CS4337 Lab Week 3: Spring Boot, Docker and MySQL

Lab Title: Developing, Debugging, and Analyzing a Spring Boot Application with Docker and MySQL

Duration: 2 hours

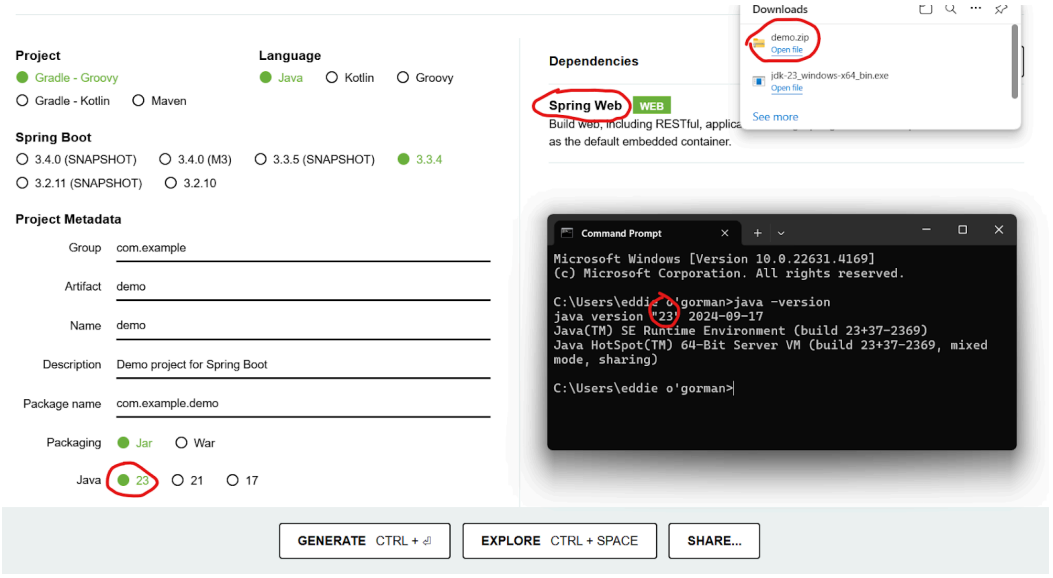
Overview:
In this lab, students will create a Spring Boot REST API using the IntelliJ Community Edition IDE, containerize it using Docker, and test it with Postman. They will connect the API to a MySQL database running in Docker, develop a client application to send bulk requests to the API, and observe the effects of varying traffic loads on Docker resource usage.

---

1. Creating a Spring Boot Application in IntelliJ (20 mins)

To initialise a simple Spring Boot project using Spring Initializr, open https://start.spring.io/ in a web browser.

a. Navigate to the “Dependencies” section and include the “Spring Web” dependency. Select the Java version that you are currently using under the “Project Metadata” section (this can be checked by running `java -version` in a command prompt or terminal).
b. Click “Generate”, then unzip the downloaded folder.
c. Open the folder in IntelliJ.



1.1 Dependency Selection in Spring Initializr

Set up a basic REST API endpoint using Spring Boot annotations like @RestController and @GetMapping.

a. Navigate to the Java file in the directory: src/main/java/com/example/demo/<your_file_name>
b. Paste in the following code:

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Map;

@SpringBootApplication
@RestController
public class DemoApplication {

   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }

   @GetMapping("/hello")
   public Map<String, Object> sayHello() {
      return Map.of("message", "Hello World!");
   }

}
```

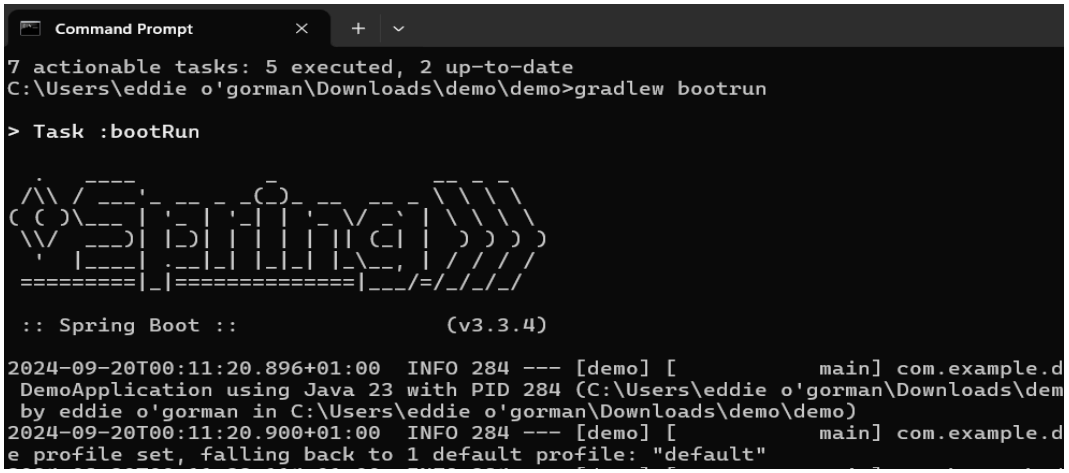
1.2 Code snippet for creating a basic Spring Boot REST API

a. Open a command prompt or terminal, then cd to the folder that contains your Java file.
b. Build and run your application using:

```
./gradlew build
./gradlew bootRun
```

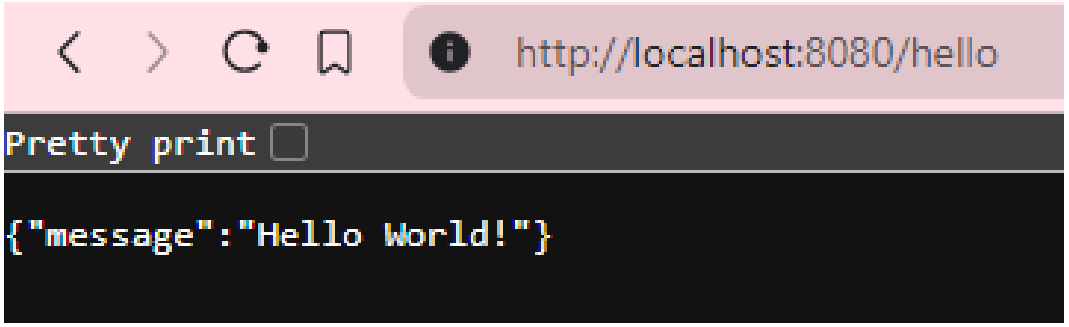
```
Welcome to Gradle 8.10.1!

Here are the highlights of this release:
 - Support for Java 23
 - Faster configuration cache
 - Better configuration cache reports

For more details see https://docs.gradle.org/8.10.1/release-notes.html

Starting a Gradle Daemon, 1 incompatible and 1 stopped Daemons could not be reused, use --status for details
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath
 has been appended

BUILD SUCCESSFUL in 23s
7 actionable tasks: 7 executed
```

1.3 Output from gradlew build command



1.4 Output of gradlew bootRun command

Open http://localhost:8080/hello in a web browser to view your output, then terminate in the terminal with Ctrl + C. Note that because the return type for this endpoint is a Map, Spring Boot automatically returns JSON thanks to its integration with *Jackson*, a Java JSON library.



1.5 Viewing the output in a browser.

Explore the directory structure and config files of your project. e.g., application.properties

Explore the options on https://start.spring.io/, e.g., different dependencies like Spring Boot DevTools for live reloading etc.

2. Debugging the Spring Boot Application in Docker (20 mins)

Write a Dockerfile to containerize the Spring Boot application. Make sure the entrypoint info is all on one line.

```
# base image
FROM openjdk:21-jdk-slim
# copy across build
COPY build/libs/demo-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java",
"-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005",
"-jar", "app.jar"]
```

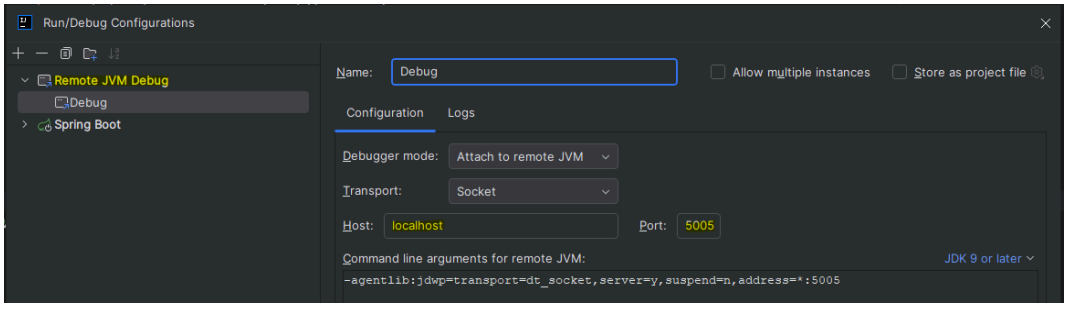
2.1 Dockerfile configuration

Make sure Docker Desktop is running, then run the application in a Docker container.

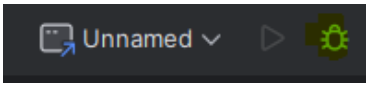
a. Create the image based on Dockerfile: `docker build -t springbootdemo .`
b. Run container from image: `docker run -p 8080:8080 -p 5005:5005 springbootdemo`
   -p 8080:8080 exposes port 8080 from the container to our local port 8080 so we can easily access the endpoints.

If you run into issue setting up your image you may not have the .jar file in scope. Go back where the .jar is in scope (it's in the build folder) and add -f to your docker build command and add the path to your Dockerfile

Attach a debugger from IntelliJ to the running container to identify and fix any issues. To do this, add a new Remote JVM Debug configuration with host localhost and port 5005. Then click the Debug icon and verify a connection to the target is made successfully.



2.2 Remote JVM Debug Configuration in IntelliJ.

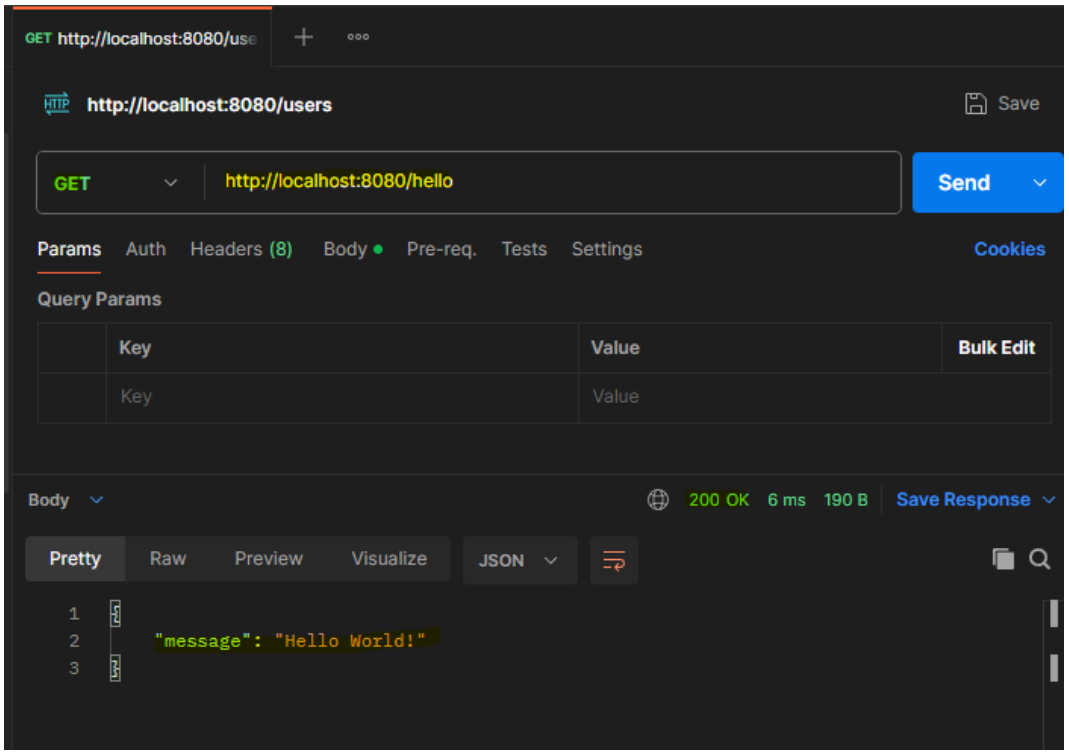


2.3 IntelliJ debug button.

2.4 Debugger Successfully Connected Output.

3. Testing the API Locally with Postman (10 mins)

Use Postman to test the API by sending GET requests. Verify that the API is working as expected by reviewing the responses.



3.1 Testing API Responses with Postman

4. Connecting Spring Boot to a MySQL Database in Docker (20 mins)

Set up a MySQL container in Docker. Create a docker-compose.yml with the following - this allows us to have multiple containers working together:

```yaml
services:
 mysql:
   image: mysql:latest
   container_name: db
   environment:
     MYSQL_ROOT_PASSWORD: 1234
     MYSQL_DATABASE: testing
   networks:
     - net
   ports:
   - "3306:3306"
 java-app:
   image: springbootdemo
   container_name: app
   depends_on:
     - mysql
   networks:
     - net
   ports:
     - "8080:8080"
networks:
   net:
     driver: bridge
```

4.1 Docker Compose Configuration for Spring Boot and MySQL Containers.

Configure the Spring Boot application to query a MySQL database (via application.properties or application.yml).

1. Add the following to application.properties (found in src/main/resources) or application.yml to configure the DB connection.
   ```
   spring.datasource.url=jdbc:mysql://db:3306/testing
   spring.datasource.username=root
   spring.datasource.password=1234
   spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
   ```
2. Connect to the database to add a table.
   a. Run the containers with `docker-compose up`
   b. Connect to the db container with `docker exec -it db bash`
   c. `mysql -u root -p`
   d. 1234
   e. `USE testing;`

f. `CREATE table users (id INT AUTO_INCREMENT, email VARCHAR(255), PRIMARY KEY(id));`

g. `INSERT INTO users (email) VALUES ("test@gmail.com");`

3. Test the API's integration with the MySQL database by creating API endpoints that retrieve or manipulate data.

   a. Add the following dependencies to your build.gradle for Java Database Connectivity (JDBC) and then load Gradle changes:

   ```
   implementation 'org.springframework.boot:spring-boot-starter-jdbc'
   implementation 'mysql:mysql-connector-java:8.0.33'
   ```

   b. Add the following attribute to your class - @Autowired ensures Spring Boot automatically creates a database connection based on config in application.properties file.

   ```
   @Autowired
   private JdbcTemplate jdbcTemplate;
   ```

   c. Add a GET endpoint to get a list of users. Example:

```
@GetMapping("/users")
public List<Map<String, Object>> users() {
   return jdbcTemplate.queryForList("SELECT * FROM users");
}
```

4.2 Spring Boot GET endpoint to get users list.

   d. Add a POST endpoint to add a new user to the users table. Example:

```
@PostMapping("/user")
public Map<String, Object> createUser(@RequestBody Map<String, String> fields)
{
   int success = jdbcTemplate.update("INSERT INTO users (email) VALUES (?)",
fields.get("email"));
   return Map.of("success", success, "email", fields.get("email"));
}
```

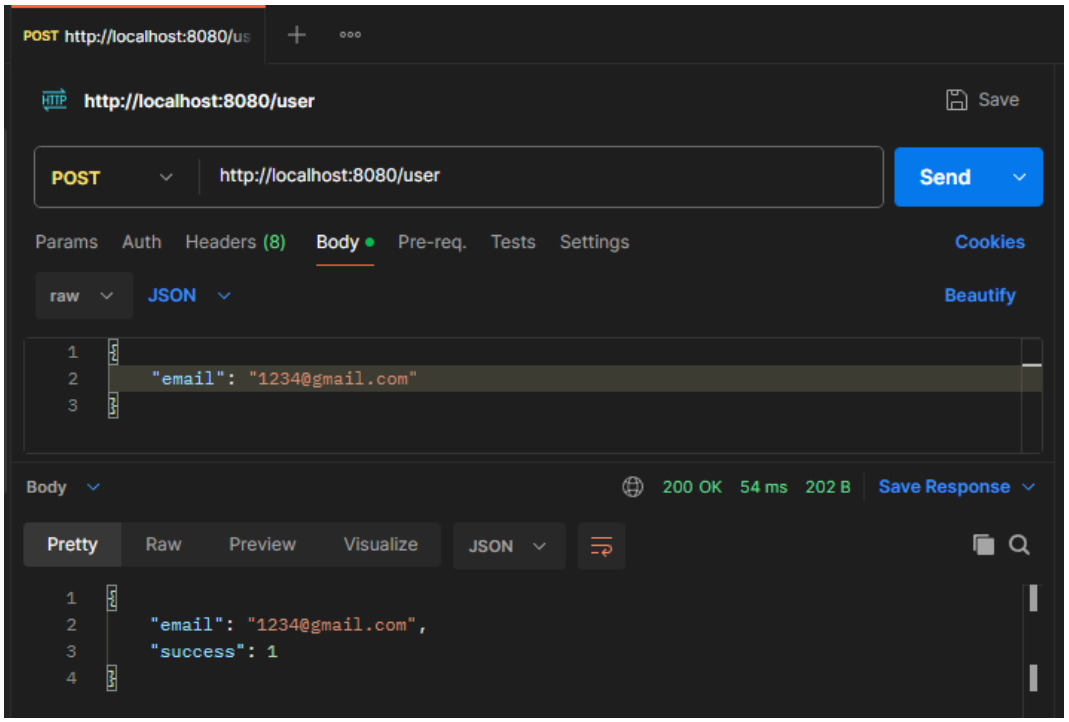
4.3 Spring Boot POST endpoint to create new user.

   e. Add the necessary imports suggested by IntelliJ (alt + enter).

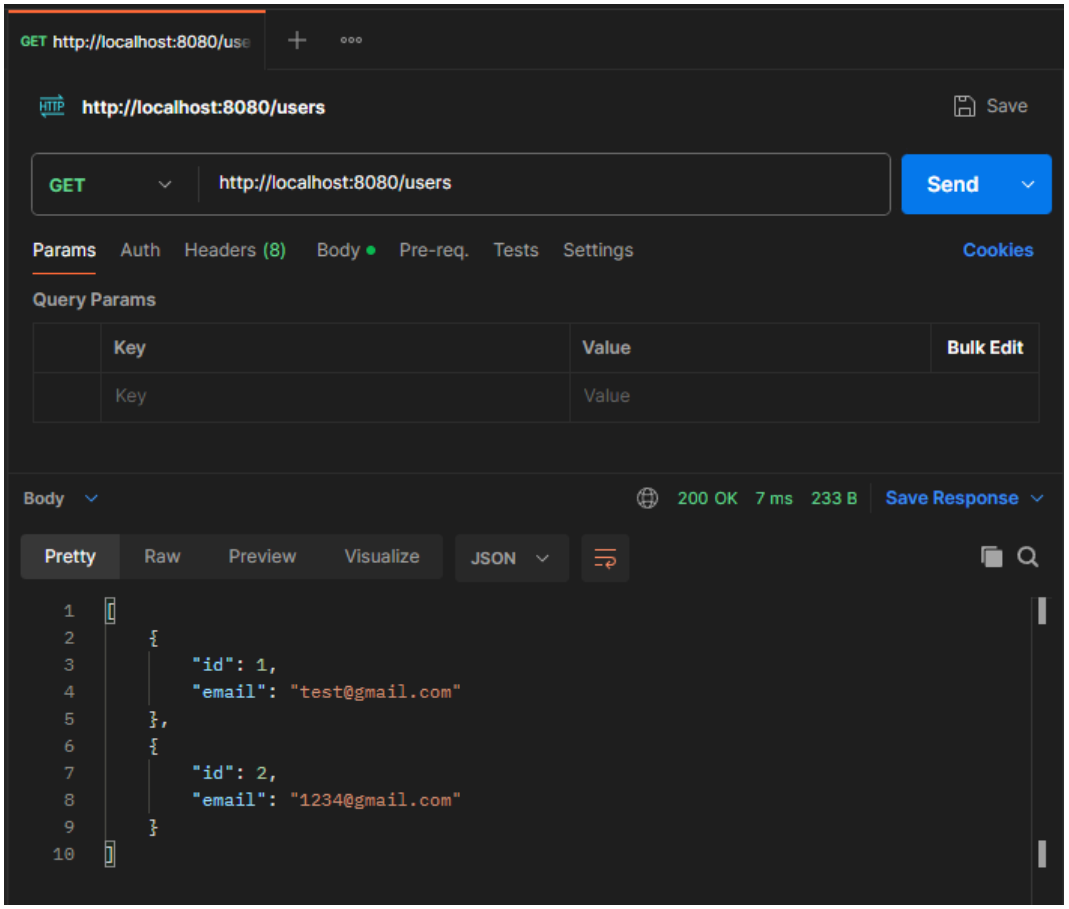
   f. Rebuild your application, Docker image and this time run using Docker Compose (for multi-container applications):

   ```
   ./gradlew clean build
   docker build -t springbootdemo .
   docker-compose up
   ```

   g. Test the new endpoints in Postman



4.4 Testing POST API endpoints for user creation in Postman.



4.5 Testing GET API endpoint for user retrieval in Postman.

5. Developing a Client Application for Bulk Requests (20 mins)

Create a simple Java client application that sends multiple requests in bulk to the Spring Boot API. Implement configurable parameters in the client app to set the number of requests and simulate different traffic loads.

For simplicity, the first step is to reset the DemoApplication to what it was in the beginning, for this we can repeat step 1 and 2, this should only take a few minutes. Once you have the container running and see “hello world” showing again in http://localhost:8080/hello you're good to go!

Now we can simulate multiple requests. This can be done by creating a HttpClientApp.java in the same folder.

Try to come up with how this system could work yourselves first and if you get stuck you can follow along with the code sample:

```
package com.example.demo;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class HttpClientApp {

    private static final String API_URL = "http://localhost:8080/hello";

    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Usage: java HttpClientApp <numberOfRequests>
<threadPoolSize>");
            return;
        }

        int numberOfRequests = Integer.parseInt(args[0]);
        int threadPoolSize = Integer.parseInt(args[1]);

        ExecutorService executor =
Executors.newFixedThreadPool(threadPoolSize);
        for (int i = 0; i < numberOfRequests; i++) {
            executor.submit(() -> sendRequest());
        }
```

```
        executor.shutdown();
    }

    private static void sendRequest() {
        try {
            URL url = new URL(API_URL);
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("GET");

            BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
            String inputLine;
            StringBuilder response = new StringBuilder();

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();

            System.out.println("Response: " + response.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.1 Java Client Application for Sending Bulk Requests and Its Compilation and Execution.

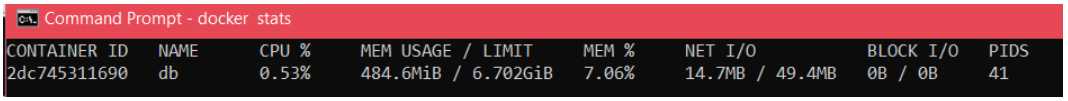
1. To build this use `./gradlew build`
2. Then run `javac -d bin\ src/main/java/com/example/demo/HttpClientApp.java`

   This command compiles the HttpClientApp.java file and stores the resulting compiled .class file(s) in the bin\ directory, keeping the package structure intact.
3. Now step out into the root folder. For me that's a folder called demo and run `java -cp demo/bin com.example.demo.HttpClientApp 100 10`

The JVM looks in the bin directory for the class com.example.client.HttpClientApp. It runs the main method of the HttpClientApp class, passing the arguments "100" and "10". 100 is the number of requests (try increasing this to 1000 and see if this makes any difference in performance. 10 is the thread pool size (for this lab we recommend leaving this the same as a control). We encourage you to play around with these in your own time.

6. Monitoring Docker Resource Usage (20 mins)

Use Docker tools like docker stats to observe CPU, memory, and network usage for the Spring Boot and MySQL containers.



6.1 Using the docker stats Command to Monitor Resource Usage

Monitor how varying traffic loads impact resource consumption.

Analyse the system performance based on different levels of traffic generated by the client app. You can do this through the Docker Desktop. Find your container in the Containers tab. Go to the stats section. Run HttpClient again with different inputs and note the change in Resource Usage.

View the charts in the Stats tab for each container in Docker Desktop.

6.2 Docker Desktop Resource Usage Charts for Spring Boot and MySQL Containers.