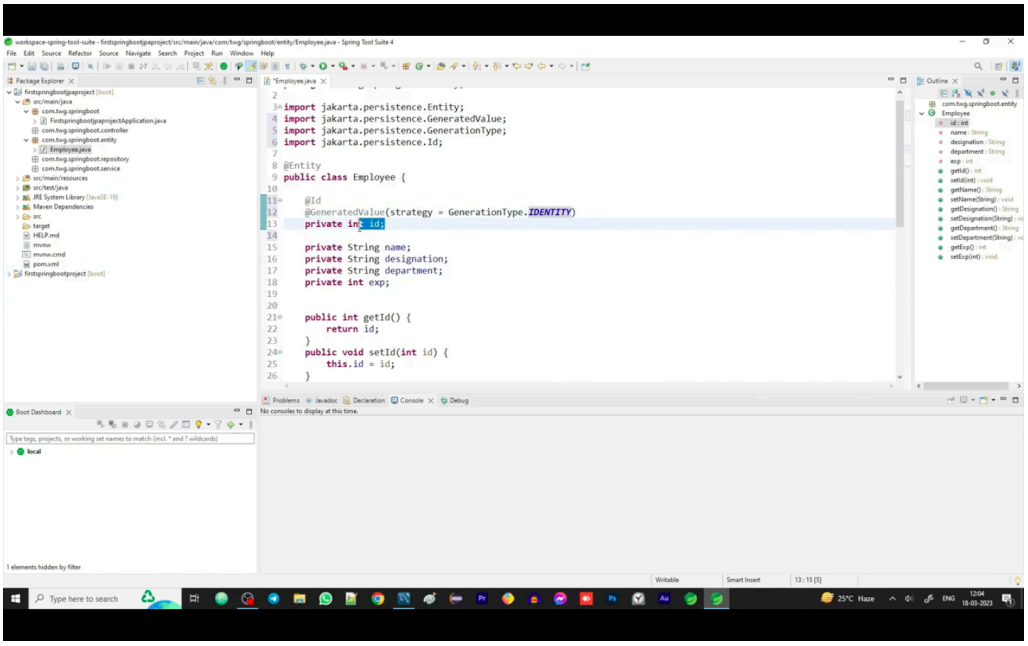
1) Create a Spring Boot Project with required dependencies.
2) Develop models, views and controllers
3) Configure settings in application.properties file

Here, developers need not develop the entire DAO layer. Instead, it is just enough to create an interface that is inherited from JpaRepository

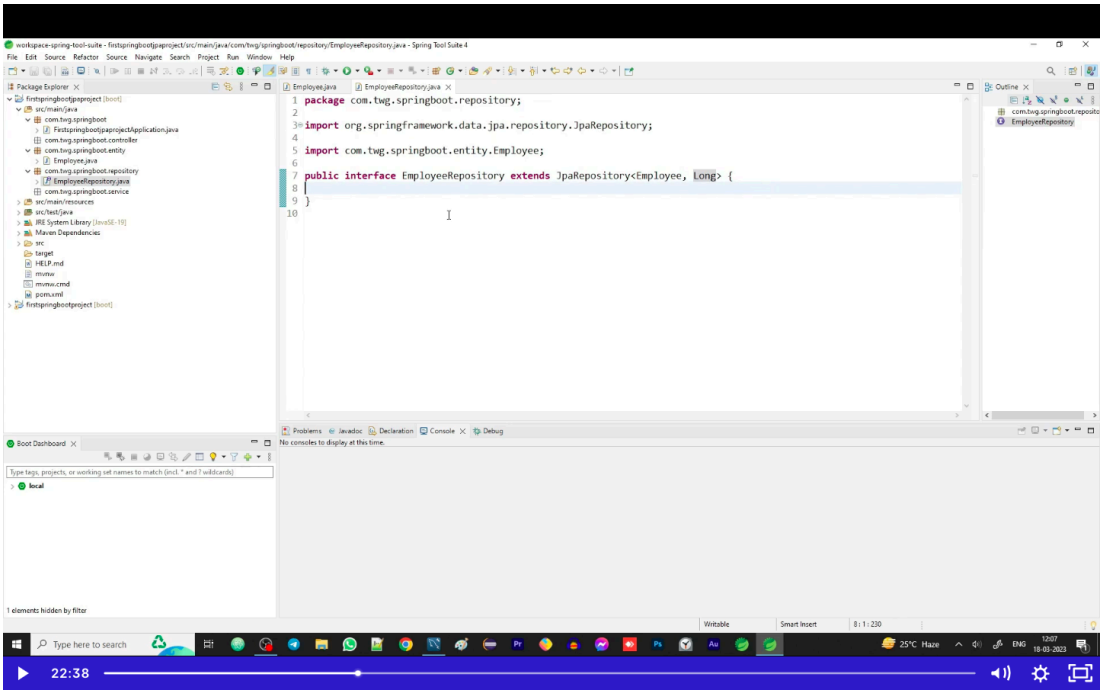
Let us First create a project with required dependencies



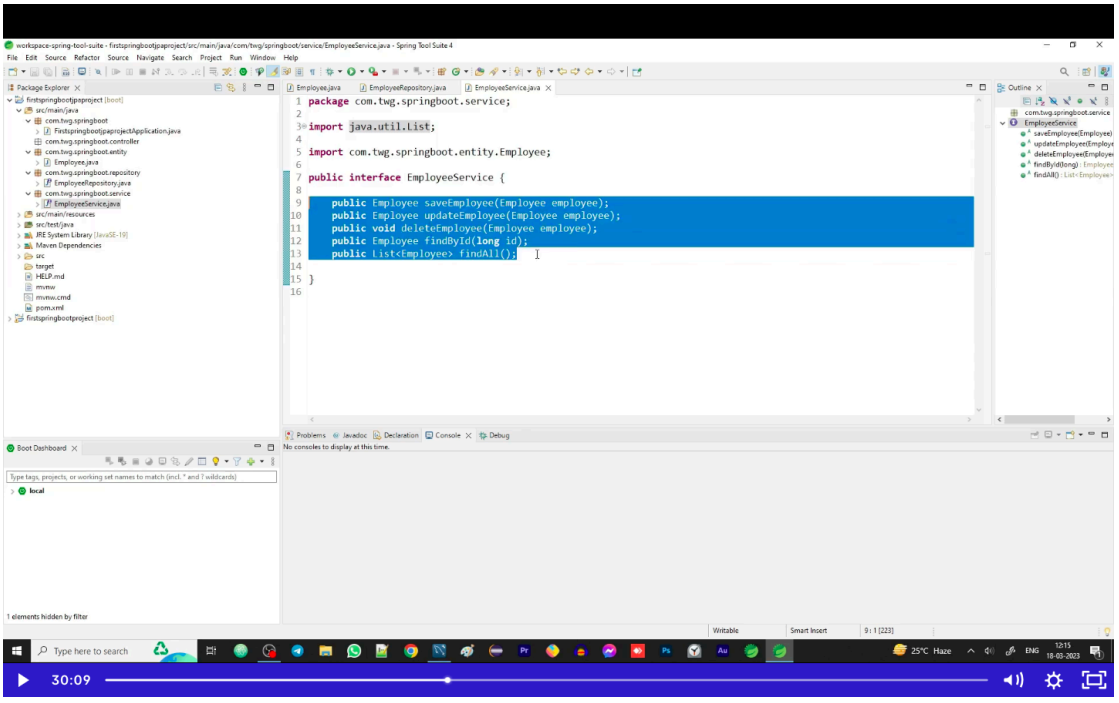
Create an entity named “Employee” under entities sub package as shown below

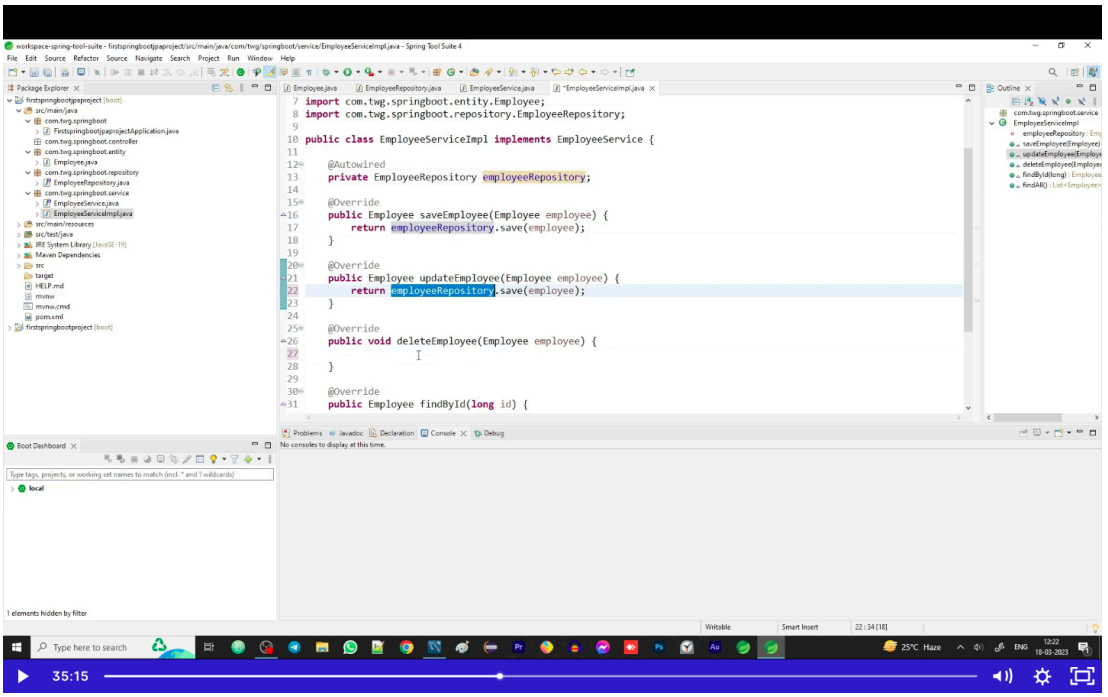


Create Repository interface

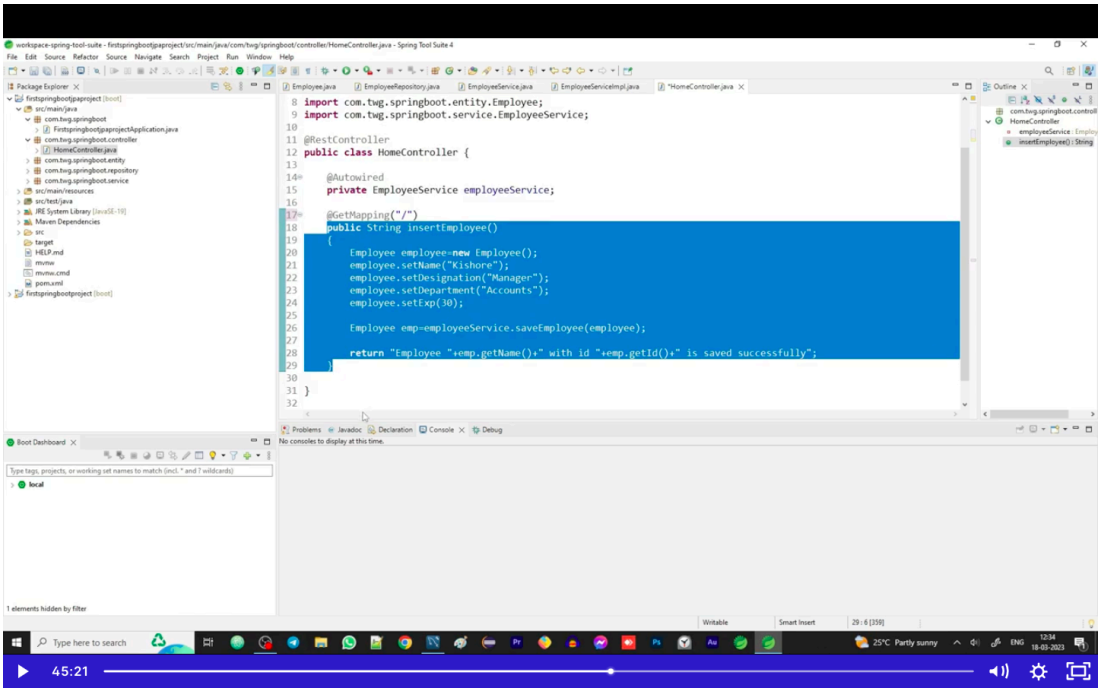


Now create Service layer (interfaces and implemented classes)

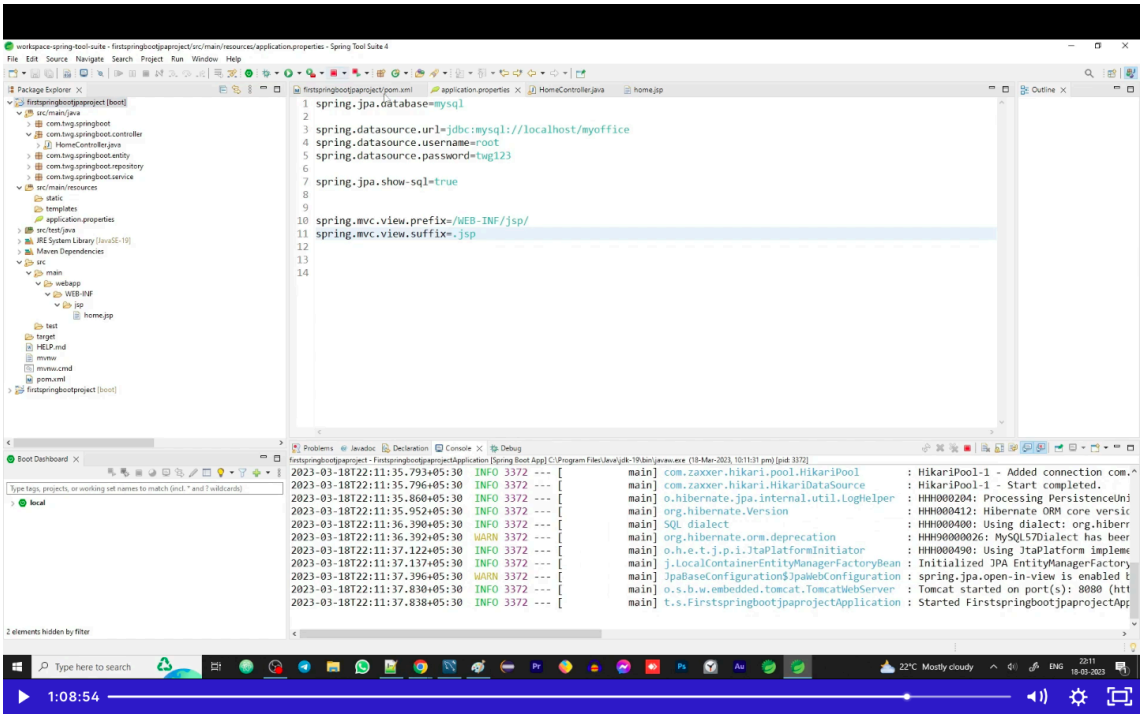


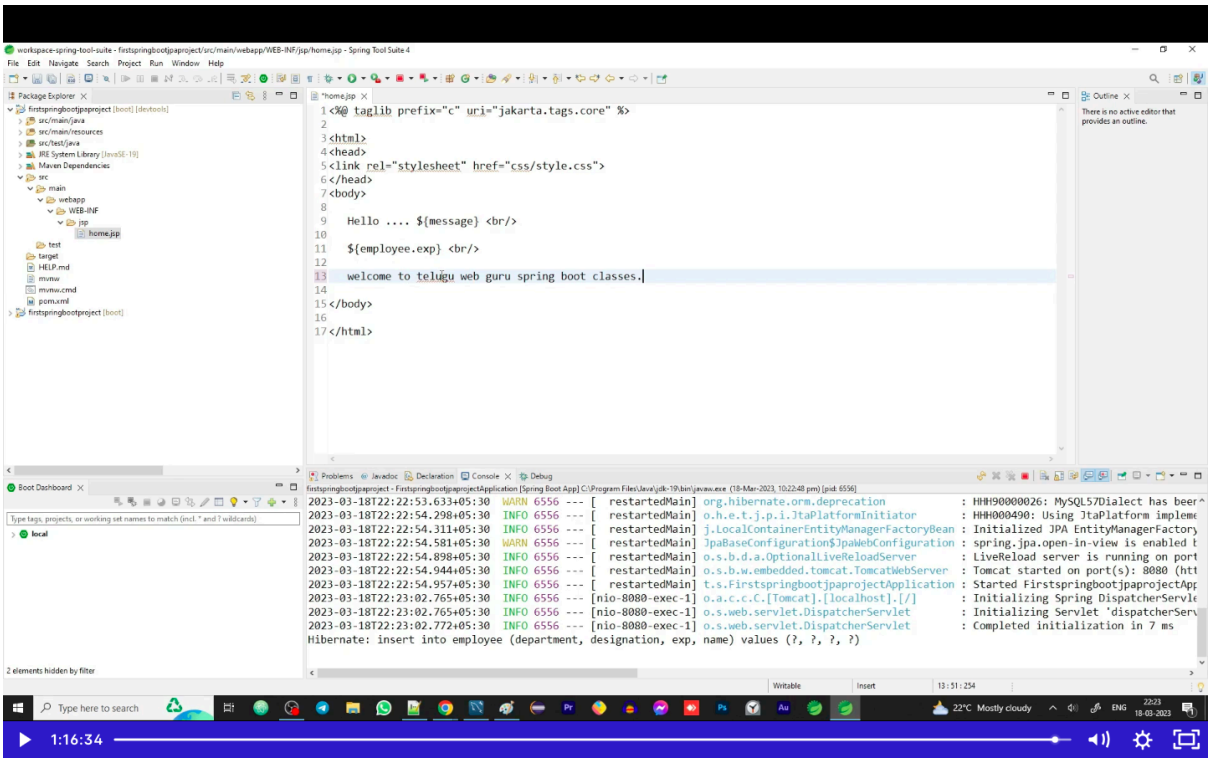


Create controller class with configuring required end points.



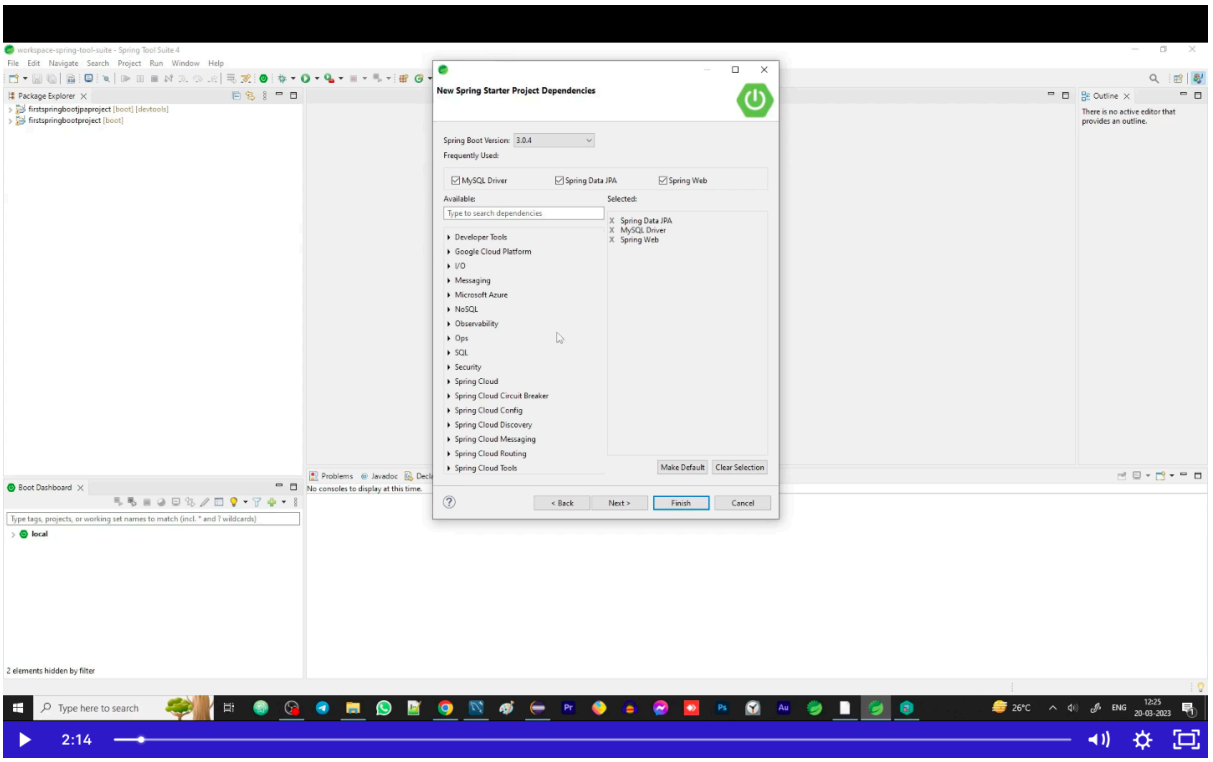
Configure database settings in application.properties


spring.jpa.database=mysql
spring.datasource.url=jdbc:mysql://localhost/myoffice
spring.datasource.username=root
spring.datasource.password=twg123
spring.jpa.show-sql=true
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
1:08:54

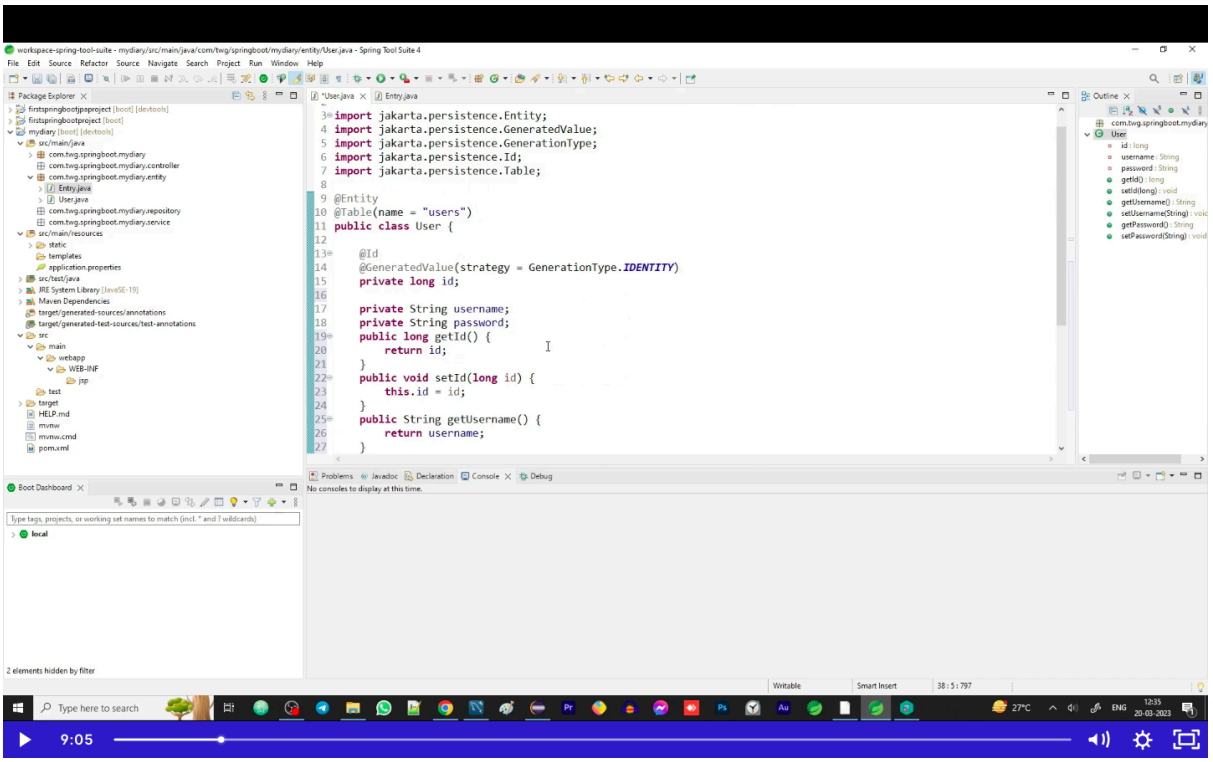

<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
<head>
<link rel="stylesheet" href="css/style.css">
</head>
<body>
Hello .... ${message}
${employee.exp}
welcome to teluğu web guru spring boot classes.
</body>
</html>
1:16:34

Implementing MyDiary app using Spring Boot:

First let us create a new project in STS IDE with required dependencies



Create entity classes for User and Entry entities with required annotations. Following pic shows User entity as an example



Create repository and service layers as we created in our first spring boot app.

We will copy all the controller methods from our spring mvc mydiary app.

Replace all request mapping annotations with corresponding methods based on methods like GetMapping, PostMapping and so on.

Project's source code is added in the portal for your reference. Please practise on your own so that you will gain more practical knowledge.

JSON Tutorial :

JSON stands for JavaScript Object Notation.

This format will be used in developing Rest API. JSON is very helpful to create an environment where different technology based apps communicate with each other.

Initially it is developed for javascript objects. But due to its simplicity, it is extensively used by multiple technologies as it represents data in simple text.

Example:

```
{
   “name” : “teluguwebguru”,
   “category”: “Software Training”
}
```

While giving names we must use double quotes

While giving values, double quotes are required for string type values.

What is API ?

API stands for Application Programming Interface. Here Application means software.

API are mechanisms that enable software components to communicate with each other using a set of definitions and protocols

Different type of APIs:

Private API: These are internal to enterprise and only used for connecting systems and data within enterprise or business

Public API : These are open to the public and may be used by anyone

Partner API : These are only accessible by external developers to aid business to business partnerships

Composite APIs : These combine different APIs to address complex system requirements or behaviour

What is REST API ?

These APIs are called remote procedure calls. REST has quickly become the de facto standard for building web services on the web because REST services are easy to build and easy to consume.

By building on top of HTTP, REST APIs provide the means to build:

- Backwards compatible APIs
- Evolvable APIs
- Scalable services
- Securable services
- A spectrum of stateless to stateful services

Creating REST API using Spring Boot:

Steps to create REST API in Spring Boot:

1) Create a new project with required dependencies related to Spring Web, Spring Data Jpa and Database
2) Creating entities
3) Create a Repository interface that is extended from JpaRepository
4) Create Service Layer (interfaces and implementation classes)
5) Create Controller class
6) Creating views
7) Configuring application.properties

REST API for POST Requests: POST request is used to save objects into database

To create REST API for post requests we need to use the annotation called @PostMapping

```
@PostMapping
public Entry saveEntry(Entry entry)
{

        // code to save this particular entity into database with the help of service
          Object


 }
```

REST API for PUT Requests: PUT requests are used to update database objects

To create REST API for put requests we need to use the annotation called @PutMapping

```
@PutMapping
public Entry saveEntry(Entry entry)
{
        // code to save this particular entity into database with the help of service
          Object

 }
```

REST API for GET Requests: GET requests are used to update database objects

To create REST API for Get requests we need to use the annotation called @GetMapping

```
@GetMapping(“/entries/{id}”)
public Entry saveEntry(@PathVariable(“id”) int id)
{
        // code to get this particular entity from database
 }
```

REST API for Delete Requests: Delete requests are used to update database objects

To create REST API for Delete requests we need to use the annotation called @DeleteMapping

```
@DeleteMapping(“/entries/{id}”)
public Entry saveEntry(@PathVariable(“id”) int id)
{
        // code to get this particular entity from database and delete it
 }
```

To create REST API for Update requests we need to use the annotation called @PutMapping

```
@PutMapping(“/entries/{id}”)
public Entry saveEntry(@PathVariable(“id”) int id, @RequestBody Entry entry)
{
        // code to get this particular entity from database and update  it
 }
```

To create a REST API for Patch requests we need to use the annotation called @PatchMapping. Patch method allows us to update selected field in an object.

```
@PatchMapping("/entries/{id}")
public Entry saveEntry(@PathVariable("id") int id, @RequestBody Entry entry)
{
        // code to get this particular entity from database and update  it
 }
```

Spring Boot Annotations:

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

   `@EnableAutoConfiguration`: enable Spring Boot's auto-configuration mechanism

   `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)

   `@Configuration`: allow to register extra beans in the context or import additional configuration classes

2) **`@Configuration`** is a class-level annotation indicating that an object is a source of bean definitions. `@Configuration` classes declare beans through @Bean-annotated methods. Calls to `@Bean` methods on `@Configuration` classes can also be used to define inter-bean dependencies.
3) **@PathVariable** Annotation which indicates that a method parameter should be bound to a URI template variable.
4) **@RequestBody** You can use the `@RequestBody` annotation to have the request body read and deserialized into an `Object` through an HttpMessageReader.
5) **@Autowired** Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection facilities.
6) **@RestController** A convenience annotation that is itself annotated with `@Controller` and `@ResponseBody`.

   Types that carry this annotation are treated as controllers where `@RequestMapping` methods assume `@ResponseBody` semantics by default.

NOTE: `@RestController` is processed if an appropriate `HandlerMapping-HandlerAdapter` pair is configured such as the `RequestMappingHandlerMapping-RequestMappingHandlerAdapter` pair which are the default in the MVC Java config and the MVC namespace.

7) **@Bean** The @Bean annotation is used to indicate that a method instantiates, configures, and initializes a new object to be managed by the Spring IoC container. For those familiar with Spring's <beans/> XML configuration, the @Bean annotation plays the same role as the <bean/> element.
8) **@EnableAutoConfiguration** Enable auto-configuration of the Spring Application Context, attempting to guess and configure beans that you are likely to need. Auto-configuration classes are usually applied based on your classpath and what beans you have defined. For example, if you have `tomcat-embedded.jar` on your classpath you are likely to want a `TomcatServletWebServerFactory` (unless you have defined your own `ServletWebServerFactory` bean).

   When using `@SpringBootApplication`, the auto-configuration of the context is automatically enabled and adding this annotation has therefore no additional effect.

9) **@Component** Indicates that the annotated class is a *component*.

   Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

10) **@Repository** Indicates that an annotated class is a "Repository", originally defined by Domain-Driven Design (Evans, 2003) as "a mechanism for encapsulating storage, retrieval, and search behaviour which emulates a collection of objects".

    Teams implementing traditional Jakarta EE patterns such as "Data Access Object" may also apply this stereotype to DAO classes, though care should be taken to understand the distinction between Data Access Object and DDD-style repositories before doing so. This annotation is a general-purpose stereotype and individual teams may narrow their semantics and use as appropriate.

microservices :

Microservices are a modern approach to software whereby application code is delivered in small, manageable pieces, independent of others.

Why build microservices?

Their small scale and relative isolation can lead to many additional benefits, such as easier maintenance, improved productivity, greater fault tolerance, better business alignment, and more.

Microservices will solve the problems that occur with monolith architectures. Those are

1) Monolith architecture maintains single and large code base
2) It is tightly coupled
3) Unable to scale specific modules
4) Unable to use multiple dependencies
5) Deployment process delayed on updates
6) Bug in a single module leads to shutdown of entire project

Microservices addresses all the above said issues with its advanced level architecture.

Microservices make these applications Loosely coupled. With this, we can develop, deploy and scale the applications independently.

Microservices Criteria :

On what basis total application is divided into microservices?

1) Always criteria should be Business functionality.

How microservices communicate with each other?

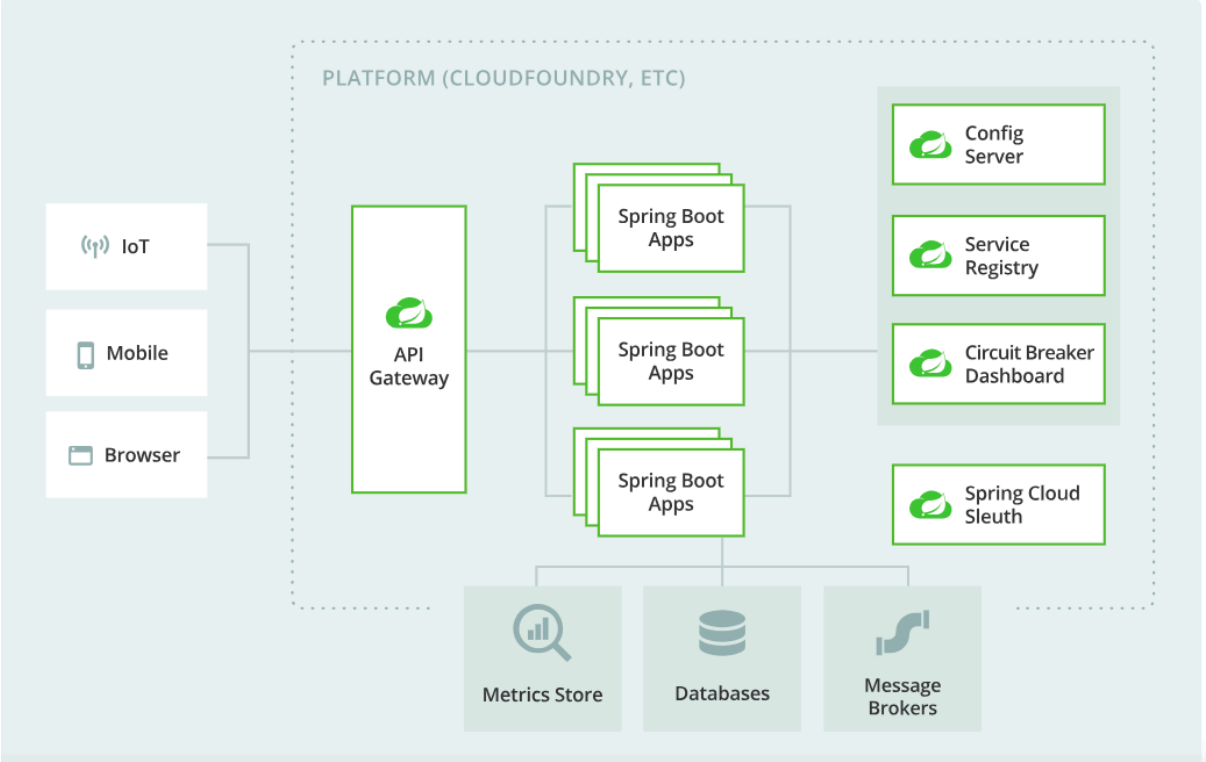
Microservices will communicate in two ways

1) Synchronous : request and response will be processed immediately. So upon receiving response, sending microservice will send next request

2) Asynchronous: requesting microservice will not wait for response from the other end.

Microservice Architecture :

Following figure shows microservices architecture.



API Gateway : is an intelligent and programmable router based on Spring Framework and Spring Boot.

Config Server : This is used to maintain all configurations at central level

Circuit Breaker Dashboard : If any of the microservice failed, then this module controls the frequency of requests to this failed one from other microservices

Service Registry : It maintains the details of multiple instances of microservices.

Spring cloud Sleuth : This is used for tracing. How a request is traversing through various microservices, everything is maintained by this sleuth

Metrics Store : Used to maintain in-depth metrics

Message Brokers : Communication between microservices are established through these message brokers.

Spring Cloud :

To implement microservices architecture, we may need to depend on multiple technologies because it contains multiple modules like service registry, sleuth, gateway etc., Instead, we can implement all these things by using spring cloud.

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems

Features

Spring Cloud focuses on providing a good out of box experience for typical use cases and extensibility mechanisms to cover others.

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Distributed messaging
- Short lived microservices (tasks)
- Consumer-driven and producer-driven contract testing

Service Registry, Discovery using netflix eureka server:

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon).

Steps to implement Eureka Server:

1) Create eureka server spring boot application by including required eureka server dependencies
   This eureka server runs at port 8761 by default
2) Enable eureka server functionality with the help of annotation @EnableEurekaServer
3) Add Eureka client dependencies in microservices apps
4) Set configurations in application.properties file
   eureka.client.register-with-eureka=false
   eureka.client.fetch-registry=false
5) Set names to client microservices by using spring.application.name property

RestClient implementation using Spring Cloud OpenFeign :

For implementing RestClient, we use OpenFeign instead of RestTemplate class. This will simplify the implementation process.

Steps:

1) Add openfeign dependency
2) Create an interface with @FeignClient annotation
3) Add required methods

API Gateway:

Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

Spring Cloud Gateway features:

- Built on Spring Framework and Spring Boot
- Able to match routes on any request attribute.
- Predicates and filters are specific to routes.
- Circuit Breaker integration.
- Spring Cloud DiscoveryClient integration
- Easy to write Predicates and Filters
- Request Rate Limiting
- Path Rewriting

Steps to implement:

1) Create an application for API Gateway
2) Register this application with eureka server
3) Configure settings in application.properties

```
spring.cloud.gateway.routes[0].id=
spring.cloud.gateway.routes[0].uri=
spring.cloud.gateway.routes[0].predicates[0]=
```

Set above set of configuration settings for each and every service.

Now after configuring, client request will be forwarded to microservices through above API Gateway.