Project Title: Consuming and Exposing API in Spring Boot

Project Description:

This project aims to demonstrate how to consume and expose APIs using Spring Boot, a popular Java-based framework for building web applications.

In this project, we will create a Spring Boot application that consumes an external REST API and exposes its own REST API.
The external API will be a simple weather API that returns the current weather information for a given location.
Our application will consume this API and expose its own API that returns the weather information in a specific format.

Technologies and Tools Used:

Spring Boot
Spring Web
Spring RestTemplate
JSON
Maven
Project Requirements:

Consume an external API:
Create a Spring Boot application that consumes an external weather API using Spring RestTemplate.
The API should accept a location as input and return the current weather information for that location in JSON format.
Expose own API:
Create a REST API endpoint that accepts a location as input and returns the weather information in a specific format.
The format should include the current temperature, humidity, wind speed, and description of the weather condition.
Handle Errors:
Handle exceptions and error responses returned by the external API.
Handle exceptions and error responses returned by our own API.
Documentation:
Document the APIs using Swagger UI.
Project Implementation:

Step 1: Create a new Spring Boot project using the Spring Initializr.

Add the following dependencies: Spring Web and Spring RestTemplate.
Update the pom.xml file to include the necessary dependencies.
Step 2: Create a WeatherService class that uses the RestTemplate to consume the external weather API.

Create a method that accepts a location as input and returns the weather information for that location.
Handle exceptions and error responses returned by the API.

Step 3: Create a WeatherController class that exposes the weather information as a REST API endpoint.

Create a method that accepts a location as input and returns the weather information in a specific format.
Handle exceptions and error responses returned by our own API.
Step 4: Document the APIs using Swagger UI.

Add the necessary Swagger dependencies to the pom.xml file.
Annotate the REST API endpoints with Swagger annotations.
Step 5: Run the application and test the APIs using Swagger UI.

Conclusion:

In this project, we have demonstrated how to consume an external API and expose our own API using Spring Boot.
We have also learned how to handle errors and document our APIs using Swagger UI.
This project provides a good starting point for building more complex applications that consume and expose APIs.

First, create a new Spring Boot project and add the necessary dependencies to the pom.xml file.

```
 <!-- Spring Web -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- Spring RestTemplate -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Next, create a WeatherService class that uses the RestTemplate to consume an external weather API:

```
@Service
public class WeatherService {

    @Autowired
    private RestTemplate restTemplate;

    public WeatherData getWeatherData(String location) {
        String url = "http://api.openweathermap.org/data/2.5/weather?q=" + location
+ "&appid=<API_KEY>";
```

```
        WeatherData weatherData = restTemplate.getForObject(url,
WeatherData.class);
        return weatherData;
    }
}
```

In the WeatherService class, we use the RestTemplate to make a GET request to an external weather API.
The getWeatherData method accepts a location as input and returns the weather information for that location.

Next, create a WeatherController class that exposes the weather information as a REST API endpoint:

```
 @RestController
@RequestMapping("/weather")
public class WeatherController {

    @Autowired
    private WeatherService weatherService;

    @GetMapping("/{location}")
    public WeatherInfo getWeatherInfo(@PathVariable String location) {
        WeatherData weatherData = weatherService.getWeatherData(location);
        WeatherInfo weatherInfo = new WeatherInfo();
        weatherInfo.setTemperature(weatherData.getMain().getTemp());
        weatherInfo.setHumidity(weatherData.getMain().getHumidity());
        weatherInfo.setWindSpeed(weatherData.getWind().getSpeed());

weatherInfo.setDescription(weatherData.getWeather().get(0).getDescription());
        return weatherInfo;
    }
}
```

In the WeatherController class, we create a REST API endpoint that accepts a location as input and returns the weather information in a specific format.
The getWeatherInfo method calls the getWeatherData method in the WeatherService class to get the weather information for the given location.
It then creates a WeatherInfo object and sets the temperature, humidity, wind speed, and description of the weather condition.

Finally, run the application and test the APIs using Swagger UI. You can use the following WeatherInfo class:

```
 public class WeatherInfo {
    private double temperature;
    private int humidity;
    private double windSpeed;
    private String description;

    // getters and setters
}
```

That's it! With this code, you should be able to consume and expose your own API in Spring Boot.

or

api integration
First, add the necessary dependencies to the pom.xml file:

```
<!-- Spring Web -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- Spring RestTemplate -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Next, create a UserService class that uses the RestTemplate to integrate with an external user API:

```
@Service
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    public User getUserById(int id) {
        String url = "https://jsonplaceholder.typicode.com/users/" + id;
        User user = restTemplate.getForObject(url, User.class);
        return user;
    }
}
```

Next, create a UserController class that exposes the user information as a REST API endpoint:

```
@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public User getUserById(@PathVariable int id) {
        return userService.getUserById(id);
    }
}
```

Finally, run the application and test the APIs using Swagger UI. You can use the following User class:

```
public class User {

    private int id;
    private String name;
    private String username;
    private String email;
    private Address address;
    private String phone;
    private String website;
    private Company company;

    // getters and setters
}
```

Sure! Here is a sample code for redirecting from one server to another in Spring Boot:

```
@Controller
public class RedirectController {

    @GetMapping("/redirect")
    public String redirect() {
        return "redirect:http://www.example.com";
    }
}
```

Sure! Here is a sample code for encrypting and decrypting data in Spring Boot using the Java Cryptography Extension (JCE) library:

```
@Service
public class EncryptionService {

    @Value("${encryption.key}")
    private String encryptionKey;

    public String encrypt(String data) throws Exception {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        SecretKeySpec secretKeySpec = new SecretKeySpec(encryptionKey.getBytes(),
"AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec);
        byte[] encryptedData = cipher.doFinal(data.getBytes());
        return Base64.getEncoder().encodeToString(encryptedData);
    }

    public String decrypt(String encryptedData) throws Exception {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        SecretKeySpec secretKeySpec = new SecretKeySpec(encryptionKey.getBytes(),
"AES");
        cipher.init(Cipher.DECRYPT_MODE, secretKeySpec);
        byte[] decryptedData =
cipher.doFinal(Base64.getDecoder().decode(encryptedData));
        return new String(decryptedData);
    }
}
```

In the EncryptionService class, we create two methods for encrypting and decrypting data using the AES encryption algorithm with Electronic Code Book (ECB) mode and PKCS5 padding. The encryptionKey field is a secret key used to encrypt and decrypt the data.

Note that this is a simple example and in practice, it is recommended to use a more secure encryption algorithm and a more robust key management system.

You can also use the @RestController annotation to expose these methods as a REST API endpoint:

```
@RestController
public class EncryptionController {

    @Autowired
    private EncryptionService encryptionService;

    @PostMapping("/encrypt")
    public String encryptData(@RequestBody String data) throws Exception {
        return encryptionService.encrypt(data);
    }

    @PostMapping("/decrypt")
    public String decryptData(@RequestBody String encryptedData) throws Exception {
        return encryptionService.decrypt(encryptedData);
    }
}
```

In the EncryptionController class, we create two REST API endpoints for encrypting and decrypting data.
The encryptData method accepts data as input and returns the encrypted data, while the decryptData method accepts encrypted data as input and returns the decrypted data.

Note that the @RequestBody annotation is used to accept data in the request body.

That's it! With this code, you should be able to encrypt and decrypt data in Spring Boot.