**Microservice Infrastructure**

**Project layers –**

**Entire microservice is divided in 4 layers as follows**

* **Controller Layer**

**In this layer exposed endpoints are mapped to respective functions of service layer to perform specific action or fetch data. Returned output of service layer is sent in response with proper status.**

* **Service Layer**

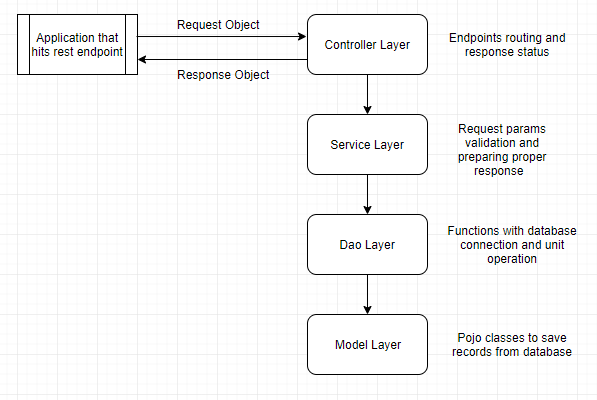
**Here we validate the request data and call multiple Dao functions which perform a single database operation and link their calls together. The output of all the Dao methods is processed and proper object to be passed in response body is returned to controller layer.**

* **Dao Layer (Data access object)**

**This layer contains the functions which has database connections. Simple database operations are done in in these Dao functions. One Dao function may be used in multiple service functions. Dao uses Models i.e. Pojo (Plain old Java Object) to get the database records.**

* **Model**

**Model layer has the all Pojos required to store the database records. This layer contains only Pojo classes.**

****

**Libraries, APIs and Other Technologies Used**

1. **Spring Actuator**

Spring Boot provides actuator to monitor and manage our application. Actuator is a tool which has HTTP endpoints. When application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints. To set the spring actuator you need to follow following steps

* + - * 1. **Add dependency in pom.xml**

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
 <version>${springframework.version}</version>  
</dependency>

* + - * 1. **Configuring Existing Endpoints**

Each endpoint can be customized with properties using the following format: *endpoints.[endpoint name].[property to customize]*

Three properties are available:

* *id –*by which this endpoint will be accessed over HTTP
* *enabled* – if true then it can be accessed otherwise not
* *sensitive* – if true then need the authorization to show crucial information over HTTP

For example, add the following properties will customize the /*beans* endpoint*:*

endpoints:  
 beans:  
 id: springbeans  
 sensitive: false  
 enabled: true

* + - * 1. **Some Examples that we used in Infra Microservice**

endpoints:  
 health:  
 sensitive: false  
 enabled: true  
 metrics:  
 id: springmetrics  
 sensitive: false  
 enabled: true

To get new health information from actuator hit

[*BaseUrl/actuator/health*](localhost:8080/actuator/health)(Base\_Url = localhost:8080)

You will get a Json object as

{

"status":"UP"

}

To get the metrics information from actuator hit

[*BaseUrl/actuator/metrics*](localhost:8080/actuator/metrics)(Base\_Url = localhost:8080)

The metrics endpoint publishes information about OS, JVM as well as application level metrics. Once enabled, we get information such as memory, heap, processors, threads, classes loaded, classes unloaded, thread pools along with some HTTP metrics as well.

Here’s what the output of this endpoint looks like out of the box:

{

"names": [

"jvm.threads.states",

"jdbc.connections.active",

"jvm.memory.used",

"jvm.gc.memory.promoted",

"jvm.memory.max",

"jvm.gc.max.data.size",

"jdbc.connections.max",

"jdbc.connections.min",

"jvm.memory.committed",

"system.cpu.count",

"logback.events",

"http.server.requests",

"tomcat.global.sent",

"jvm.gc.pause",

"jvm.buffer.memory.used",

"tomcat.sessions.created",

"jvm.threads.daemon",

"system.cpu.usage",

"jvm.gc.memory.allocated",

"tomcat.global.request.max",

"hikaricp.connections.idle",

"hikaricp.connections.pending",

"tomcat.global.request",

"tomcat.sessions.expired",

"hikaricp.connections",

"jvm.threads.live",

"jvm.threads.peak",

"tomcat.global.received",

"hikaricp.connections.active",

"hikaricp.connections.creation",

"process.uptime",

"tomcat.sessions.rejected",

"process.cpu.usage",

"tomcat.threads.config.max",

"jvm.classes.loaded",

"hikaricp.connections.max",

"hikaricp.connections.min",

"jvm.classes.unloaded",

"tomcat.global.error",

"tomcat.sessions.active.current",

"tomcat.sessions.alive.max",

"jvm.gc.live.data.size",

"hikaricp.connections.usage",

"tomcat.threads.current",

"hikaricp.connections.timeout",

"jvm.buffer.count",

"jvm.buffer.total.capacity",

"tomcat.sessions.active.max",

"hikaricp.connections.acquire",

"tomcat.threads.busy",

"process.start.time"

]

}

* + - * 1. **Info endpoint**

info:  
 app:  
 name: Spring Actuator Example  
 java:  
 version: 8  
 type: Spring Boot

To get new health information from actuator hit

[*BaseUrl/actuator/info*](localhost:8080/actuator/info) (Base\_Url = localhost:8080)

1. **JPA**

Spring Data JPA provides complete abstraction over the DAO layer. We don’t need to write the implementation for the DAO layer anymore; Spring Data auto-generates the implementation DAO implementations.

We use inbuilt JpaRepository<pojo\_class, id\_type> to get the basic crud operation for table which is mapped to pojo\_class using @Entity and @Table(name = ‘tableName’) annotations.

JpaRepository gives CRUD methods like save(), delete() and other basic functions like findById(), findAll() etc.

To add custom method write own class and extend it from JpaRepository

* + - * 1. We need to add following dependency for JPA

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 <version>${springframework.version}</version>  
</dependency>

* + - * 1. Add properties in yaml file

spring:  
 datasource:  
 platform: POSTGRESQL  
 url: jdbc:postgresql://localhost:5432/test  
 username: postgres  
 password: postgres  
 jpa:  
 database: POSTGRESQL  
 show-sql: true  
 generate-ddl: true  
 properties:  
 hibernate:  
 jdbc:  
 lob:  
 non\_contextual\_creation: true  
 ddl-auto:  
 create-drop: true

as shown in above code datasource object contains information about database which is to be used. ‘jpa’ object contains the information about operations like create-drop says about if table is already present then drop it and create a new one. It has many properties to explore.

* + - * 1. Create a class and extend it with the JpaRepository

@Repository  
public interface CustomerRepository extends JpaRepository<Customer, String> {  
  
}

@Repository uses the class at the time of Jpa setup when we run the code.

We can add more customer method here.

1. **Lombok**

**For every Pojo class we write getter, setter method and default constructor. Also for controller, services and dao layer we need logger object. Lombok library provide set of annotations which takes away this repetitive code. For example @Getter provides getter method for all the attributes, @Slf4j provides a ‘log’ variable which is used for logging. In infa microservice we used @Data for Pojo class which provides getters, setters and default constructor. They are many other annotations that can be used under Lombok can be found** [here](https://projectlombok.org)**.**

**To use Lombok in intelliJ Idea we need to install Lombok plugin and set annotation processing enable in settings.**

1. **SonarQube and SonarLint**

SonarLint is an IDE extension that helps you detect and fix quality issues as you write code.  
Like a spell checker, SonarLint squiggles flaws so that they can be fixed before committing code.

To do sonarlint analysis you need to add the sonalint plugin. Then right click on the project and start the sonalint analysis. You can see the issues in the sonarlint console under report tab. Click on any issue and read the instruction to fix it.

SonarQube provides the detailed UI based view of Reliability, Security, Maintainability, Coverage, Duplications. We can browse through any file and see the code quality and other factors and the instruction to fix it too with an example. It helps to maintain code standards.

SonarQube service can be run simple using the docker. Use following steps to integrate sonarqube.

1. Pull the docker and run using following command

docker run -d --name sonarqube -p 9000:9000 sonarqube

1. Then Add the following dependency in pom.xml

<plugin>  
 <groupId>org.sonarsource.scanner.maven</groupId>  
 <artifactId>sonar-maven-plugin</artifactId>  
 <version>${sonarscanner.version}</version>  
 <executions>  
 <execution>  
 <phase>test</phase>  
 <goals>  
 <goal>sonar</goal>  
 </goals>  
 </execution>  
 </executions>  
</plugin>

By default it maven looks for the service running on [localhost:9000](localhost:9000) but if you want to specify sonarqube from somewhere else then use following properties under properties in pom.xml

<properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <sonar.host.url>http://www.example.com/</sonar.host.url>  
 <sonar.jdbc.url>jdbc:postgresql://www.example.com/sonar</sonar.jdbc.url>  
 <sonar.jdbc.driver>org.postgresql.Driver</sonar.jdbc.driver>  
 <sonar.jdbc.username>sonar</sonar.jdbc.username>  
 <sonar.jdbc.password>sonar</sonar.jdbc.password>  
 <sonar.exclusions>org/binarytherapy/generated/\*\*/\*, \*\*/GuiceBindComposer.java</sonar.exclusions>  
 <sonar.dynamic>reuseReports</sonar.dynamic>  
</properties>

1. Run the following command to see the reports

mvn clean test

And hit the url specified in pom.xml or else default one [localhost:9000](localhost:9000)

1. **Gatling**

Gatling is a powerful open-source load testing solution.

Gatling is designed for continuous load testing and integrates with your development pipeline. Gatling includes a web recorder and colourful reports.

Load testing for web applications consists in:

1) Simulating a large number of users with complex behaviours, by crafting requests on the HTTP protocol, or any other protocol.

2) Collecting and aggregating all the requests’ response times

3) Creating reports & analysing data

Gatling’s code-like scripting enables you to easily maintain your testing scenarios and easily automate them in your continuous delivery pipeline.

Use following steps to integrate Gatling

1. Add the dependencies and plugins in pom.xml

Plugin

<plugin>  
 <groupId>io.gatling</groupId>  
 <artifactId>gatling-maven-plugin</artifactId>  
 <version>${gatling-maven-plugin.version}</version>  
 <executions>  
 <execution>  
 <phase>test</phase>  
 <goals>  
 <goal>test</goal>  
 </goals>  
 <configuration>  
 <simulationClass>com.ikea.isx.simulation.MicroserviceSimulation</simulationClass>  
 </configuration>  
 </execution>  
 </executions>  
</plugin>

Dependencies

<dependency>  
 <groupId>io.gatling.highcharts</groupId>  
 <artifactId>gatling-charts-highcharts</artifactId>  
 <version>${gatling.version}</version>  
</dependency>  
<dependency>  
 <groupId>io.gatling</groupId>  
 <artifactId>gatling-app</artifactId>  
 <version>${gatling.version}</version>  
</dependency>  
<dependency>  
 <groupId>io.gatling</groupId>  
 <artifactId>gatling-recorder</artifactId>  
 <version>${gatling.version}</version>  
</dependency>

1. **Create one package under test named resources and copy paste three files**

****

1. Create Scala package under test and follow the same package structure as followed in src /main/java. Save the environment variables in one file to be used in the project. There is basic example of load test in the code.
2. **Jacoco**

Code coverage is a software metric used to measure how many lines of our code are executed during automated tests. We are going to use Jacoco for code coverage.

To use jacoco go through following steps

1. **Add the jacoco plugin in pom.xml**

<plugin>  
 <groupId>org.jacoco</groupId>  
 <artifactId>jacoco-maven-plugin</artifactId>  
 <version>0.8.2</version>  
 <configuration>  
 <excludes>  
 <exclude>\*\*/\*com/ikea/isx/model/Customer.class</exclude>  
 </excludes>  
 </configuration>  
 <executions>  
 <execution>  
 <goals>  
 <goal>prepare-agent</goal>  
 </goals>  
 </execution>  
 <!-- attached to Maven test phase -->  
 <execution>  
 <id>report</id>  
 <phase>test</phase>  
 <goals>  
 <goal>report</goal>  
 </goals>  
 </execution>  
 </executions>  
</plugin>

1. **We can see jacoco reports form file index.html in folder target/site/jacoco. Just double click the index.html and it will open in browser.**
2. **Artifactory**

**The dependencies we add in pom.xml are downloaded from maven central repository. If any dependency becomes unavailable while we deploy he project on new environment then we won’t be able to run the project or deploy it. We can have our own repository in which we can save the dependencies instead of maven central repository. If any dependency is not already in the repository that we set, then it will download from maven and save the dependency jar for future use. This can be done with artifactory.**

**To set the artifactory in maven project use following steps**

1. **Setup the artifactory on cloud**
2. **Add the repository information in pom.xml**

<repositories>  
 <repository>  
 <id>dml</id>  
 <name>IKEA JFrog Artifactory</name>  
 <url>https://artifactory.build.ingka.ikea.com/artifactory/erix-maven-virtual/</url>  
 <releases>  
 <enabled>true</enabled>  
 </releases>  
 <snapshots>  
 <enabled>true</enabled>  
 </snapshots>  
 </repository>  
</repositories>  
  
<pluginRepositories>  
 <pluginRepository>  
 <id>dml-plugins</id>  
 <name>IKEA JFrog Artifactory</name>  
 <url>https://artifactory.build.ingka.ikea.com/artifactory/erix-maven-virtual/</url>  
 <releases>  
 <enabled>true</enabled>  
 </releases>  
 <snapshots>  
 <enabled>true</enabled>  
 </snapshots>  
 </pluginRepository>  
</pluginRepositories>

1. **Edit the settings.xml file in the .m2 folder**

<?xml version="1.0" encoding="UTF-8"?>  
<settings xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 http://maven.apache.org/xsd/settings-1.1.0.xsd" xmlns="http://maven.apache.org/SETTINGS/1.1.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
<servers>  
 <server>  
 <id>dml</id>  
 <username>{artifactory username}</username>  
 <password>{artifactory password/token}</password>  
 </server>  
 <server>  
 <id>dml-plugins</id>  
 <username>{artifactory username}</username>  
 <password>{artifactory password/token}</password>  
 </server>  
</servers>  
</settings>