**Microservices Testing using Spring Boot+JUnit5+Mockito**

Structure of Spring Boot Application

A diagram of a software system

Description automatically generated

1. **Monolithic and Microservices Advantages and Disadvantages?**

**Monolithic:**

**Monolithic Application Advantages:**

* Low complexity
* Easy to develop test and deploy.

**Monolithic Application Disadvantages:**

* Tightly coupled (Rigid)
* Hard to scale this application
* Slow performance
* Single point of failure
* Slow continuous development.

**Microservices:**

**Microservices Application Advantages:**

* Loosely coupled
* Agility and flexibility
* Independent development and deployment
* Falt Isolation (We can modify the specific component without touching the rest of the component)
* Mixed technology stack

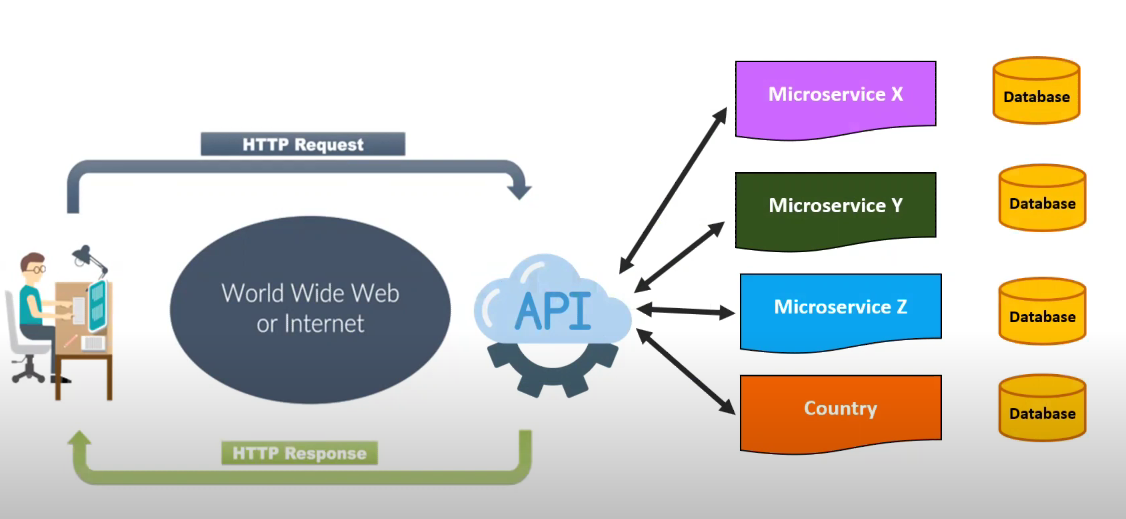
**Microservices Application disadvantages:**

* High complexity
* Consistency issues (We difficult to getting the response from one component database to another)
* Automation (We have a greater number of services, databases, we are difficult to find out all the services)
* Debugging. (Compare to monolithic very difficult to debugging in microservices)
* Complexity to add the security features.

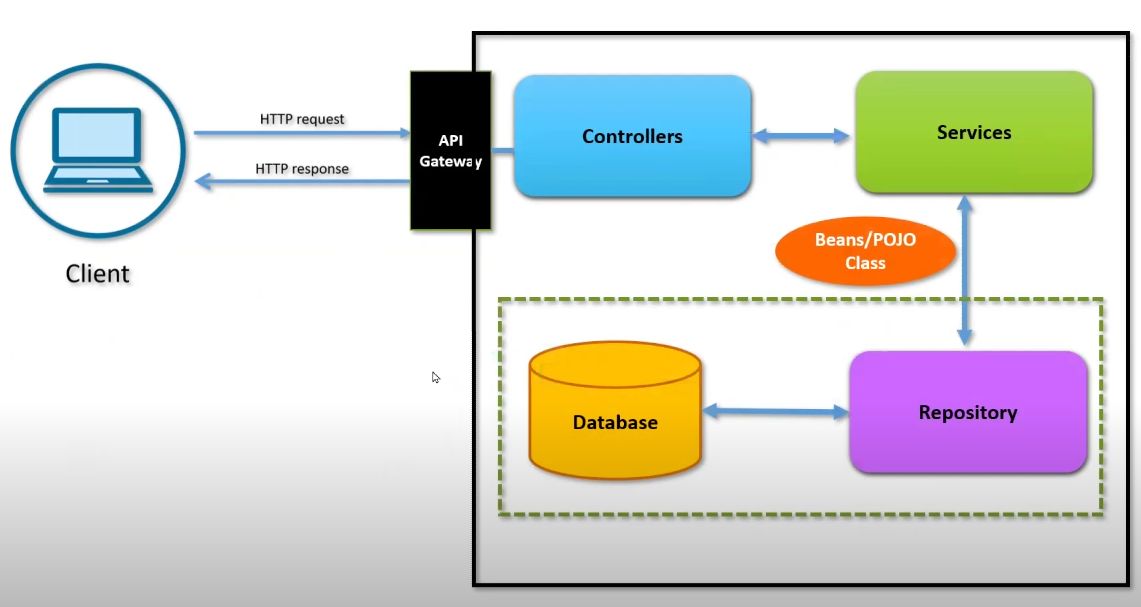
So, each approaches having their advantages and disadvantages, but companies are following both approaches even if its small company or small project a smaller number of team members then go with the monolithic approach.

Suppose our application is very huge like amazon or flipchart there are so many modules and services need to be run, its very complex project keep on add a new feature as per the market in our project in these cases go with microservices approach.

**Developing our Microservices application:**



Once we develop the microservice then we can access those services by using API endpoints the flow diagram below:



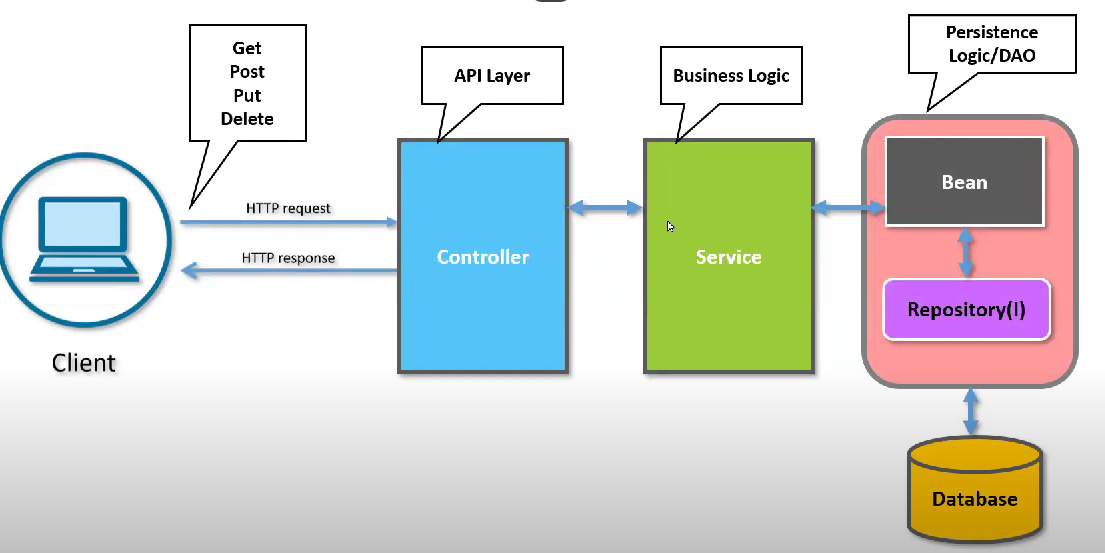
**Unit Testing:**

* Unit testing means testing the single component or method by passing different types of inputs, we can see how the results are coming weather is expected or not.
* We conduct the unit testing either controller methods or service layer.
* When we do the unit testing in the service level this comes from under the microservices testing. All the microservices placed into this layer.
* We can also directly test the controller methods. It directly talks to the services. This kind of testing we can do only unit testing. Because we need to test directly to the methods.

We are going to test the service by directly invoking methods which are available in the service through unit testing and after that we can also test the controller methods, and this can be done by using different types of frameworks which are available below.

1. **Junit** – It is most popular unit testing framework; it Is a fundamental java-based unit testing framework. All the developers will use to perform the unit testing. It is a basic framework which we write a unit test case**.**
2. **Mockito and MockMVC** – These are the two important framework which are available to test the microservices and APIs**.**

* **Mockito** –The advantage of Mockito is we can Mocking the **external dependencies**. Here the external dependencies are DAO or Repository layer and database layer, we are mock the entire external dependencies using Mockito framework.
* Mockito is a general framework anybody can use for java related applications not only for spring boot applications.
* Using Mockito framework to mock the external dependencies for the service layer as well as the controller layer.
* Use to invoke the controller’s methods directly, here MockMVC is not required when we are using Mockito.
* **MockMVC** – it is a build in framework which is available in the spring boot application.
* Using MockMVC framework invoking the controller method without starting the spring boot server.
* If we want to invoke the controller methods directly, we are using MockMVC framework. Without starting the- spring boot server we still want to invoke the controller methods. In this case we go with the MockMVC framework.
* We can use MockMVC along with the Mockito also.
* **RestTemplate** – Whenever testing methods on the service layer or controller layer this comes under unit testing.
* If we want to perform any integration testing, then we can use RestTemplate.
* **H2 Database** – Suppose in the entire project structure the controller talks to the service -> service again talk to the database through DAO or repository. Suppose If the DB is not available or I don’t want to use real time database which is available in the production, but I can mock the database using H2 database.

****

The above diagram based on that we developed country service spring boot application. Testing the same application using the JUnit and Mockito framework.

* For Controller class -> Service layer is the external dependency
* For service layer -> DAO or Repository and Database is the external dependency.

1. **Test the service layer using Mockito framework:**

* Without starting a spring boot server testing a service layer.
* @SpringBootTest(classes = {SpringBootCountryServiceApplication.class})
* This annotation is help us to explicitly specifying spring boot test class.
* @Mock – Mocking the external dependency.
* @InjectMocks – Whatever the methods we have in the countryService layer, if we invoke all these methods we must use this annotation.
* @Test – We have use in methods, Junit will execute this method. The package we import for @Test annotation is (import org.junit.jupiter.api.Test;)
* @Order(1) – Which method we need to execute first in that method we use this annotation.
* Whatever external methods we are calling in the service we must specify that external method findAll () then return our own data.
* *when*(countryDbRepository.findAll()).thenReturn(mycountries);
* This is a mocking statement purpose we are using this
* The above **when** and **thenReturn** we are using when the method is returning something.
* If the method is not return anything (void method) we are not using this above statement.
* *assertEquals*(3, countryDbService.getAllCountries().size());
* This will return how many countries we have that we need to validate using Junit **assertEquals** method.
* Before executing a test method, we must configure Junit 5 test.
* @TestMethodOrder(MethodOrderer.OrderAnnotation.class)
* This is a class level annotation Once we add this annotation @Order Annotation will work, otherwise by default it gives an alphabetical order.

1. **Test the controller layer using Mockito framework:**

* Basically, this controller methods are internally invoking the service method. While testing the controller method we Moke the service methods. Because the controller method the external dependencies is service layer.
* Whatever methods are available in the service layer we have to Moke those service methods inside the controller unit test cases.
* How we can use Mockito framework by testing controller methods.
* Unit test case will be created src/test/java -> In this package create new test class and write out all the scenarios.

1. **MockMvc along with the Mockito framework:**

* Testing the controller methods by passing the URLs or Http request. But when using the URL we must start the spring boot server, without starting the server test using URLs that is possible by using MockMvc framework.
* Before calling every controller method we need to do some kind of setup that use to help of mockMvc builder.
* @Autowired  
  MockMvc mockMvc;
* In this above object we can get or access one method that is called standalone setup. In this setup we have built the country controller method. For that we have write one Junit method below
* import org.junit.jupiter.api.BeforeEach;

@BeforeEach

public void setUp() {

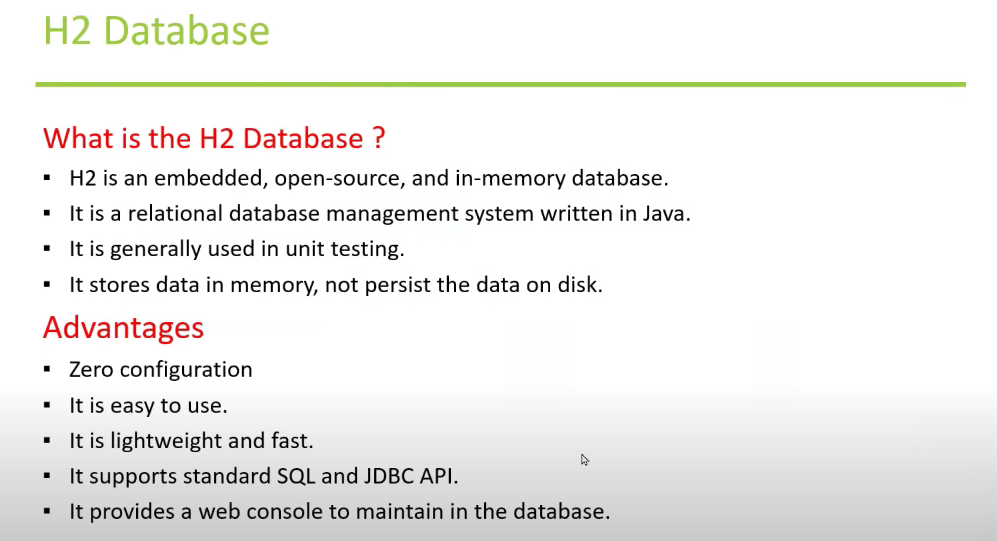
this.mockMvc = MockMvcBuilders.*standaloneSetup*(countryDbController).build();

}

* This method we want to execute before every test method started.
* MockMvcBuilders - This is a pre-defined class inside this we have a standaloneSetup
* this.mockMvc.perform(*get*("/getAllDbCountries"))  
   .andExpect(*status*().isFound())  
   .andDo(*print*());
* This is the way we are sending the controller request. Based upon the type of request controller method will get execute.
* ObjectMapper – this is a special class it will help us to convert java object into json object.

**H2 Database (Mock Database for Unit/Integration Testing):**

* H2 database is a mock database for unit and integration testing. When we testing the APIs are microservices we can use the h2 database as a mock database instead of using the real database.
* Now, we are mock the database. It means when we sent a request in the controller method talk to the service method service will talk repo and repo talk to the database.

****

This is the query we have to use when we are fetching the data from this file in runtime.CREATE TABLE "COUNTRY\_TABLE" (  
 "ID" INT AUTO\_INCREMENT PRIMARY KEY,  
 "COUNTRY\_NAME" VARCHAR(255),  
 "COUNTRY\_CAPITAL" VARCHAR(255)  
);  
  
INSERT INTO "COUNTRY\_TABLE" ("COUNTRY\_NAME", "COUNTRY\_CAPITAL") VALUES ('India', 'New Delhi');  
INSERT INTO "COUNTRY\_TABLE" ("COUNTRY\_NAME", "COUNTRY\_CAPITAL") VALUES ('United States', 'Washington DC');

In application.properties use this below for connecting an h2 database

#H2 database  
spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=  
spring.h2.console.enabled=true  
spring.jpa.show-sql=true  
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
spring.jpa.hibernate.ddl-auto=update

This is dependency we have use:  
<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
</dependency>

**Rest Template in Spring Boot:**

* Rest template is used to create applications that consume RESTful web services.
* We can do integration testing of Rest APIs with database.

How we can use rest template along with the H2 database to perform integration testing on microservices.

* Here we are creating the H2 database purpose of doing a unit or integration testing
* Now we are writing the integration testing through HTTP requests. Because weather HTTP request when we send this request properly going to the database or not, database properly giving the response or not. Integration between the REST API along with the database this scenario we are going to test.
* Why H2 database means, while writing the integration testing, we do not have any external dependencies like databases, because database can access anyone in our team this reason, we do not disturb the real database.
* Integration test cases we write to the controller methods. By using the Rest template writing the integration test cases consuming the Rest Api’s.
* While executing the unit test cases we don’t need to run the spring boot server because we are mocked it, we are not depending on the server. But whenever we are doing the integration testing, we have started the spring boot application server so that the database up and running APIs request able to call through the URL.