Containerization Project

Rawan Afaneh

Implement one of the following ideas, and upload your solution on Schoology before the deadline:

1. Build a containerized microservices data collection and analytics system as shown in Fig.1 You need to write a docker file for each image (service), and docker compose file to run the system.

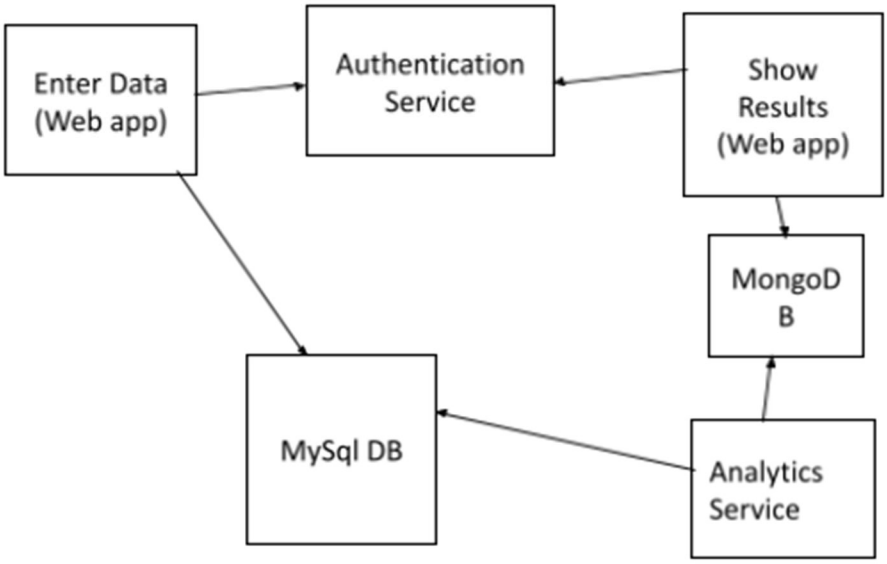

Fig. 1 The Data collection and Analytics System Architecture

First the authenticate :

Controller:

```
package com.example.auth.controller;


import com.example.auth.service.AuthenticationService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class AuthenticationController {

   @Autowired
   private AuthenticationService service ;


   @PostMapping("/authenticate")
   public String handlingData(@RequestBody String userFields){
      String username = userFields.split(",")[0];
      String password = userFields.split(",")[1];
      if (service.isUser(username,password))
         return "valid";
      else return "invalid";
   }

}
```

Explanation:

1. **@RestController**: This annotation marks the class as a controller that handles incoming HTTP requests and returns responses, typically in JSON format.
2. **@Autowired private AuthenticationService service**: This line injects an instance of **AuthenticationService** into the controller. The **@Autowired** annotation enables Spring to automatically instantiate and manage the **AuthenticationService** bean.
3. **@PostMapping("/authenticate")**: This annotation indicates that the following method (**handlingData**) will handle HTTP POST requests sent to the **/authenticate** endpoint.
4. **public String handlingData(@RequestBody String userFields)**: This method takes a single parameter, **userFields**, which represents the data sent in the request body. The **@RequestBody** annotation indicates that the method expects the request body to be bound to this parameter.
5. **String username = userFields.split(",")[0];**: This line splits the **userFields** string using a comma as the delimiter and assigns the first part (username) to the **username** variable.
6. **String password = userFields.split(",")[1];**: Similarly, this line assigns the second part (password) of the **userFields** string to the **password** variable.
7. **if (service.isUser(username, password))**: This conditional statement uses the injected **AuthenticationService** to check if the provided username and password are valid by calling the **isUser** method.
8. **return "valid";**: If the **isUser** method returns **true**, the controller responds with the string "valid", indicating that the credentials are valid.
9. **else return "invalid";**: If the **isUser** method returns **false**, the controller responds with the string "invalid", indicating that the credentials are not valid.

Dao:

```
package com.example.auth.dao;
```

```
import com.example.auth.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;

import java.sql.SQLException;



@Repository
public class UserDao {
   @Autowired
   private JdbcTemplate template;

   private static final class UserMapper implements RowMapper<User>{

      @Override
      public User mapRow(ResultSet rs, int rowNum) throws SQLException {
         User user = new User();
         user.setUsername(rs.getString("username"));
         user.setPassword(rs.getString("password"));
         return user;
      }
   }
   public User getUserByUsername(String username){
      User user =template.queryForObject("SELECT username,password FROM user WHERE username=?"
, new UserMapper() , username);
      return user;
   }

}
```

Explanation:

1. **@Repository**: This annotation marks the class as a Spring Data repository, indicating that it should be considered for Spring's component scanning and treated as a data access object.
2. **@Autowired private JdbcTemplate template**: This line injects an instance of the **JdbcTemplate** class, which provides simplified access to a relational database, into the **UserDao** class. The **JdbcTemplate** is a Spring class that helps with database operations.
3. **private static final class UserMapper implements RowMapper<User>**: This is an inner class named **UserMapper** that implements the **RowMapper** interface. It's responsible for mapping

database rows to **User** objects. The **RowMapper** interface provides a way to customize how database rows are converted into Java objects.

4. **public User mapRow(ResultSet rs, int rowNum) throws SQLException**: This method is part of the **UserMapper** inner class. It is implemented from the **RowMapper** interface and defines how a row from the database result set is mapped to a **User** object.
5. **User user = new User();**: This line creates a new **User** object that will hold the data from the database.
6. **user.setUsername(rs.getString("username"));**: This line sets the username of the **User** object by extracting it from the **ResultSet** obtained from the database.
7. **user.setPassword(rs.getString("password"));**: Similarly, this line sets the password of the **User** object from the **ResultSet**.
8. **public User getUserByUsername(String username)**: This method is used to retrieve a **User** object from the database based on the provided username.
9. **User user = template.queryForObject(...)**: This line uses the **JdbcTemplate** to query the database for a user based on the provided SQL query. The **queryForObject** method returns a single result mapped to the specified class (**User** in this case).
10. **"SELECT username, password FROM user WHERE username = ?"**: This SQL query fetches the **username** and **password** columns from the **user** table where the **username** matches the provided parameter.
11. **new UserMapper()**: This is the instance of the **UserMapper** class that defines how the result set should be mapped to a **User** object.
12. **username**: This parameter is used to specify the value for the placeholder (**?**) in the SQL query.
13. **return user;**: If a user with the provided username is found in the database, the method returns the corresponding **User** object. If no user is found, it may return **null** or you can handle this situation as needed.
14. **catch (EmptyResultDataAccessException e)**: This block catches the exception that might be thrown if the **queryForObject** method doesn't find any results in the database.
15. **return null;**: In the case that no user is found, the method returns **null**.

Model:

```
package com.example.auth.model;

public class User {
   private String username;
   private String password;

   public User() {
   }
   public User(String username, String password) {
      this.username = username;
      this.password = password;
   }

   public String getUsername() {
      return username;
```

```
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Explanation:

1. **private String username; and private String password;**: These are private instance variables that hold the username and password of a user.
2. **public User()**: This is a default constructor with no parameters. It allows you to create a new **User** object without providing initial values for the **username** and **password**.
3. **public User(String username, String password)**: This is a parameterized constructor that allows you to create a **User** object and set its **username** and **password** values during object creation.
4. **public String getUsername()**: This method is a getter for retrieving the **username** of the user.
5. **public void setUsername(String username)**: This method is a setter for setting the **username** of the user.
6. **public String getPassword()**: This method is a getter for retrieving the **password** of the user.
7. **public void setPassword(String password)**: This method is a setter for setting the **password** of the user.

Service:

```
package com.example.auth.service;


import com.example.auth.dao.UserDao;
import com.example.auth.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;

import java.util.HashMap;
import java.util.Map;
@Service
public class AuthenticationService {
```

```
    @Autowired
    private UserDao dao ;


    public AuthenticationService(){

    }
    public boolean isUser(String username , String password){

        try {
            User user = dao.getUserByUsername(username);
            if (password.equals(user.getPassword()))
                return true;
            return false;
        }
        catch (EmptyResultDataAccessException e){
            return false;
        }
    }

}
```

Explanation:

1. **@Service**: This annotation marks the class as a Spring service, indicating that it holds the business logic of the application.
2. **@Autowired private UserDao dao**: This field is autowired with an instance of **UserDao**. This means that Spring will inject an instance of the **UserDao** class into this field, allowing you to use the data access methods defined in **UserDao**.
3. **public AuthenticationService() { }**: This is the default constructor of the **AuthenticationService** class. It is empty in your example.
4. **public boolean isUser(String username, String password)**: This method checks whether a user with the given username and password exists in the system.
5. **User user = dao.getUserByUsername(username);**: This line retrieves a **User** object from the database using the **getUserByUsername** method of the injected **UserDao** instance.
6. **if (password.equals(user.getPassword())) { return true; }**: This block compares the provided password with the password stored in the retrieved **User** object. If they match, the method returns **true**, indicating successful authentication.
7. **return false;**: If the passwords don't match, the method returns **false**.
8. **catch (EmptyResultDataAccessException e)**: This block catches the exception that might be thrown if no user is found in the database for the given username.
9. **return false;**: If no user is found (i.e., the **EmptyResultDataAccessException** is caught), the method returns **false**.

AuthenticationApplication:

```
package com.example.auth;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AuthenticationApplication {

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

Explanation:

1. **package com.example.auth;**: This line indicates the package name where the class belongs.
2. **import org.springframework.boot.SpringApplication;**: This import statement brings in the **SpringApplication** class from the Spring Boot framework, which is used to bootstrap and run the Spring Boot application.
3. **import org.springframework.boot.autoconfigure.SpringBootApplication;**: This import statement brings in the **SpringBootApplication** annotation, which is used to declare the class as the main application class and to enable Spring Boot's auto-configuration features.
4. **@SpringBootApplication**: This annotation is applied to the class to indicate that it is the main Spring Boot application class. It combines three annotations: **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**.
5. **public class AuthenticationApplication { ... }**: This is the main class of your Spring Boot application. The class name is **AuthenticationApplication**.
6. **public static void main(String[] args) { ... }**: This is the **main** method, which serves as the entry point of your application.
7. **SpringApplication.run(AuthenticationApplication.class, args);**: This line of code starts the Spring Boot application. It uses the **run** method of **SpringApplication**, passing in the class name **AuthenticationApplication.class** as the primary source and the command-line arguments **args**.

Application.proparties :

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://db/dockerr
spring.datasource.username=root
spring.datasource.password=rawan
```

dockerfile:

```
FROM maven:3.8.3-openjdk-17 AS build
COPY src /home/app/src
COPY pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package

FROM openjdk:17-alpine
```

```
COPY --from=build /home/app/target/AuthenticationService-0.0.1-SNAPSHOT.jar
/usr/local/lib/AuthenticationService.jar
ENTRYPOINT ["java","-jar","/usr/local/lib/AuthenticationService.jar"]
```

For enter data container:

Controller:

```
package com.example.enterdata.controller;


import com.example.enterdata.Data.Data;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

import javax.servlet.http.HttpServletRequest;

@Controller
public class DataEntryController {
   @Autowired
   private JdbcTemplate template;

   @PostMapping("/show_data_form")
   public String viewDataForm(Model model){
      model.addAttribute("dataObject" , new Data());
      return"enter-data";
   }

   @PostMapping("/submit_datas")
   public String handlingGrades(@ModelAttribute Data dataObject , HttpServletRequest request){
      String data = dataObject.getdata();
      String[] arr = data.split(",");
      Data[] users = new Data[arr.length];
      for (int i =0 ; i<users.length ; i++){
         users[i] = new Data();
         users[i].generateUser(arr[i]);
      }
      for (Data d : users){
         template.update("INSERT INTO Datas values(?,?)",d.getUserName() , d.getData());
      }
```

```
        return"forward:/microservice";
    }



}
```

Explanation:

1. **@Controller**: This annotation marks the class as a Spring MVC controller, indicating that it handles HTTP requests and returns views.
2. **@Autowired private JdbcTemplate template**: This field is autowired with an instance of **JdbcTemplate**. This allows you to use the **JdbcTemplate** to interact with the database.
3. **@PostMapping("/show_data_form")**: This annotation specifies that the following method (**viewDataForm**) handles HTTP POST requests to the **/show_data_form** endpoint.
4. **public String viewDataForm(Model model)**: This method handles the request for showing the data entry form. It takes a **Model** parameter to add attributes for rendering the view.
5. **model.addAttribute("dataObject", new Data());**: This line adds a new **Data** object as a model attribute with the name "dataObject". This object will be used to bind form data.
6. **return "enter-data";**: This line returns the view name "enter-data", which will be used to render the data entry form.
7. **@PostMapping("/submit_datas")**: This annotation specifies that the following method (**handlingGrades**) handles HTTP POST requests to the **/submit_datas** endpoint.
8. **public String handlingGrades(@ModelAttribute Data dataObject, HttpServletRequest request)**: This method handles the submission of data. It uses the **@ModelAttribute** annotation to bind the form data to a **Data** object.
9. **String data = dataObject.getdata();**: This line retrieves the data from the submitted form using the **getdata()** method of the **Data** object.
10. **String[] arr = data.split(",");**: This line splits the submitted data into an array of strings using a comma (**,**) as the delimiter.
11. **Data[] users = new Data[arr.length];**: This line creates an array of **Data** objects to hold the user data.
12. **Loop for creating Data objects and generating user data**: This loop iterates over the array of data strings, creating **Data** objects and generating user data using the **generateUser** method.
13. **Loop for inserting user data into the database**: This loop iterates over the array of **Data** objects and uses the **JdbcTemplate** to insert user data into the database.
14. **return "forward:/microservice";**: This line forwards the request to the **/microservice** endpoint. It's worth noting that this endpoint is not defined in the provided code, so it would need to be defined elsewhere in your application.

```
package com.example.enterdata.controller;


import com.example.enterdata.user.User;
```

```
import com.google.gson.Gson;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.StreamingHttpOutputMessage;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.net.*;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.http.HttpResponse.*;

@Controller
public class LoginController {


   @Autowired
   private JdbcTemplate template;

   @GetMapping("/")
   public String showHome(){
      return "home";
   }

   @GetMapping("/login")
   public String showLogin(Model model){
      model.addAttribute("user" , new User());
      return "login";
   }

   @PostMapping("/submit")
   public String handleForm(@ModelAttribute User user , Model model) throws IOException,
URISyntaxException, InterruptedException {
      String name = user.getName();
      String password = user.getPassword();
```

```
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request =HttpRequest.newBuilder()
              .uri(new URI("http://authentication:8080/authenticate"))
              .POST(HttpRequest.BodyPublishers.ofString(name+","+password))
              .build();

        HttpResponse<String> response=client.send(request ,BodyHandlers.ofString());
        if (!response.body().equals("valid")){
           model.addAttribute("message" , response.body());
           return "login";
        }

        return "forward:/show_data_form";
     }
}
```

Explanation:

1. **@GetMapping("/")**: This annotation specifies that the **showHome** method handles HTTP GET requests to the root URL ("/").
2. **public String showHome() { ... }**: This method returns the view name "home" to render the home page.
3. **@GetMapping("/login")**: This annotation specifies that the **showLogin** method handles HTTP GET requests to the "/login" URL.
4. **public String showLogin(Model model) { ... }**: This method prepares the login page by adding an empty **User** object as a model attribute and returns the view name "login".
5. **@PostMapping("/submit")**: This annotation specifies that the **handleForm** method handles HTTP POST requests to the "/submit" URL.
6. **public String handleForm(@ModelAttribute User user, Model model) throws IOException, URISyntaxException, InterruptedException { ... }**: This method handles the form submission for user authentication.
7. **String name = user.getName();** and **String password = user.getPassword();**: These lines retrieve the name and password from the submitted form data.
8. **HttpClient client = HttpClient.newHttpClient();**: This line creates an instance of **HttpClient** for making HTTP requests.
9. **HttpRequest request = HttpRequest.newBuilder() ... .build();**: This section creates an **HttpRequest** to authenticate the user by sending a POST request to the "/authenticate" URL using the provided name and password.
10. **HttpResponse<String> response = client.send(request, BodyHandlers.ofString());**: This line sends the authentication request using the **HttpClient** and receives the response.
11. **if (!response.body().equals("valid")) { ... }**: This condition checks if the authentication response is not equal to "valid".
12. **model.addAttribute("message", response.body());**: This line adds an error message from the authentication response to the model.
13. **return "login";**: This line returns the view name "login" to render the login page with the error message.

14. **return "forward:/show_data_form";**: If authentication is successful, this line forwards the request to the "/show_data_form" endpoint to display the data entry form.

```
package com.example.enterdata.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PostMapping;

import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

@Controller
public class MicroServicesController {

   @Autowired
   private JdbcTemplate template;

   @PostMapping("/microservice")
   public String useOtherServices(Model model) throws URISyntaxException, IOException,
InterruptedException {


      HttpClient client = HttpClient.newHttpClient();
      HttpRequest request =HttpRequest.newBuilder()
            .uri(new URI("http://analytics:8080/analytics"))
            .POST(HttpRequest.BodyPublishers.ofString("doAnalytics"))
            .build();
      HttpResponse<String> response=client.send(request , HttpResponse.BodyHandlers.ofString());


      if (response.body().equals("incomplete")){
         model.addAttribute("message" , "incomplete");
         return "result";
      }

      HttpClient client1 = HttpClient.newHttpClient();
```

```
        HttpRequest request1 =HttpRequest.newBuilder()
                .uri(new URI("http://showresult:8080/showresult"))
                .POST(HttpRequest.BodyPublishers.ofString(String.valueOf(1)))
                .build();
        HttpResponse<String> response1=client.send(request1 , HttpResponse.BodyHandlers.ofString());
        model.addAttribute("message" , response1.body());
        return "result";

    }
}
```

Explanation:

1. **@PostMapping("/microservice")**: This annotation specifies that the **useOtherServices** method handles HTTP POST requests to the "/microservice" URL.
2. **public String useOtherServices(Model model) throws URISyntaxException, IOException, InterruptedException { ... }**: This method handles interactions with other microservices and displays the result.
3. **HttpClient client = HttpClient.newHttpClient();**: This line creates an instance of **HttpClient** for making HTTP requests.
4. **HttpRequest request = HttpRequest.newBuilder() ... .build();**: This section creates an **HttpRequest** to interact with the "analytics" microservice by sending a POST request with the content "doAnalytics".
5. **HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());**: This line sends the analytics request using the **HttpClient** and receives the response.
6. **if (response.body().equals("incomplete")) { ... }**: This condition checks if the analytics response is equal to "incomplete".
7. **model.addAttribute("message", "incomplete");**: This line adds an "incomplete" message to the model.
8. **return "result";**: This line returns the view name "result" to render the result page with the "incomplete" message.
9. **HttpClient client1 = HttpClient.newHttpClient();**: This line creates another instance of **HttpClient** for making another HTTP request.
10. **HttpRequest request1 = HttpRequest.newBuilder() ... .build();**: This section creates an **HttpRequest** to interact with the "showresult" microservice by sending a POST request with the content "1" (assuming it's some kind of identifier).
11. **HttpResponse<String> response1 = client.send(request1, HttpResponse.BodyHandlers.ofString());**: This line sends the "showresult" request and receives the response.
12. **model.addAttribute("message", response1.body());**: This line adds the received message from the "showresult" microservice to the model.
13. **return "result";**: This line returns the view name "result" to render the result page with the received message.

Data:

```
package com.example.enterdata.Data;

import java.util.ArrayList;

public class Data {
   private String data;
   private String UserName;
   private double Data;


   public Data(){

   }
   public void generateUser(String s){
      UserName = s.split(":")[0];
      Data = Double.parseDouble(s.split(":")[1]);
   }


   public String getUserName() {
      return UserName;
   }

   public void setUserName(String UserName) {
      this.UserName = UserName;
   }

   public double getData() {
      return Data;
   }

   public void setData(double Data) {
      this.Data = Data;
   }

   public String getdata() {
      return data;
   }

   public void setdata(String data) {
      this.data = data;
   }
}
```

Explanation:

1. **public Data() { }: This is the default constructor of the Data class. It is empty in your example.**
2. **public void generateUser(String s) { ... }: This method takes a string s as input and generates user data by splitting the string into parts using the colon (:) delimiter. It assigns the first part to the UserName and the second part (converted to a double) to the Data.**
3. **public String getUserName() { ... }: This method is a getter for retrieving the UserName of the user.**
4. **public void setUserName(String UserName) { ... }: This method is a setter for setting the UserName of the user.**
5. **public double getData() { ... }: This method is a getter for retrieving the Data value of the user.**
6. **public void setData(double Data) { ... }: This method is a setter for setting the Data value of the user.**
7. **public String getdata() { ... }: This method is a getter for retrieving the data string.**
8. **public void setdata(String data) { ... }: This method is a setter for setting the data string.**

User:

```
package com.example.enterdata.user;

public class User {
   private String name;
   private String password;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getPassword() {
      return password;
   }

   public void setPassword(String password) {
      this.password = password;
   }
}
```

Explanation:

1. **private String name;** and **private String password;**: These are private instance variables that hold the user's name and password.
2. **public String getName() { ... }**: This method is a getter for retrieving the user's name.
3. **public void setName(String name) { ... }**: This method is a setter for setting the user's name.
4. **public String getPassword() { ... }**: This method is a getter for retrieving the user's password.

5. **public void setPassword(String password) { ... }**: This method is a setter for setting the user's password.

EnterDataApplication:

```
package com.example.enterdata;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class EnterDataApplication {

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

}
```

Explanation:

1. **package com.example.enterdata;**: This line indicates the package name where the class belongs.
2. **import org.springframework.boot.SpringApplication;**: This import statement brings in the **SpringApplication** class from the Spring Boot framework, which is used to bootstrap and run the Spring Boot application.
3. **import org.springframework.boot.autoconfigure.SpringBootApplication;**: This import statement brings in the **SpringBootApplication** annotation, which is used to declare the class as the main application class and to enable Spring Boot's auto-configuration features.
4. **@SpringBootApplication**: This annotation is applied to the class to indicate that it is the main Spring Boot application class. It combines three annotations: **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**.
5. **public class EnterDataApplication { ... }**: This is the main class of your Spring Boot application. The class name is **EnterDataApplication**.
6. **public static void main(String[] args) { ... }**: This is the **main** method, which serves as the entry point of your application.
7. **SpringApplication.run(EnterDataApplication.class, args);**: This line of code starts the Spring Boot application. It uses the **run** method of **SpringApplication**, passing in the class name **EnterDataApplication.class** as the primary source and the command-line arguments **args**.

Application.properties :

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://db/dockerr
```

```
spring.datasource.username=root
spring.datasource.password=Rawan


templete:
```

enter data :

```
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org">
<head>
   <meta charset="UTF-8">
   <title>Title</title>
</head>
<body>

<form method="POST" action="/submit_datas" th:object="${dataObject}">
   <label for="entry">enter the data</label>
   <input id="entry" type="text"  th:field="*{data}">
   <input type="submit" value="Submit" />
</form>
</body>
</html>
```

Explanation:

1. **<!DOCTYPE html>**: This line declares the document type and version of the HTML document.
2. **<html lang="en" xmlns:th="https://www.thymeleaf.org">**: This opening **<html>** tag specifies the language of the document and includes the XML namespace for Thymeleaf.
3. **<head>**: The **<head>** section contains metadata about the document, such as character encoding and title.
4. **<meta charset="UTF-8">**: This **<meta>** tag sets the character encoding of the document to UTF-8.
5. **<title>Title</title>**: This **<title>** tag sets the title of the HTML document (replace "Title" with your desired title).
6. **<body>**: The **<body>** section contains the visible content of the webpage.
7. **<form method="POST" action="/submit_datas" th:object="${dataObject}">**: This **<form>** tag defines a form element. It uses the **POST** method and specifies the action attribute as "/submit_datas" to indicate where the form data should be sent. The **th:object="${dataObject}"** attribute binds the form data to the **dataObject** model attribute.
8. **<label for="entry">enter the data</label>**: This **<label>** tag creates a label for the input field with the id "entry". The **for** attribute associates the label with the input field.
9. **<input id="entry" type="text" th:field="*{data}">**: This **<input>** tag creates a text input field. The **id** attribute is used to uniquely identify the input element. The **type** attribute is set to "text"

to create a text input. The **th:field="*{data}"** attribute binds the input field to the "data" property of the **dataObject**.

10. **<input type="submit" value="Submit" />**: This **<input>** tag creates a submit button that users can click to submit the form.
11. **</form>**: This closing **</form>** tag ends the form element.
12. **</body>**: This closing **</body>** tag ends the **<body>** section.
13. **</html>**: This closing **</html>** tag ends the HTML document.

Home page :

```
<!DOCTYPE html>
<html lang="en"  xmlns:th="https://www.thymeleaf.org">
<head>
   <meta charset="UTF-8">
   <title>Title</title>
</head>
<body>

<h1> <a href="/login">login</a></h1>

</body>
</html>
```

Explanation:

1. **<!DOCTYPE html>**: This line declares the document type and version of the HTML document.
2. **<html lang="en" xmlns:th="https://www.thymeleaf.org">**: This opening **<html>** tag specifies the language of the document and includes the XML namespace for Thymeleaf.
3. **<head>**: The **<head>** section contains metadata about the document, such as character encoding and title.
4. **<meta charset="UTF-8">**: This **<meta>** tag sets the character encoding of the document to UTF-8.
5. **<title>Title</title>**: This **<title>** tag sets the title of the HTML document (replace "Title" with your desired title).
6. **<body>**: The **<body>** section contains the visible content of the webpage.
7. **<h1> login</h1>**: This line creates an **<h1>** (header level 1) element that contains a link to the "/login" page. The **<a>** (anchor) element creates a hyperlink with the text "login." When users click on the link, they will be directed to the login page.
8. **</body>**: This closing **</body>** tag ends the **<body>** section.
9. **</html>**: This closing **</html>** tag ends the HTML document.

Login page :

```
<!DOCTYPE html>
<html lang="en"  xmlns:th="https://www.thymeleaf.org">
<head>
   <meta charset="UTF-8">
   <title>Title</title>
</head>
<body>
<span th:text="${message}"/>

<form method="POST" action="/submit" th:object="${user}">
   <label for="name">Name</label>
   <input type="text" id="name" th:field="*{name}"/>

   <label for="password">Password</label>
   <input type="password" id="password" th:field="*{password}" />
   <input type="submit" value="Submit" />
</form>
</body>
</html>
```

Explanation:

1. **<!DOCTYPE html>**: This line declares the document type and version of the HTML document.
2. **<html lang="en" xmlns:th="https://www.thymeleaf.org">**: This opening **<html>** tag specifies the language of the document and includes the XML namespace for Thymeleaf.
3. **<head>**: The **<head>** section contains metadata about the document, such as character encoding and title.
4. **<meta charset="UTF-8">**: This **<meta>** tag sets the character encoding of the document to UTF-8.
5. **<title>Title</title>**: This **<title>** tag sets the title of the HTML document (replace "Title" with your desired title).
6. **<body>**: The **<body>** section contains the visible content of the webpage.
7. **<span th:text="${message}"/>**: This line displays the value of the **${message}** variable using Thymeleaf's **th:text** attribute. The actual message content will be inserted at this location.
8. **<form method="POST" action="/submit" th:object="${user}">**: This **<form>** tag defines a form element. It uses the **POST** method and specifies the action attribute as "/submit" to indicate where the form data should be sent. The **th:object="${user}"** attribute binds the form data to the **user** model attribute.
9. **<label for="name">Name</label>**: This **<label>** tag creates a label for the input field with the id "name". The **for** attribute associates the label with the input field.
10. **<input type="text" id="name" th:field="*{name}" />**: This **<input>** tag creates a text input field for the user's name. The **type** attribute is set to "text". The **id** attribute is used to uniquely identify the input element. The **th:field="*{name}"** attribute binds the input field to the "name" property of the **user** object.

11. **<label for="password">Password</label>**: This **<label>** tag creates a label for the input field with the id "password". The **for** attribute associates the label with the input field.
12. **<input type="password" id="password" th:field="*{password}" />**: This **<input>** tag creates a password input field for the user's password. The **type** attribute is set to "password". The **id** attribute is used to uniquely identify the input element. The **th:field="*{password}"** attribute binds the input field to the "password" property of the **user** object.
13. **<input type="submit" value="Submit" />**: This **<input>** tag creates a submit button that users can click to submit the form.
14. **</form>**: This closing **</form>** tag ends the form element.
15. **</body>**: This closing **</body>** tag ends the **<body>** section.
16. **</html>**: This closing **</html>** tag ends the HTML document.

Result page :

```
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org">
<head>
   <meta charset="UTF-8">
   <title>Title</title>
</head>
<body>
<h1><span th:text="${message}"/>
</h1>
</body>
</html>
```

Explanation:

1. **<!DOCTYPE html>**: This line declares the document type and version of the HTML document.
2. **<html lang="en" xmlns:th="https://www.thymeleaf.org">**: This opening **<html>** tag specifies the language of the document and includes the XML namespace for Thymeleaf.
3. **<head>**: The **<head>** section contains metadata about the document, such as character encoding and title.
4. **<meta charset="UTF-8">**: This **<meta>** tag sets the character encoding of the document to UTF-8.
5. **<title>Title</title>**: This **<title>** tag sets the title of the HTML document (replace "Title" with your desired title).
6. **<body>**: The **<body>** section contains the visible content of the webpage.
7. **<h1><span th:text="${message}"/></h1>**: This line creates an **<h1>** (header level 1) element that contains a **<span>** element. The **th:text="${message}"** attribute of the **** element indicates that the content of the **** should be replaced with the value of the **${message}** variable.
8. **</body>**: This closing **</body>** tag ends the **<body>** section.
9. **</html>**: This closing **</html>** tag ends the HTML document.

Dockerfile:

```
FROM maven:3.8.3-openjdk-17 AS build
COPY src /home/app/src
COPY pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package

FROM openjdk:17-alpine
COPY --from=build /home/app/target/webApp-0.0.1-SNAPSHOT.jar /usr/local/lib/webApp.jar
ENTRYPOINT ["java","-jar","/usr/local/lib/webApp.jar"]
```

The analytics container

Controller :

```
package com.example.analytics.Controller;


import com.example.analytics.service.AnalyticsService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class AnalyticsController {

   @Autowired
   private AnalyticsService service;


   @PostMapping("/analytics")
   public String doAnalytics(@RequestBody String command){
      if (!command.equals("doAnalytics"))
         return "incomplete";
      service.doService();
      return "completed"}}
```

Explanation:

1. **package com.example.analytics.Controller;**: This line specifies the package where the **AnalyticsController** class belongs.
2. **import com.example.analytics.service.AnalyticsService;**: This import statement brings in the **AnalyticsService** class from the **com.example.analytics.service** package.

3. **import org.springframework.beans.factory.annotation.Autowired;**: This import statement brings in the **Autowired** annotation from the Spring Framework, which is used for automatic dependency injection.
4. **import org.springframework.web.bind.annotation.GetMapping;**, **import org.springframework.web.bind.annotation.PostMapping;**, **import org.springframework.web.bind.annotation.RequestBody;**, **import org.springframework.web.bind.annotation.RestController;**: These import statements bring in annotations for defining RESTful APIs and handling HTTP requests in Spring.
5. **@RestController**: This annotation is applied to the class to indicate that it is a Spring MVC controller that handles RESTful API requests.
6. **public class AnalyticsController { ... }**: This is the class definition for the **AnalyticsController** class.
7. **@Autowired**: This annotation is used to inject the **AnalyticsService** bean into the controller.
8. **@PostMapping("/analytics")**: This annotation maps the **doAnalytics** method to handle HTTP POST requests with the path "/analytics".
9. **public String doAnalytics(@RequestBody String command) { ... }**: This is the **doAnalytics** method that processes the analytics request. It takes a **String** parameter **command** as the request body.
10. **if (!command.equals("doAnalytics"))**: This condition checks if the **command** received is not equal to the expected command "doAnalytics". If so, it returns the string "incomplete".
11. **service.doService();**: This line calls the **doService** method of the **AnalyticsService** bean to perform the analytics task.
12. **return "completed";**: If the analytics task is successfully completed, this line returns the string "completed".

Dao:

```
package com.example.analytics.Dao;

import com.example.analytics.model.DataStatistics;
import com.example.analytics.model.UserData;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

@Repository
public class AnalyticsDao {
```

```
    @Autowired
    private JdbcTemplate template;

    private static final class UserDataMapper implements RowMapper<UserData> {
        @Override
        public UserData mapRow(ResultSet rs, int rowNum) throws SQLException {
            UserData user = new UserData();
            user.setData(rs.getDouble("data"));
            user.setUserName(rs.getString("user_name"));
            return user;
        }
    }




    public List<UserData> readGrades(){
        return template.query("SELECT * FROM datas" ,new UserDataMapper() );
    }

    public void writeStatistics(DataStatistics dataStatistics){

        template.update("INSERT INTO statistics values(?,?,?,?)" , 1 , dataStatistics.getMin() ,
dataStatistics.getMax(),dataStatistics.getAvg());
    }

}
```

Explanation:

1. **package com.example.analytics.Dao;**: This line specifies the package where the **AnalyticsDao** class belongs.
2. **import com.example.analytics.model.DataStatistics;**, **import com.example.analytics.model.UserData;**: These import statements bring in the **DataStatistics** and **UserData** classes, which are used as model objects for analytics data.
3. **import org.springframework.beans.factory.annotation.Autowired;**: This import statement brings in the **Autowired** annotation from the Spring Framework, which is used for automatic dependency injection.
4. **import org.springframework.jdbc.core.JdbcTemplate;**, **import org.springframework.jdbc.core.RowMapper;**: These import statements bring in classes from Spring's JDBC framework for database interactions.
5. **import org.springframework.stereotype.Repository;**: This import statement brings in the **Repository** annotation, which indicates that the class is a Spring Data Access Object (DAO).
6. **@Repository**: This annotation is applied to the class to indicate that it is a Spring Data Access Object.
7. **public class AnalyticsDao { ... }**: This is the class definition for the **AnalyticsDao** class.

8. **@Autowired**: This annotation is used to inject the **JdbcTemplate** bean into the DAO.
9. **private static final class UserDataMapper implements RowMapper<UserData> { ... }**: This nested class defines a **RowMapper** for mapping **UserData** objects from a **ResultSet**. It implements the **RowMapper** interface and overrides the **mapRow** method to create and populate a **UserData** object.
10. **public List<UserData> readGrades() { ... }**: This method queries the database using the **JdbcTemplate** to retrieve a list of **UserData** objects from the "datas" table. It uses the **UserDataMapper** to map the rows from the result set to **UserData** objects.
11. **public void writeStatistics(DataStatistics dataStatistics) { ... }**: This method writes statistics data into the "statistics" table using the **JdbcTemplate**. It inserts a new row with statistics values such as min, max, and average.

Model:

```
package com.example.analytics.model;

public class DataStatistics{
   private double min;
   private double max;
   private double avg;

   public DataStatistics() {
   }

   public DataStatistics(double min, double max, double avg) {
      this.min = min;
      this.max = max;
      this.avg = avg;
   }

   public double getMin() {
      return min;
   }

   public void setMin(double min) {
      this.min = min;
   }

   public double getMax() {
      return max;
   }

   public void setMax(double max) {
      this.max = max;
   }
```

```
    public double getAvg() {
        return avg;
    }

    public void setAvg(double avg) {
        this.avg = avg;
    }
}
```

And:

```
package com.example.analytics.model;
public class UserData {
    private String UserName;
    private double Data;

    public UserData() {
    }

    public UserData(String UserName, double Data) {
        this.UserName = UserName;
        this.Data = Data;
    }

    public String getUserName() {
        return UserName;
    }

    public void setUserName(String UserName) {
        this.UserName = UserName;
    }

    public double getData() {
        return Data;
    }

    public void setData(double Data) {
        this.Data = Data;
    }
}
```

Service:

```
package com.example.analytics.service;
```

```
import com.example.analytics.Dao.AnalyticsDao;
import com.example.analytics.model.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class AnalyticsService {
   @Autowired
   private AnalyticsDao dao;


   public void doService(){
      List<UserData> data = dao.readGrades();
      DataStatistics statistics = getStatistics(data);
      dao.writeStatistics(statistics);
   }

   private DataStatistics getStatistics(List<UserData> data){
      double min = Double.MAX_VALUE;
      double max = Double.MIN_VALUE;
      double avg = 0;

      for (UserData Data : data){
         if (Data.getData()<min)
            min=Data.getData();
         if (Data.getData()>max)
            max=Data.getData();
         avg+=Data.getData();
      }
      return new DataStatistics(min , max , avg/data.size());
   }


}
```

Explanation:

1. **package com.example.analytics.service;**: This line specifies the package where the **AnalyticsService** class belongs.
2. **import com.example.analytics.Dao.AnalyticsDao;**: This import statement brings in the **AnalyticsDao** class, which is used to interact with the database.

3. **import com.example.analytics.model.*;**: This import statement brings in all the model classes (**DataStatistics** and **UserData**) from the **com.example.analytics.model** package.
4. **import org.springframework.beans.factory.annotation.Autowired;**: This import statement brings in the **Autowired** annotation from the Spring Framework, which is used for automatic dependency injection.
5. **import org.springframework.stereotype.Service;**: This import statement brings in the **Service** annotation, which indicates that the class is a Spring service component.
6. **@Service**: This annotation is applied to the class to indicate that it is a Spring service.
7. **public class AnalyticsService { ... }**: This is the class definition for the **AnalyticsService** class.
8. **@Autowired**: This annotation is used to inject the **AnalyticsDao** bean into the service.
9. **public void doService() { ... }**: This method is the main service logic that performs the analytics tasks. It first retrieves user data from the database using the **readGrades** method from the DAO. Then, it calculates the statistics using the **getStatistics** method and writes the calculated statistics to the database using the **writeStatistics** method.
10. **private DataStatistics getStatistics(List<UserData> data) { ... }**: This private method calculates statistics (minimum, maximum, and average) based on the provided list of user data (**UserData** objects). It iterates through the data to find the minimum and maximum values and calculates the average. It then creates a new **DataStatistics** object with the calculated values and returns it.

AnalyticsApplication:

```
package com.example.analytics;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AnalyticsApplication {

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

}
```

Application.probabilities

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://db/dockerr
spring.datasource.username=root
spring.datasource.password=rawan
```

dockerfile:

```
FROM maven:3.8.3-openjdk-17 AS build
COPY src /home/app/src
COPY pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package

FROM openjdk:17-alpine
COPY --from=build /home/app/target/analytics-0.0.1-SNAPSHOT.jar /usr/local/lib/analytics.jar
ENTRYPOINT ["java","-jar","/usr/local/lib/analytics.jar"]
```

Show results container :

Model:

```
package com.example.showresults.model;

public class DataStatistics {
   private double min;
   private double max;
   private double avg;

   public DataStatistics() {
   }

   public DataStatistics(double min, double max, double avg) {
      this.min = min;
      this.max = max;
      this.avg = avg;
   }

   public double getMin() {
      return min;
   }

   public void setMin(double min) {
      this.min = min;
   }

   public double getMax() {
      return max;
   }

   public void setMax(double max) {
      this.max = max;
   }
```

```
    public double getAvg() {
        return avg;
    }

    public void setAvg(double avg) {
        this.avg = avg;
    }
    @Override
    public String toString(){
        return"Minimum data: "+min +"\ndata: "+max +"\nAverage: "+avg;
    }
}
```

Explanation:

1. **package com.example.showresults.model;**: This line specifies the package where the **DataStatistics** class belongs.
2. **public class DataStatistics { ... }**: This is the class definition for the **DataStatistics** class.
3. **private double min;**, **private double max;**, **private double avg;**: These private instance variables represent the minimum, maximum, and average statistical values.
4. **public DataStatistics() { }**: This is a default constructor for the **DataStatistics** class. It initializes the object without setting any values for the statistical properties.
5. **public DataStatistics(double min, double max, double avg) { ... }**: This constructor takes three arguments (**min**, **max**, and **avg**) and initializes the **DataStatistics** object with these values.
6. **public double getMin() { ... }**: This method is a getter for the **min** property.
7. **public void setMin(double min) { ... }**: This method is a setter for the **min** property.
8. **public double getMax() { ... }**: This method is a getter for the **max** property.
9. **public void setMax(double max) { ... }**: This method is a setter for the **max** property.
10. **public double getAvg() { ... }**: This method is a getter for the **avg** property.
11. **public void setAvg(double avg) { ... }**: This method is a setter for the **avg** property.
12. **@Override** and **public String toString() { ... }**: This is an overridden **toString** method that returns a formatted string representation of the **DataStatistics** object. It includes the minimum, maximum, and average values in the output string.

ShowResultsApplication:

```
package com.example.showresults;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ShowResultsApplication {

    public static void main(String[] args) {
```

```
        SpringApplication.run(ShowResultsApplication.class, args);
    }

}
```

ShowResultsController:

```
package com.example.showresults;

import com.example.showresults.model.DataStatistics;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.sql.ResultSet;
import java.sql.SQLException;

@RestController
public class ShowResultsController {


    @Autowired
    private JdbcTemplate template;

    @PostMapping("/showresult")
    public String showResult(@RequestBody String id ){
        DataStatistics statistics = template.queryForObject("SELECT min , max , avg FROM statistics WHERE
id=?" , new DataStatisticsMapper() , id);
        return statistics.toString();
    }



    private static final class DataStatisticsMapper implements RowMapper<DataStatistics> {
        @Override
        public DataStatistics mapRow(ResultSet rs, int rowNum) throws SQLException {
            DataStatistics statistics = new DataStatistics();
            statistics.setAvg(rs.getDouble("avg"));
```

```
            statistics.setMin(rs.getDouble("min"));
            statistics.setMax(rs.getDouble("max"));
            return statistics;
        }
    }


}
```

Explanation:

1. **package com.example.showresults;**: This line specifies the package where the **ShowResultsController** class belongs.
2. **import com.example.showresults.model.DataStatistics;**: This import statement brings in the **DataStatistics** class, which represents statistical data.
3. **import org.springframework.beans.factory.annotation.Autowired;**: This import statement brings in the **Autowired** annotation from the Spring Framework, which is used for automatic dependency injection.
4. **import org.springframework.jdbc.core.JdbcTemplate;**, **import org.springframework.jdbc.core.RowMapper;**: These import statements bring in classes from Spring's JDBC framework for database interactions.
5. **import org.springframework.web.bind.annotation.PostMapping;**, **import org.springframework.web.bind.annotation.RequestBody;**, **import org.springframework.web.bind.annotation.RestController;**: These import statements bring in annotations for defining RESTful APIs and handling HTTP requests in Spring.
6. **@RestController**: This annotation is applied to the class to indicate that it is a Spring MVC controller that handles RESTful API requests.
7. **public class ShowResultsController { ... }**: This is the class definition for the **ShowResultsController** class.
8. **@Autowired**: This annotation is used to inject the **JdbcTemplate** bean into the controller.
9. **@PostMapping("/showresult")**: This annotation maps the **showResult** method to handle HTTP POST requests with the path "/showresult".
10. **public String showResult(@RequestBody String id) { ... }**: This method retrieves data statistics from the database based on the provided **id**. It uses the **JdbcTemplate** to perform the database query and map the result using the **DataStatisticsMapper**.
11. **private static final class DataStatisticsMapper implements RowMapper<DataStatistics> { ... }**: This nested class defines a **RowMapper** for mapping **DataStatistics** objects from a **ResultSet**. It implements the **RowMapper** interface and overrides the **mapRow** method to create and populate a **DataStatistics** object.

Application.probabilities

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://db/dockerr
spring.datasource.username=root
spring.datasource.password=Rawan
```

Dockerfile:

```
FROM maven:3.8.3-openjdk-17 AS build
COPY src /home/app/src
COPY pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package

FROM openjdk:17-alpine
COPY --from=build /home/app/target/showResult-0.0.1-SNAPSHOT.jar /usr/local/lib/showResult.jar
ENTRYPOINT ["java","-jar","/usr/local/lib/showResult.jar"]
```

Sql:

```
CREATE database dockerr;
USE dockerr;
CREATE TABLE user(id int , username varchar(50) , password varchar(50));
CREATE TABLE Datas(user_name varchar(50) , data double);
CREATE TABLE statistics(id int , min double , max double , avg double);
INSERT INTO user VALUES (1 , "a" , "1");
INSERT INTO user VALUES (2 , "b" , "2");
INSERT INTO user VALUES (3 , "c" , "3");
```

Finally the docker compose

```
version: "3.9"

networks:
  isolation-network:
    driver: bridge

services:

  analytics:
    build: ./analytics
    container_name: analytics
```

```
  ports:
   - "8082:8080"
  environment:
   spring.datasource.url: "jdbc:mysql://db/dockerr"
   spring.datasource.username: "root"
   spring.datasource.password: "Rawan"
  networks:
   - isolation-network


authentication:
  build: ./authenticate
  container_name: authentication
  ports:
   - "8081:8080"
  environment:
   spring.datasource.url: "jdbc:mysql://db/dockerr"
   spring.datasource.username: "root"
   spring.datasource.password: "Rawan"
  networks:
   - isolation-network


showresult:
  build: ./show-results
  container_name: showresult
  ports:
   - "8083:8080"
  environment:
   spring.datasource.url: "jdbc:mysql://db/dockerr"
   spring.datasource.username: "root"
   spring.datasource.password: "Rawan"
  networks:
   - isolation-network

dataentry:
  build: ./enter-data
  container_name: dataentry
  ports:
   - "8080:8080"
  environment:
   spring.datasource.url: "jdbc:mysql://db/dockerr"
   spring.datasource.username: "root"
   spring.datasource.password: "Rawan"
```

```
    networks:
      - isolation-network


  db:
    image: mysql:8
    container_name: mysql
    restart: always
    command: --init-file /data/application/init.sql
    environment:
      MYSQL_ROOT_PASSWORD: "Rawan"
      MYSQL_DATABASE: docker
    volumes:
      - database:/var/lib/mysql
      - ./init.sql:/data/application/init.sql
    networks:
      - isolation-network


volumes:
  database:
    external: false
```

Explanation:

1. **version: "3.9"**: This specifies the version of the Docker Compose file.
2. **networks:**: This section defines a Docker network named **isolation-network** with the **bridge** driver.
3. **services:**: This section defines various services/components of the application.
    - **analytics:**, **authentication:**, **showresult:**, **dataentry:**: These sections define individual services for analytics, authentication, displaying results, and data entry. They specify the Docker build context, container name, exposed ports, environment variables (database configuration), and network settings.
    - **db:**: This section defines a MySQL database service using the **mysql:8** Docker image. It specifies the container name, restart policy, environment variables (root password and database name), and volumes (database data and SQL initialization script). The database is connected to the **isolation-network**.
4. **volumes:**: This section defines a Docker volume named **database** which is used for persisting MySQL database data.

In conclusion, the journey I've undertaken involves the creation and deployment of a dynamic and versatile web application. Through the amalgamation of Spring Boot and Docker, I've crafted a multi-component architecture that seamlessly blends various functionalities into a cohesive system.

My application is divided into distinct services, each with a specific purpose. The Authentication Service acts as the guardian of user credentials, rigorously validating them against the database. Once verified, users are granted access to the system, signifying successful authentication.

The Data Entry Service, on the other hand, is responsible for accommodating user-generated data. As data flows in, it's meticulously parsed and organized into individual entries. These entries are then thoughtfully stored within the database, ensuring their safekeeping.

My pursuit of insights and knowledge continues with the Analytics Service. This diligent component crunches the numbers, extracting statistical gems from the data. Minimums, maximums, and averages are computed, adding a layer of intelligence to my application. These invaluable statistics are then meticulously preserved within the database for future reference.

To visualize these achievements, the Show Results Service comes to the fore. It acts as a storyteller, skillfully retrieving and presenting the calculated statistics based on a unique identifier. This harmonious orchestration of services culminates in a holistic user experience, seamlessly delivering the information I seek.

At the heart of it all lies the database interaction, facilitated by Spring's JdbcTemplate. This intricate web of communication ensures that my services can fluidly interact with the MySQL database, creating a robust backbone for my application.

Wrapping this endeavor in the cocoon of modernity is my Docker Compose Configuration. It's the script that brings my vision to life, setting up a network to connect my services. Each service container is meticulously crafted, defining its specific environment, interactions, and data storage. The database itself, entrusted with a critical role, is shielded by Docker volumes, ensuring data continuity.

In this digital symphony of code and configuration, I've harnessed the power of Spring Boot and Docker to create a harmonious ensemble of services, each playing its unique part. The result is a web application that transcends the sum of its parts, a testament to the potential of contemporary software development practices. My journey exemplifies the principles of modularity, scalability, and encapsulation, showcasing how these elements come together to craft a seamless and efficient user experience.