Microservices

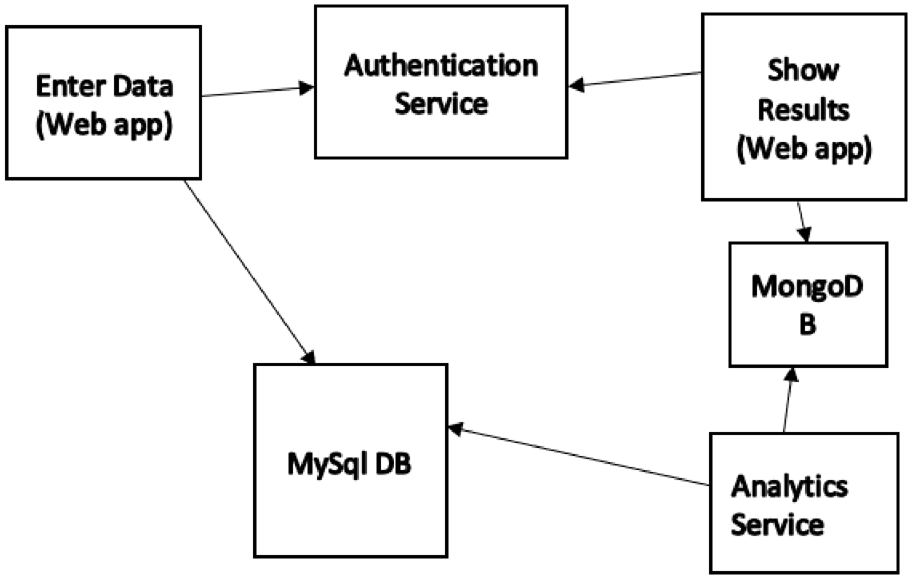
Assignment 5

Mammon Ahmad Alshamali

14/5/2022

Outline

Introduction



The system microservices can be described as follows:

- Enter data (web app) is used to collect data (any data: grades, temperatures, ...etc.). Users are allowed to enter data after validating their credentials through the Authentication Service. Any entered data will be written to the MySQL DB service.
- Show results (web app) is used to present simple analytics (max, min, average …etc.). Users are allowed to see the analytics after validating their credentials through the Authentication Service. The Show Results service reads data from the Mongo DB service.
- The Authentication Service is a simple service to validate users' credentials.
- The Analytics Service read data from the MySQL DB service and gets simple statistics like Max, Min, Avg…etc. and write that to Mongo DB Service.

From the requirements, the solution will have two web apps, four rest API and, two databases. After that six docker images and finally a docker-compose to run all of them together.

All the projects are built using spring boot, and for web apps, thyme leaf is used because the spring boot environment is helpful to deal with it.

System Details

The system was developed to be used on localhost for testing purposes. After ensuring the system was working correctly in the localhost, the next stage was replacing these localhost URLs with DNS which is created in the docker network.

- Enter Data (web app)

  This app will have two main pages and one common error page. The first page is the login page to authenticate the user. After the user inserts the login details, the data will be sent to the authentication service to validate the user. The communication method used in the code:

```
private Boolean isAuthenticated(User user){
    HttpEntity<User> request = new HttpEntity<>(user);
    String AUTHENTICATION_URL = "http://authentication:8085/authenticate/user";
    ResponseEntity<Boolean> response = restTemplate
            .exchange(AUTHENTICATION_URL, HttpMethod.POST, request, Boolean.class);
    return Boolean.TRUE.equals(response.getBody());
}
```

After login, a session will be created for the user. The user will be moved to enter the data page and can enter numbers on this page.

After entering the numbers (data), the number will be sent to the MySQL service. The communication method used in the code:

```
private void passNumberToDatabase(Numbers number){
    HttpEntity<Numbers> request = new HttpEntity<>(number);
    String MYSQL_RESTAPI_URL = "http://mysqlService:8086/addNumber";
    restTemplate.exchange(MYSQL_RESTAPI_URL,HttpMethod.POST,request,Object.class);
}
```

For errors, there is a common error handler to deal with the errors and sent the user to the error page.

The web app pages:

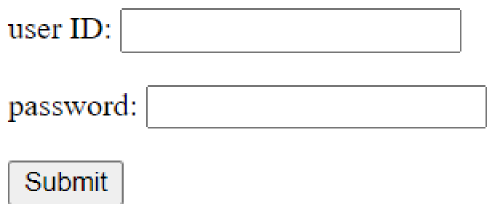


Figure 1 login page

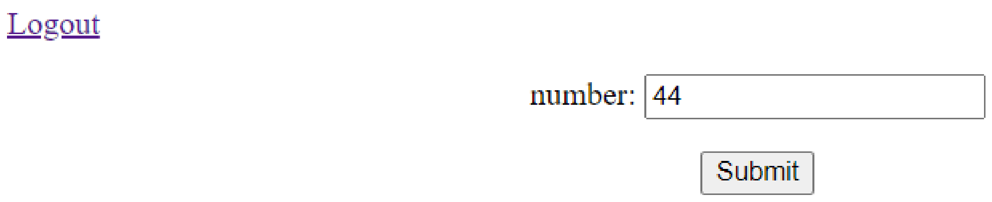
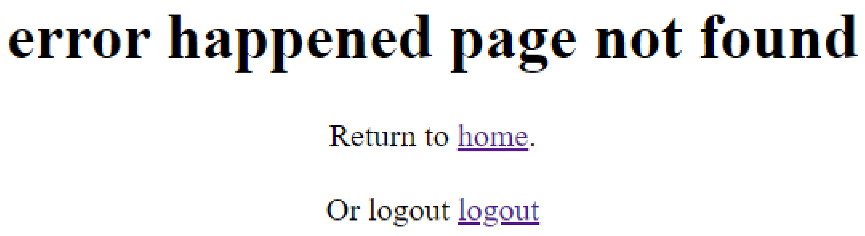


Figure 2 enter numbers page



error happened page not found

Return to home.

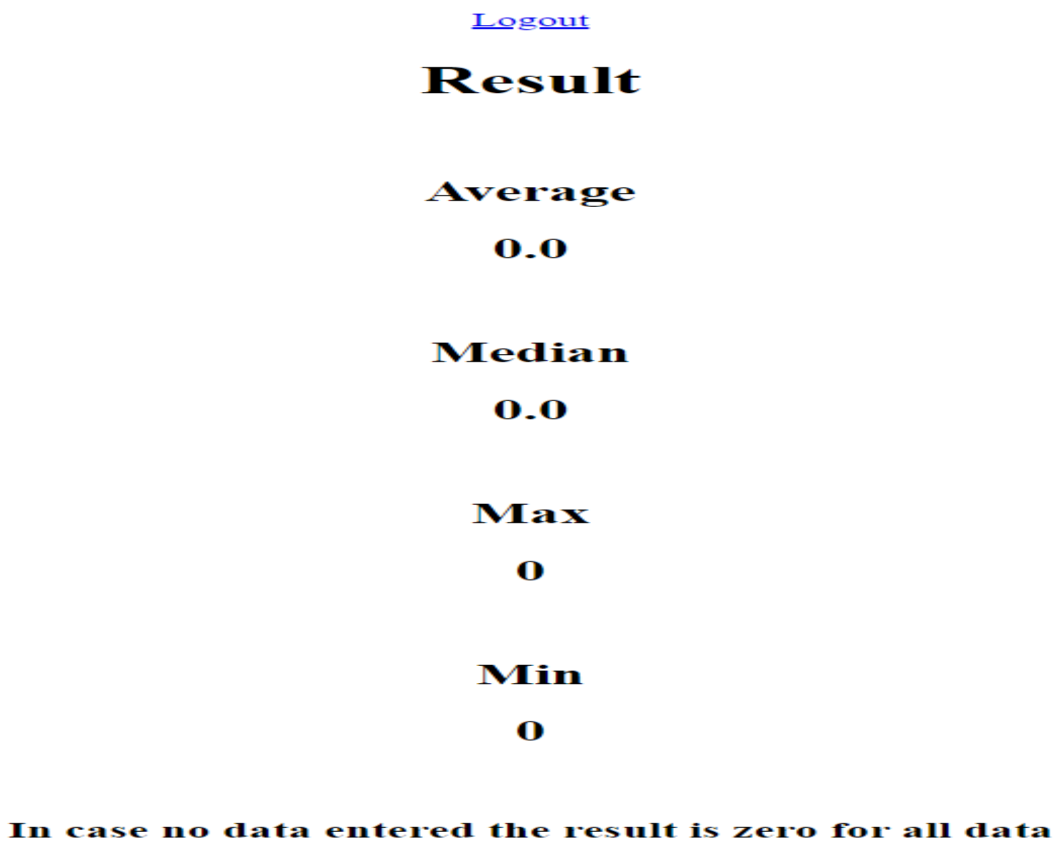
Or logout logout

Figure 3 error page

- Show Data (web app)

Most of the things which exist in the enter-data web app exist in this app too. After login, the user will be moved to the show data page, every few seconds this page sends a request to the Mongo service to get the results. The method to communicate is:

```
private NumbersData getDataFromApi() {
    String DATA_URL="http://mongoService:7080/numbersData";
    ResponseEntity<NumbersData> response =
            restTemplate.getForEntity(DATA_URL,NumbersData.class);
    return response.getBody();
}
```


Logout

Result

Average

0.0

Median

0.0

Max

0

Min

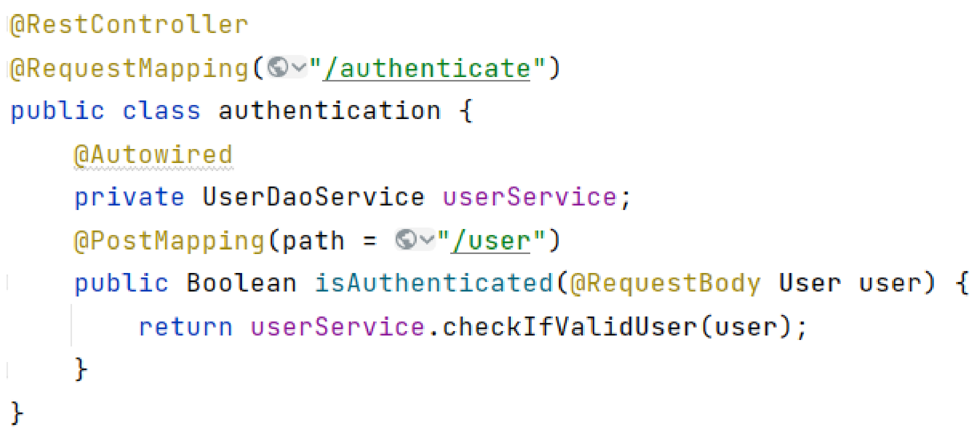
0

In case no data entered the result is zero for all data

Figure 4 show data page

- Authentication- service:

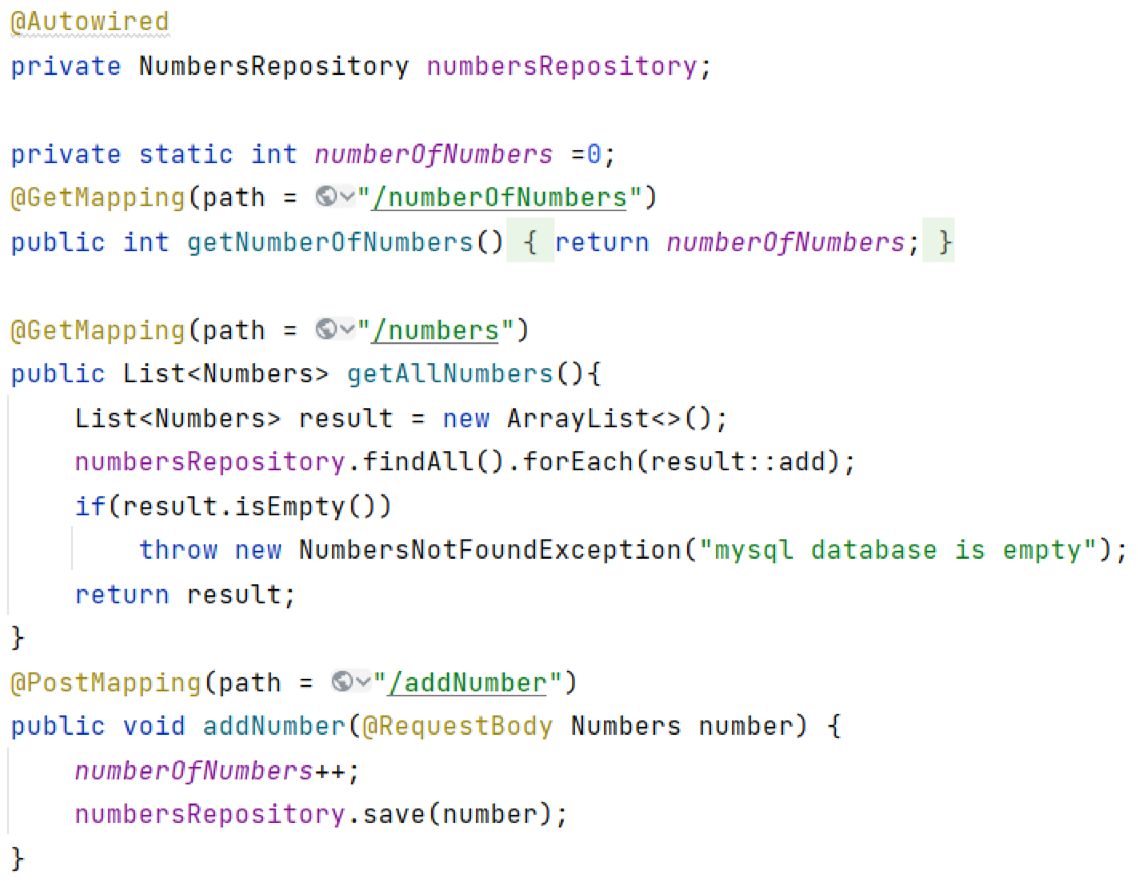
  The authentication service is a rest API, it has a static database. The user details are received by post method:



```
@RestController
@RequestMapping("/authenticate")
public class authentication {
    @Autowired
    private UserDaoService userService;
    @PostMapping(path = "/user")
    public Boolean isAuthenticated(@RequestBody User user) {
        return userService.checkIfValidUser(user);
    }
}
```

- MySQL Service

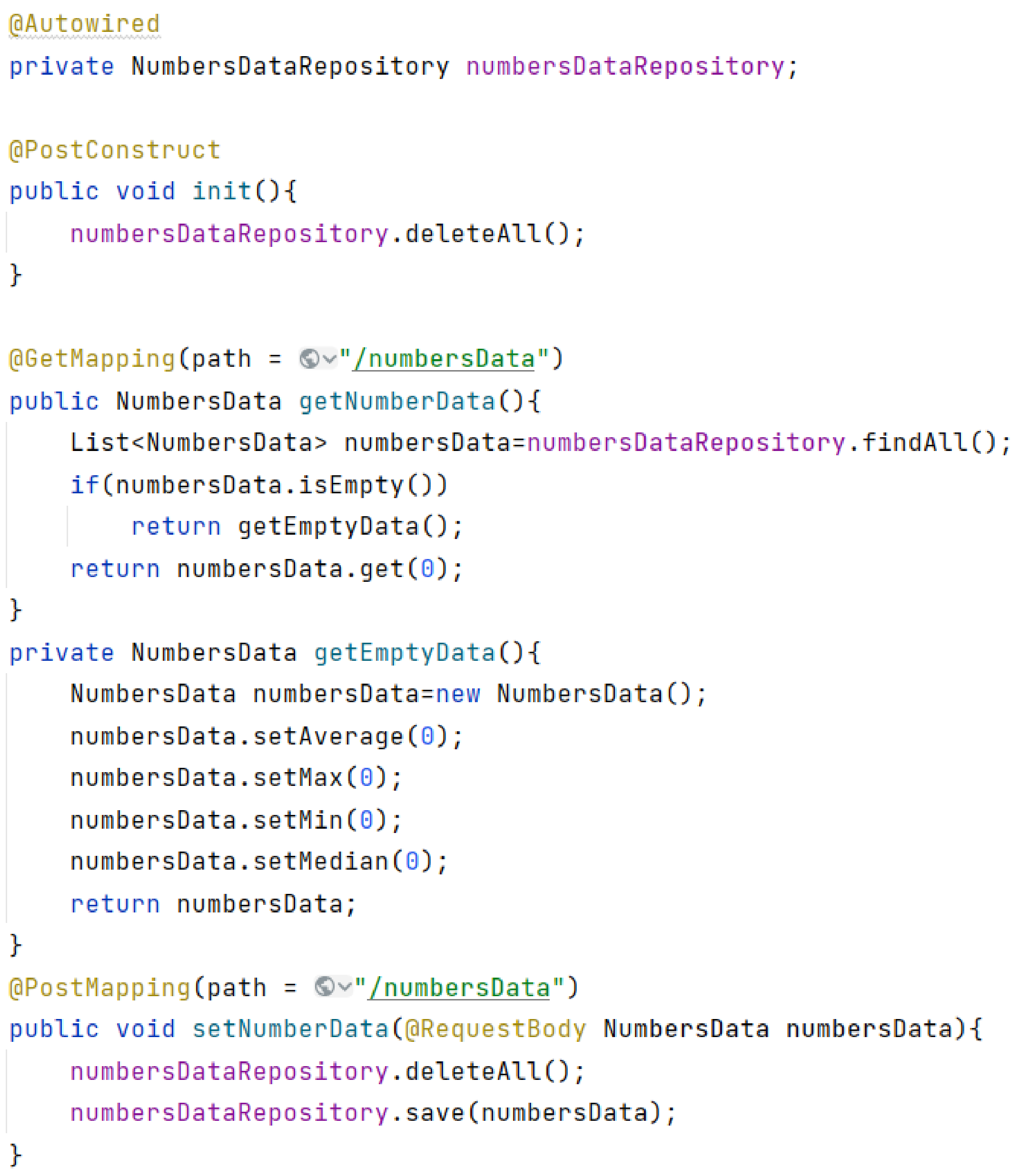
  The MySQL service is a rest API, it is connected to a MySQL database. The rest API uses spring data JPA to store and retrieve data in a relational database. The duties for this rest API are to count the number of the received input from the user, store the input from the user, and get the list of the stored data.



```java
@Autowired
private NumbersRepository numbersRepository;

private static int numberOfNumbers =0;
@GetMapping(path = "/numberOfNumbers")
public int getNumberOfNumbers() { return numberOfNumbers; }

@GetMapping(path = "/numbers")
public List<Numbers> getAllNumbers(){
    List<Numbers> result = new ArrayList<>();
    numbersRepository.findAll().forEach(result::add);
    if(result.isEmpty())
        throw new NumbersNotFoundException("mysql database is empty");
    return result;
}
@PostMapping(path = "/addNumber")
public void addNumber(@RequestBody Numbers number) {
    numberOfNumbers++;
    numbersRepository.save(number);
}
```

- Mongo Service

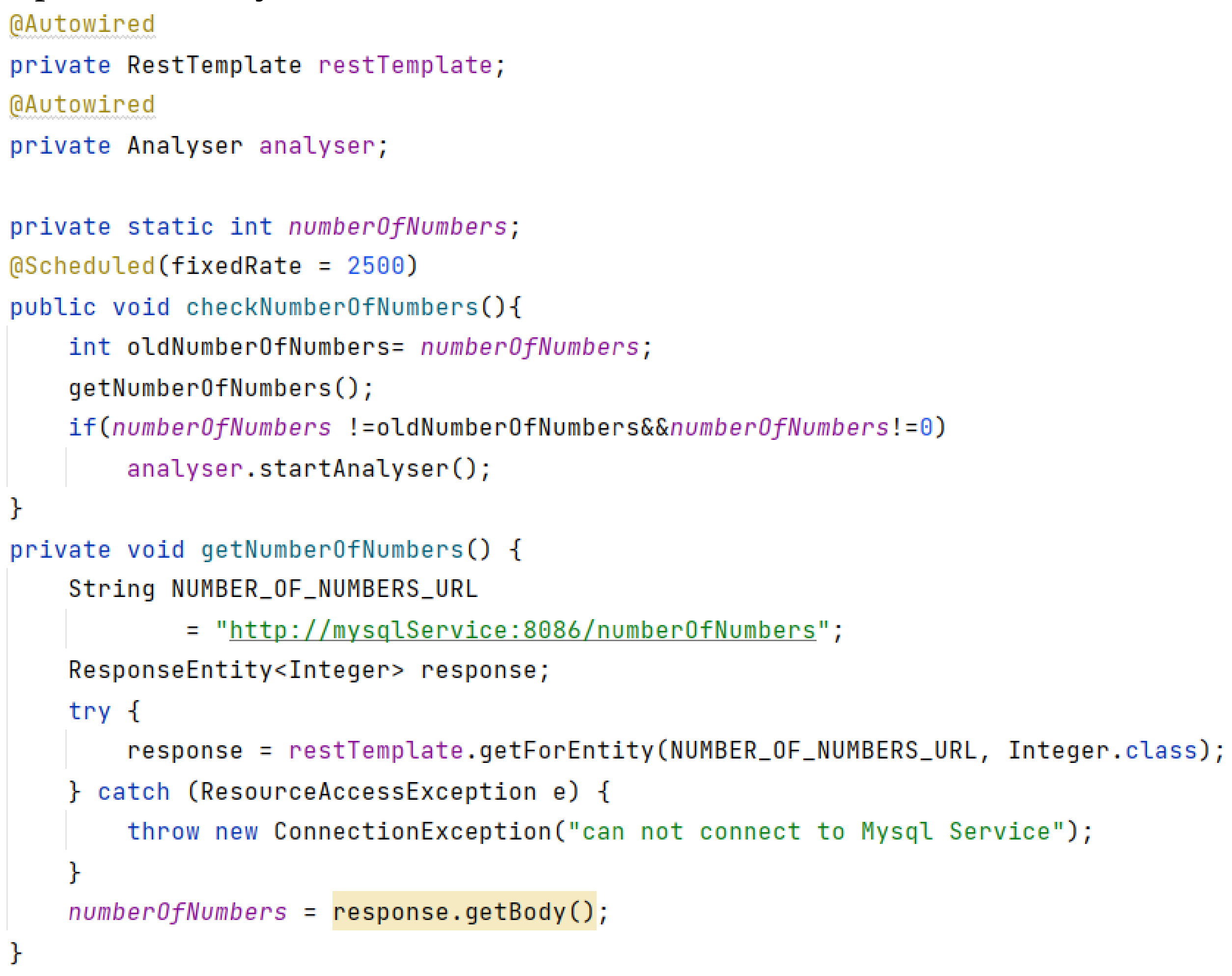
  The Mongo service is a rest API, it is connected to MongoDB. The rest API uses spring data MongoDB to store data and retrieves it from MongoDB. The rest API duties are receiving the data from the analyzer and storing the data in MongoDB. What is more, retrieve the data from MongoDB and send it to the show data web app.



```
@Autowired
private NumbersDataRepository numbersDataRepository;

@PostConstruct
public void init(){
    numbersDataRepository.deleteAll();
}

@GetMapping(path = "/numbersData")
public NumbersData getNumberData(){
    List<NumbersData> numbersData=numbersDataRepository.findAll();
    if(numbersData.isEmpty())
        return getEmptyData();
    return numbersData.get(0);
}
private NumbersData getEmptyData(){
    NumbersData numbersData=new NumbersData();
    numbersData.setAverage(0);
    numbersData.setMax(0);
    numbersData.setMin(0);
    numbersData.setMedian(0);
    return numbersData;
}
@PostMapping(path = "/numbersData")
public void setNumberData(@RequestBody NumbersData numbersData){
    numbersDataRepository.deleteAll();
    numbersDataRepository.save(numbersData);
}
```

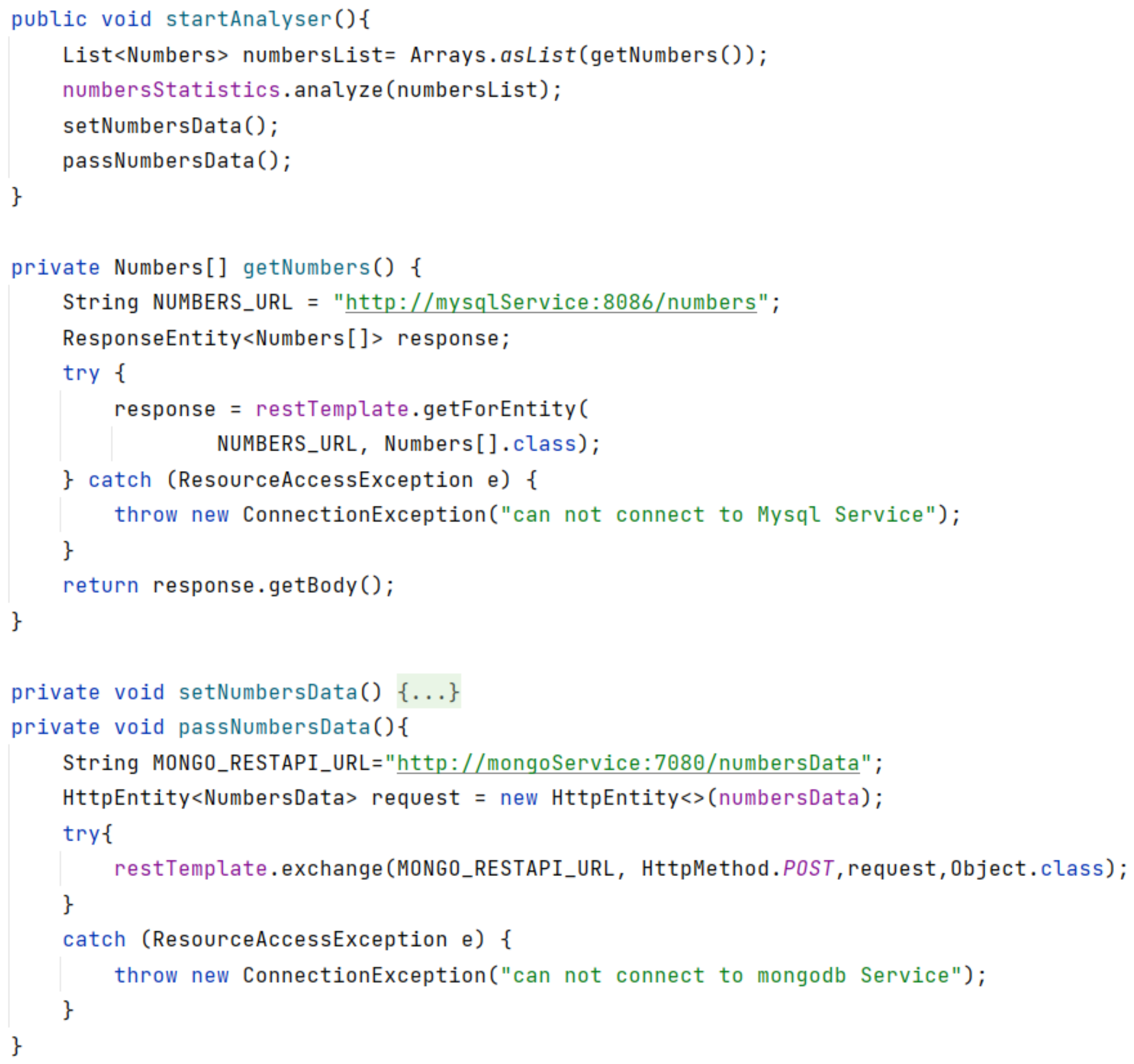
- The Analyzer

  From the requirements of the assignment, it shows the Analyzer should send the request to the MySQL service and MongoDB service. For this, the Awaitility 3.1.2 Dependency enables scheduling the tasks. The task which has been scheduled is checking the number of inputs at the MySQL service.



```java
@Autowired
private RestTemplate restTemplate;
@Autowired
private Analyser analyser;

private static int numberOfNumbers;
@Scheduled(fixedRate = 2500)
public void checkNumberOfNumbers(){
    int oldNumberOfNumbers= numberOfNumbers;
    getNumberOfNumbers();
    if(numberOfNumbers !=oldNumberOfNumbers&&numberOfNumbers!=0)
        analyser.startAnalyser();
}
private void getNumberOfNumbers() {
    String NUMBER_OF_NUMBERS_URL
            = "http://mysqlService:8086/numberOfNumbers";
    ResponseEntity<Integer> response;
    try {
        response = restTemplate.getForEntity(NUMBER_OF_NUMBERS_URL, Integer.class);
    } catch (ResourceAccessException e) {
        throw new ConnectionException("can not connect to Mysql Service");
    }
    numberOfNumbers = response.getBody();
}
```

The Analyzer’s job is to get data from the MySQL service, find statistics, and pass them to the MongoDB service



```
public void startAnalyser(){
    List<Numbers> numbersList= Arrays.asList(getNumbers());
    numbersStatistics.analyze(numbersList);
    setNumbersData();
    passNumbersData();
}

private Numbers[] getNumbers() {
    String NUMBERS_URL = "http://mysqlService:8086/numbers";
    ResponseEntity<Numbers[]> response;
    try {
        response = restTemplate.getForEntity(
                NUMBERS_URL, Numbers[].class);
    } catch (ResourceAccessException e) {
        throw new ConnectionException("can not connect to Mysql Service");
    }
    return response.getBody();
}

private void setNumbersData() {...}
private void passNumbersData(){
    String MONGO_RESTAPI_URL="http://mongoService:7080/numbersData";
    HttpEntity<NumbersData> request = new HttpEntity<>(numbersData);
    try{
        restTemplate.exchange(MONGO_RESTAPI_URL, HttpMethod.POST,request,Object.class);
    }
    catch (ResourceAccessException e) {
        throw new ConnectionException("can not connect to mongodb Service");
    }
}
```

- Clean Code and SOLID

  The code in the project is simple and follows clean code and SOLID principles. These principles are noticeable in many things like the naming convention for the classes, variables, constants, and functions. Most of the functions do one thing and don’t have a lot of arguments. Comments are avoided; there is almost no single comment in the code. There is no return null or pass null in the total system.

  Classes are small, encapsulated, and follow Uncle Bob’s format; the private function should be after the first use, the local variable before the first use. So, to find a private function in the code just downward from the first usage. Constant is not inherited as well.

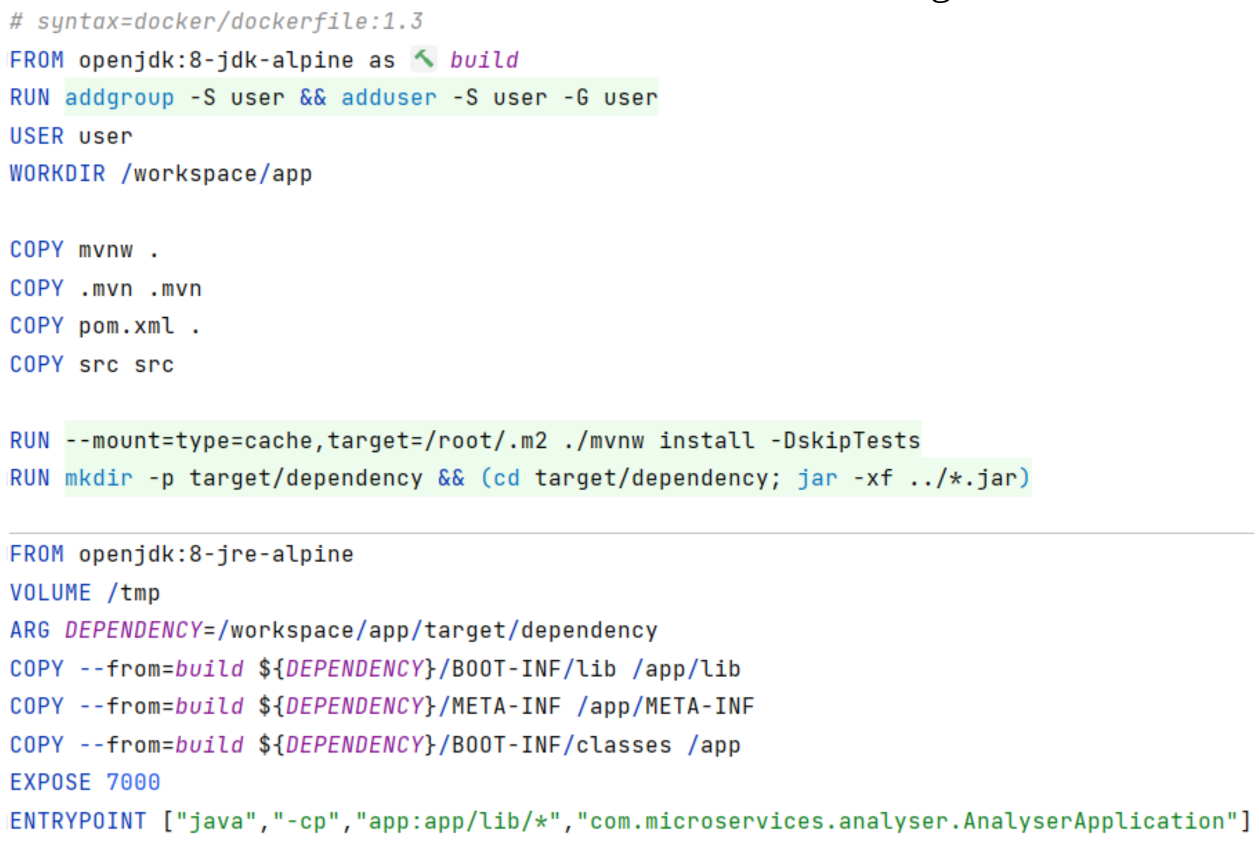
No dead code and no duplicated code in the same project. But for microservices, it is common to see the same class being used many times in different projects as an example the user class exists in three projects the two web apps and the authentication rest API.

Docker file

All the six images are the same except the last line in each docker file. Many things are considered while building the images and some fallbacks happened like using windows to build the images cannot enable the build kit feature. But the image can contain that code in case the image will rebuild in the future on another OS. Another solution is using “docker buildx” to build the images.

All the docker images are built according to https://spring.io/guides/topicals/spring-boot-docker/

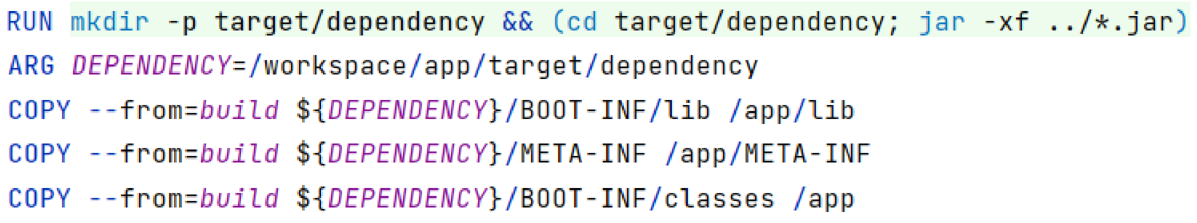
This is one of the docker files used to build the images:



```
# syntax=docker/dockerfile:1.3
FROM openjdk:8-jdk-alpine as build
RUN addgroup -S user && adduser -S user -G user
USER user
WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN --mount=type=cache,target=/root/.m2 ./mvnw install -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

FROM openjdk:8-jre-alpine
VOLUME /tmp
ARG DEPENDENCY=/workspace/app/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
EXPOSE 7000
ENTRYPOINT ["java","-cp","app:app/lib/*","com.microservices.analyser.AnalyserApplication"]
```

The base image is openjdk:8-jdk-alpine. The alpine images are smaller than the standard OpenJDK library images from the Docker hub. What is more, “20 MB can be saved in the base image by using the JRE label instead of JDK and this can be done in the next base image. Not all applications work with a JRE (as opposed to a JDK), but most do. The advantage of working with a JRE is avoiding the risk of misuse of some of the JDK features."

"A Spring Boot fat JAR naturally has "layers" because of the way that the JAR itself is packaged. If we unpack it first, it is already divided into external and internal dependencies. To do this in one step in the docker build, we need to unpack the JAR first."
The following commands will do the job:



```
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)
ARG DEPENDENCY=/workspace/app/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
```

"After advancing, the multistage was added to the docker file it allows to build JAR file and copying the result from one image to another. The first image is labeled build, and it is used to run Maven, build the fat JAR, and unpack it. the source code has been split into four layers. The later layers contain the build configuration and the source code for the application, and the earlier layers contain the build system itself (the Maven wrapper). This is a small optimization, and it also means that we do not have to copy the target directory to a docker image, even a temporary one used for the build."

"Every build where the source code changes are slow because the Maven cache must be re-created in the first RUN section. But we have a completely standalone build that anyone can run to get our application running as long as they have docker. That can be quite useful in some environments — for example, where you need to share your code with people who do not know Java."

The following commands do the job:

```
FROM openjdk:8-jdk-alpine as build
WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src
```

The first line makes a “way to cache build dependencies. The RUN directive accepts a new flag: --mount for more details” https://github.com/moby/buildkit/blob/master/frontend/dockerfile/docs/syntax.md
The following commands show do the job:

```
# syntax=docker/dockerfile:1.3
RUN --mount=type=cache,target=/root/.m2 ./mvnw install -DskipTests
```

“For security aspects, the processes should not be run with root permissions. Instead, the image should contain a non-root user that runs the application.”

“In a Docker file, this can be achieved by adding another layer that adds a (system) user and group and set it as the current user.”
The following commands do the job:

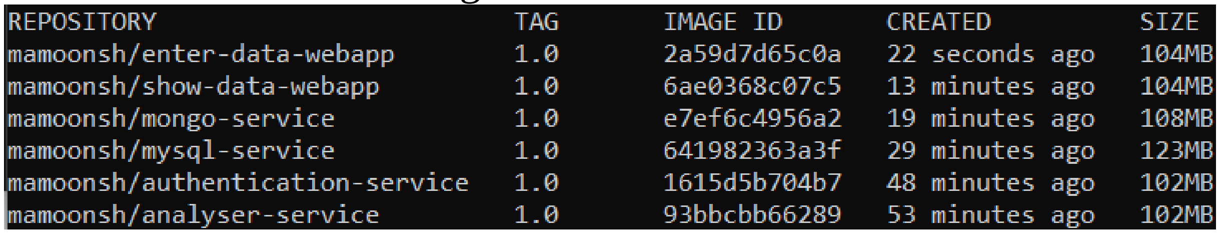
```
RUN addgroup -S user && adduser -S user -G user
USER user
```

“In case someone manages to break out of your application and run system commands inside the container, this precaution limits their capabilities”

To build the image this command was used in the terminal:

```
docker buildx build -t mamoonsh/analyser-service:1.0 .
```

The details of resulted images:

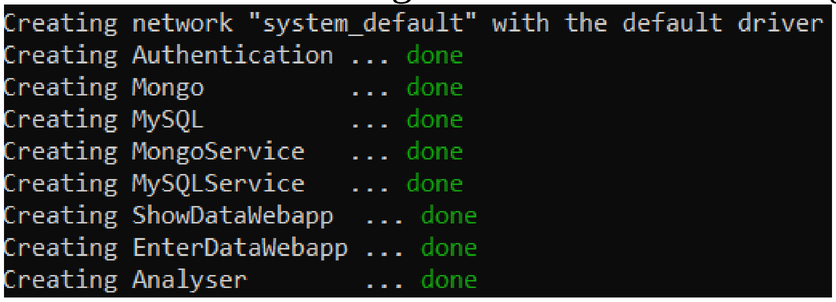


```
REPOSITORY                         TAG       IMAGE ID       CREATED          SIZE
mamoonsh/enter-data-webapp         1.0       2a59d7d65c0a   22 seconds ago   104MB
mamoonsh/show-data-webapp          1.0       6ae0368c07c5   13 minutes ago   104MB
mamoonsh/mongo-service             1.0       e7ef6c4956a2   19 minutes ago   108MB
mamoonsh/mysql-service             1.0       641982363a3f   29 minutes ago   123MB
mamoonsh/authentication-service    1.0       1615d5b704b7   48 minutes ago   102MB
mamoonsh/analyser-service          1.0       93bbcbb66289   53 minutes ago   102MB
```

Docker Compose

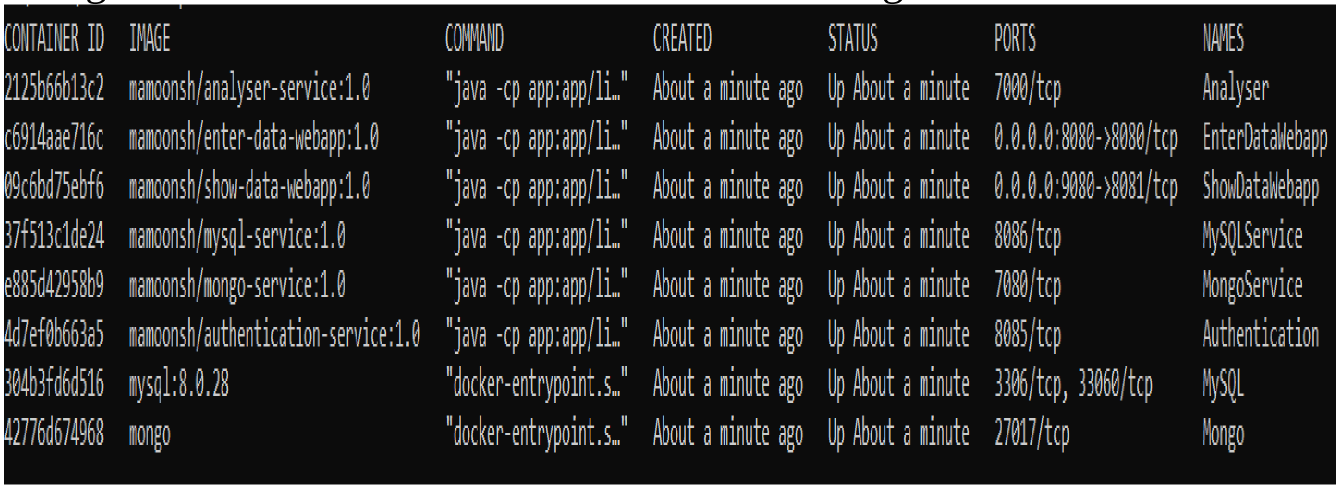
Docker-compose is important to run all the images together, this is also can be done without docker-compose but docker-compose makes things easier for the users.

In our Docker-compose file, there are 8 images which are the previous six images with MySQL image and mongo image. The next picture shows the order of creating containers for the 8 images:



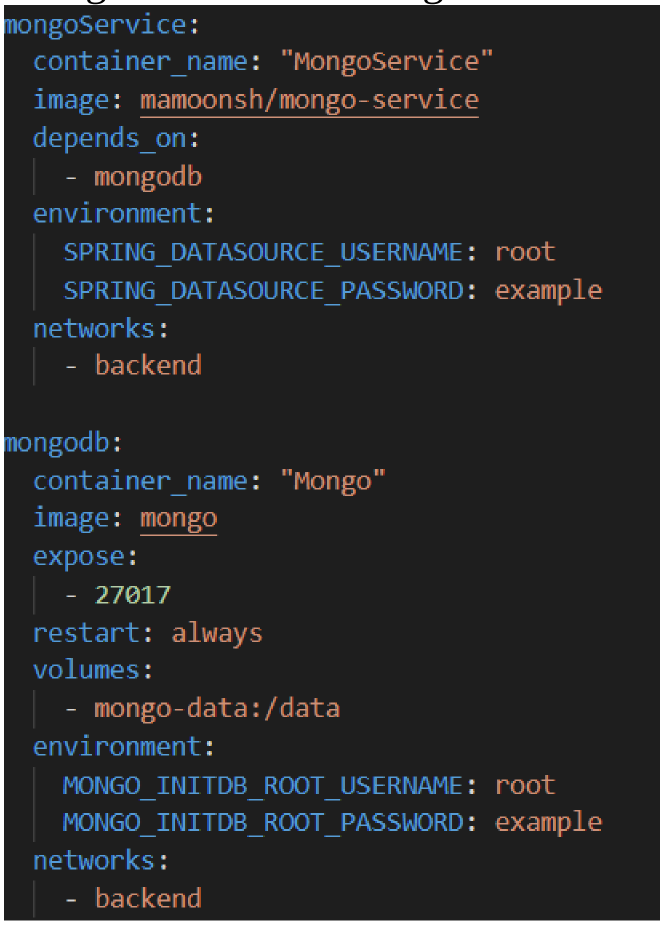
```
Creating network "system_default" with the default driver
Creating Authentication ... done
Creating Mongo          ... done
Creating MySQL          ... done
Creating MongoService   ... done
Creating MySQLService   ... done
Creating ShowDataWebapp  ... done
Creating EnterDataWebapp ... done
Creating Analyser        ... done
```

Using docker PS command we can see the following:



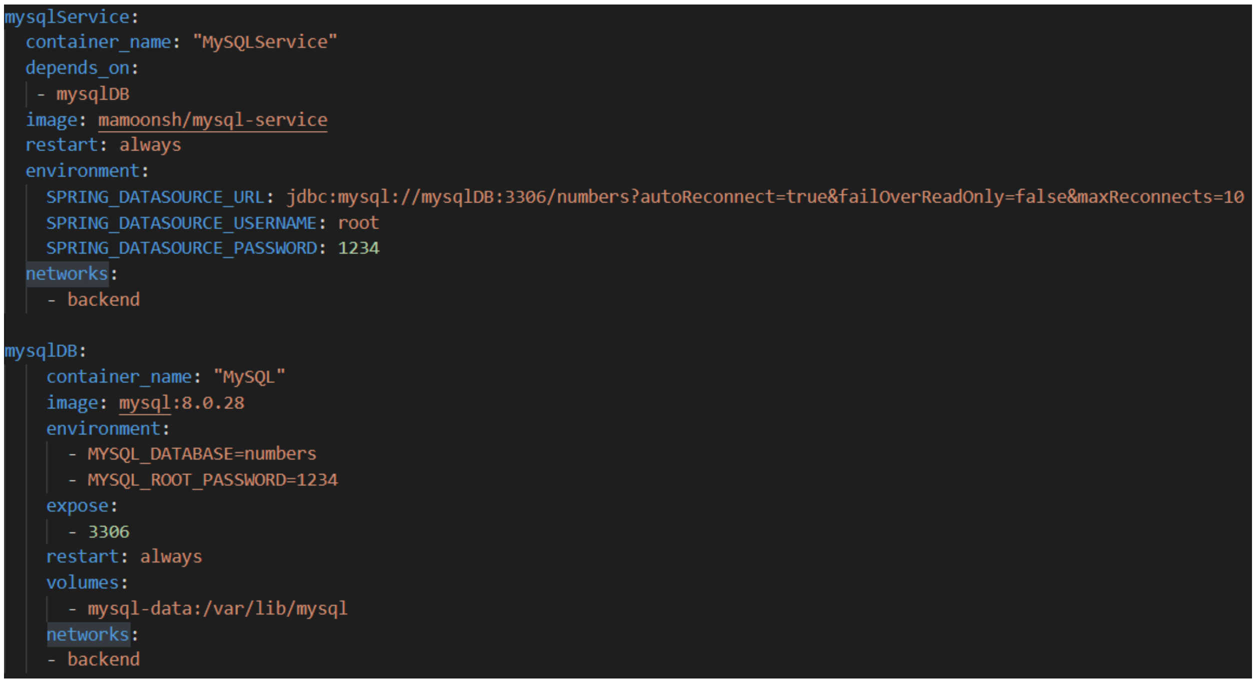
```
CONTAINER ID   IMAGE                                COMMAND                  CREATED            STATUS            PORTS                      NAMES
2125b66b13c2   mamoonsh/analyser-service:1.0        "java -cp app:app/li…"   About a minute ago Up About a minute 7000/tcp                   Analyser
c6914aae716c   mamoonsh/enter-data-webapp:1.0       "java -cp app:app/li…"   About a minute ago Up About a minute 0.0.0.0:8080->8080/tcp     EnterDataWebapp
09c6bd75ebf6   mamoonsh/show-data-webapp:1.0        "java -cp app:app/li…"   About a minute ago Up About a minute 0.0.0.0:9080->8081/tcp     ShowDataWebapp
37f513c1de24   mamoonsh/mysql-service:1.0           "java -cp app:app/li…"   About a minute ago Up About a minute 8086/tcp                   MySQLService
e885d42958b9   mamoonsh/mongo-service:1.0           "java -cp app:app/li…"   About a minute ago Up About a minute 7080/tcp                   MongoService
4d7ef0b663a5   mamoonsh/authentication-service:1.0  "java -cp app:app/li…"   About a minute ago Up About a minute 8085/tcp                   Authentication
304b3fd6d516   mysql:8.0.28                         "docker-entrypoint.s…"   About a minute ago Up About a minute 3306/tcp, 33060/tcp        MySQL
42776d674968   mongo                                "docker-entrypoint.s…"   About a minute ago Up About a minute 27017/tcp                  Mongo
```

- Docker-Compose code
  Mongo service and MongoDB:

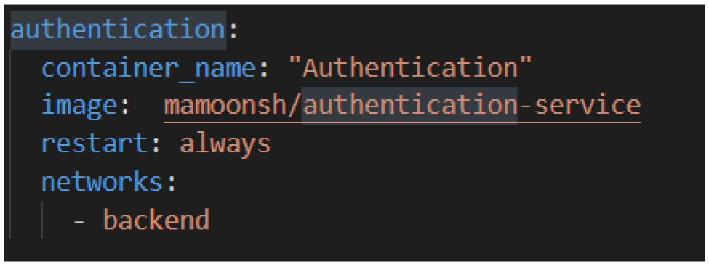


```yaml
mongoService:
  container_name: "MongoService"
  image: mamoonsh/mongo-service
  depends_on:
    - mongodb
  environment:
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: example
  networks:
    - backend

mongodb:
  container_name: "Mongo"
  image: mongo
  expose:
    - 27017
  restart: always
  volumes:
    - mongo-data:/data
  environment:
    MONGO_INITDB_ROOT_USERNAME: root
    MONGO_INITDB_ROOT_PASSWORD: example
  networks:
    - backend
```

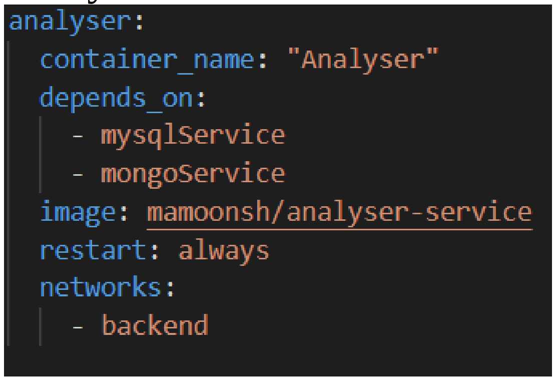
MySQL service and MySQL:


```yaml
mysqlService:
  container_name: "MySQLService"
  depends_on:
   - mysqlDB
  image: mamoonsh/mysql-service
  restart: always
  environment:
    SPRING_DATASOURCE_URL: jdbc:mysql://mysqlDB:3306/numbers?autoReconnect=true&failOverReadOnly=false&maxReconnects=10
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: 1234
  networks:
    - backend

mysqlDB:
    container_name: "MySQL"
    image: mysql:8.0.28
    environment:
      - MYSQL_DATABASE=numbers
      - MYSQL_ROOT_PASSWORD=1234
    expose:
      - 3306
    restart: always
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
    - backend
```

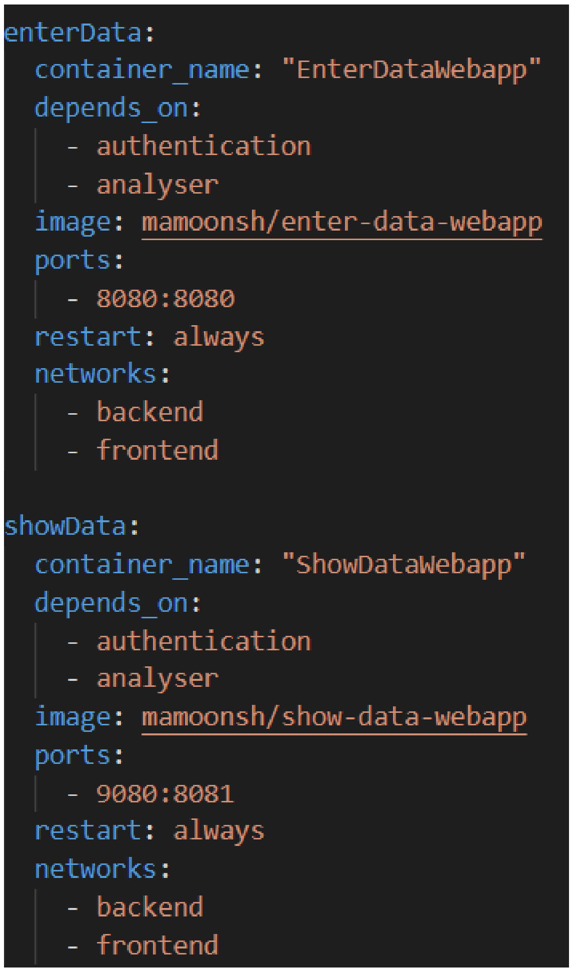
- Authentication


```yaml
authentication:
  container_name: "Authentication"
  image:  mamoonsh/authentication-service
  restart: always
  networks:
    - backend
```

- Analyzer

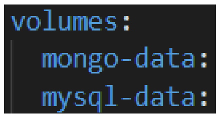

```yaml
analyser:
  container_name: "Analyser"
  depends_on:
    - mysqlService
    - mongoService
  image: mamoonsh/analyser-service
  restart: always
  networks:
    - backend
```

- Enter data web app and Show data web app



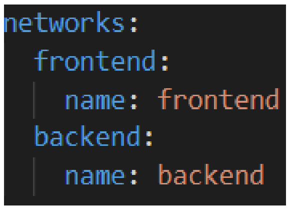
```
enterData:
  container_name: "EnterDataWebapp"
  depends_on:
    - authentication
    - analyser
  image: mamoonsh/enter-data-webapp
  ports:
    - 8080:8080
  restart: always
  networks:
    - backend
    - frontend

showData:
  container_name: "ShowDataWebapp"
  depends_on:
    - authentication
    - analyser
  image: mamoonsh/show-data-webapp
  ports:
    - 9080:8081
  restart: always
  networks:
    - backend
    - frontend
```

- Volumes



```
volumes:
  mongo-data:
  mysql-data:
```

- Network



```
networks:
  frontend:
    name: frontend
  backend:
    name: backend
```