Please create whole those things which are the specific requirements:

The two microservices must be created using Spring Boot.

Both microservices must be connected to a non-sql database.

An API gateway must be used to connect to both services.

When a request is sent to one service, the other service should display a message indicating that the first service is running and please wait.

When a request is sent to the second service, the first service should display a message indicating that the second service is running and please wait.

Grimoire

To accomplish this, we will design a system with the following components:

- **Microservice A and Microservice B**: Both will be Spring Boot applications.
- **Non-SQL Database**: We'll assume MongoDB for this scenario.
- **API Gateway**: A Spring Cloud Gateway instance to route requests to the appropriate microservices.
- **Inter-Service Communication**: To notify one service from another, we can use Spring Cloud Stream or direct HTTP calls. We will use direct HTTP calls for simplicity.

Here's a step-by-step plan:

Microservice A (ServiceA)

- **Spring Boot Application**: Create a standard Spring Boot application.
- **Controller**: A REST controller to accept requests.
- **HTTP Client**: To send a notification to Service B.
- **MongoDB Connection**: For database operations.
- **Properties**: Include service B's URL.

Microservice B (ServiceB)

- **Spring Boot Application**: Another standard Spring Boot application.
- **Controller**: A REST controller to accept requests.
- **HTTP Client**: To send a notification to Service A.
- **MongoDB Connection**: For database operations.
- **Properties**: Include service A's URL.

API Gateway

- **Spring Cloud Gateway**: Route requests to Service A and Service B.

Setup and Communication Flow:

- **Service A receives a request**: It sends an HTTP call to Service B saying "Service A is running, please wait".
- **Service B receives a request**: It sends an HTTP call to Service A saying "Service B is running, please wait".

Now, let's start with the code. I'll outline the necessary components for Microservice A and B and the API Gateway.

Microservice A (ServiceA)

pom.xml (for both Microservices A and B)

```
<dependencies> <!-- Spring Boot starter web --> <dependency>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-
web</artifactId> </dependency> <!-- Spring Boot starter data MongoDB --> <dependency>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-data-
mongodb</artifactId> </dependency> <!-- Spring Boot starter actuator for health checks -->
<dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-
actuator</artifactId> </dependency> <!-- Spring Cloud Gateway --> <dependency>
<groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-
gateway</artifactId> </dependency> <dependency>
<groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-
bootstrap</artifactId> </dependency> <!-- Other necessary dependencies --> </dependencies>
```

ServiceAController.java

java Copy code

```java
@RestController @RequestMapping("/serviceA") public class ServiceAController { @Autowired
private RestTemplate restTemplate; @PostMapping("/notifyServiceB") public
ResponseEntity<String> notifyServiceB() {
restTemplate.postForObject("http://localhost:8082/serviceB/notify", "Service A is running,
please wait", String.class); return ResponseEntity.ok("Service A running, notified Service
B"); } @GetMapping("/process") public ResponseEntity<String> process() { // Process
something notifyServiceB(); // Notify Service B return ResponseEntity.ok("Service A
processing"); } @Bean public RestTemplate restTemplate() { return new RestTemplate(); } }
```

application.properties (for ServiceA)

properties Copy code

```properties
server.port=8081 spring.data.mongodb.uri=mongodb://localhost:27017/serviceADB
```

Microservice B (ServiceB)

It will have similar structures to Microservice A but with its specific endpoints. The main difference is in the `ServiceBController.java` and `application.properties`.

API Gateway (Spring Cloud Gateway)

application.properties

properties Copy code

```properties
spring.cloud.gateway.routes[0].id=serviceA_route
spring.cloud.gateway.routes[0].uri=http://localhost:8081
spring.cloud.gateway.routes[0].predicates[0]=Path=/serviceA/**
spring.cloud.gateway.routes[1].id=serviceB_route
spring.cloud.gateway.routes[1].uri=http://localhost:8082
spring.cloud.gateway.routes[1].predicates[0]=Path=/serviceB/**
```

Steps to Implement

Create Spring Boot projects for ServiceA, ServiceB, and APIGateway.
Add the necessary dependencies.
Create the controller, services, and configuration for each microservice.
Implement the API Gateway to route requests appropriately.
Run all applications and test the communication between services through the gateway.

This is a high-level implementation plan. The actual implementation might require adjustments based on your specific requirements and environment.

Let me know if you need the code for ServiceB, API Gateway setup, or any particular part in more detail.

DE **You**

I need a code for service B as well as API gateway setup.

Grimoire

Sure, let's proceed with the code for Microservice B (ServiceB) and then move on to setting up the API Gateway.

Microservice B (ServiceB)

ServiceBController.java

```java
package com.example.serviceb; import
org.springframework.beans.factory.annotation.Autowired; import
org.springframework.http.ResponseEntity; import
org.springframework.web.bind.annotation.GetMapping; import
org.springframework.web.bind.annotation.PostMapping; import
org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RestController; import
org.springframework.web.client.RestTemplate; import
org.springframework.context.annotation.Bean; @RestController @RequestMapping("/serviceB")
public class ServiceBController { @Autowired private RestTemplate restTemplate;
@PostMapping("/notifyServiceA") public ResponseEntity<String> notifyServiceA() {
restTemplate.postForObject("http://localhost:8081/serviceA/notify", "Service B is running,
please wait", String.class); return ResponseEntity.ok("Service B running, notified Service
A"); } @GetMapping("/process") public ResponseEntity<String> process() { // Process
something notifyServiceA(); // Notify Service A return ResponseEntity.ok("Service B
processing"); } @Bean public RestTemplate restTemplate() { return new RestTemplate(); } }
```

application.properties (for ServiceB)

```properties
server.port=8082 spring.data.mongodb.uri=mongodb://localhost:27017/serviceBDB
```

pom.xml for ServiceB will be the same as for ServiceA.

API Gateway (Spring Cloud Gateway)

Create a new Spring Boot project for the API Gateway. Add the following dependencies in the `pom.xml` file:

pom.xml (API Gateway)

```xml
<dependencies> <!-- Spring Boot starter web --> <dependency>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-
web</artifactId> </dependency> <!-- Spring Cloud Gateway --> <dependency>
<groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-
gateway</artifactId> <version>YOUR_SPRING_CLOUD_VERSION</version> <!-- Use the appropriate
Spring Cloud version --> </dependency> <!-- Spring Boot Actuator --> <dependency>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-
actuator</artifactId> </dependency> <!-- Other necessary dependencies --> </dependencies>
```

Replace `YOUR_SPRING_CLOUD_VERSION` with the version compatible with your Spring Boot version.

application.properties (API Gateway)

```properties
spring.cloud.gateway.routes[0].id=serviceA_route
spring.cloud.gateway.routes[0].uri=http://localhost:8081
spring.cloud.gateway.routes[0].predicates[0]=Path=/serviceA/**
spring.cloud.gateway.routes[1].id=serviceB_route
spring.cloud.gateway.routes[1].uri=http://localhost:8082
spring.cloud.gateway.routes[1].predicates[0]=Path=/serviceB/**
```

This configuration defines two routes: one for ServiceA and one for ServiceB. The Gateway will forward any requests with a path starting with `/serviceA` to the ServiceA microservice and those starting with `/serviceB` to the ServiceB microservice.

Running the Microservices and the API Gateway:

Start MongoDB: Ensure that MongoDB is running on your machine.
Start ServiceA: Navigate to the root directory of ServiceA and run `./mvnw spring-boot:run`.
Start ServiceB: Navigate to the root directory of ServiceB and run `./mvnw spring-boot:run`.
Start API Gateway: Navigate to the root directory of the API Gateway and run `./mvnw spring-boot:run`.

After starting all services, you can test them by sending requests to the API Gateway's endpoints, which will route the requests to the appropriate microservices and handle inter-service communication.

If you have any more specific requirements or need further assistance with any of these steps, let me know!

You
Can u show me the example output of our implementation.

Grimoire
Sure, let's go through how the communication and outputs might look in our setup with Microservices A and B, and the API Gateway.

Example Scenario:

Let's assume we have everything set up and running:

MongoDB is running locally.
Service A is running on `localhost:8081`.
Service B is running on `localhost:8082`.
API Gateway is running and routing paths `/serviceA` to `localhost:8081` and `/serviceB` to `localhost:8082`.

Test Case 1: Sending a Request to Service A

You send a POST request to the API Gateway with the path `/serviceA/process`. The API Gateway routes this request to Service A.
Service A receives the request. Before it starts processing, it sends a notification to Service B through the endpoint `/serviceB/notifyServiceA` indicating that Service A is running and asking to wait.
Service B receives the notification and logs or responds with "Service B running, notified Service A".
Service A continues with its processing and eventually responds to the original request.

Example Output:

Service A Log/Output:

```css
Received processing request. Notified Service B. Service A processing completed.
```

Service B Log/Output:

```css
```

```
Received notification from Service A.
```

Client (Postman, curl, browser, etc.):

```css
Service A processing
```

Test Case 2: Sending a Request to Service B

You send a POST request to the API Gateway with the path **`/serviceB/process`**. The API Gateway routes this request to Service B.

Service B receives the request. Before it starts processing, it sends a notification to Service A through the endpoint **`/serviceA/notifyServiceB`** indicating that Service B is running and asking to wait.

Service A receives the notification and logs or responds with "Service A running, notified Service B".

Service B continues with its processing and eventually responds to the original request.

Example Output:

Service B Log/Output:

```css
Received processing request. Notified Service A. Service B processing completed.
```

Service A Log/Output:

```css
Received notification from Service B.
```

Client (Postman, curl, browser, etc.):

```css
Service B processing
```

These outputs are based on the implementation of logging and response mechanisms within each service. In real scenarios, you might also have additional logs for HTTP requests and responses, as well as database interactions. Additionally, the actual content of notifications and responses can be customized as per your application's requirements.

You

One more thing if I am using first service on that time I do not want to send message , if I am using firs service and ar the same another user send the request on second service on that time I want to show the message to other user like service a is using someone please wait for using service B

I think now u understand better what I want and please suggest the changes.

Grimoire

Understandably, you want a mechanism where if Service A is currently processing a request, any subsequent requests to Service B during that time should notify the user that Service A is in use and they should wait. Conversely, if Service B is processing, then Service A should inform its users accordingly.

To implement this, we need a shared state between Service A and Service B that tracks whether a service is currently processing a request. This could be done using a shared database or an in-memory data store. For simplicity, I'll assume an in-memory approach using a static variable, but be aware this solution only works if you have a single instance of each service. For multi-instance or scalable environments, you would need a centralized system like Redis or shared database flags.

Changes Required:

Shared State:

Both services will need a way to check if the other service is currently busy. This could be represented as a simple boolean flag in each service.

Service A:

- Add a static boolean flag `isProcessing` to track when Service A is processing a request.
- In the `/process` endpoint, set `isProcessing` to `true` when processing starts and to `false` when it finishes.
- Create a new endpoint, `/status`, that returns the status of `isProcessing`.
- Before processing in Service B, call Service A's `/status` endpoint to check if it is currently processing.

Service B:

Mirror the changes made to Service A:

- Add a static boolean flag `isProcessing`.
- Adjust the `/process` endpoint to update `isProcessing`.
- Add a `/status` endpoint to expose `isProcessing`.
- Call Service A's `/status` before processing requests.

Implementation:

For Service A (and similarly for Service B):

```java
@RestController @RequestMapping("/serviceA") public class ServiceAController { private
static boolean isProcessing = false; @Autowired private RestTemplate restTemplate;
@PostMapping("/process") public ResponseEntity<String> process() { isProcessing = true; //
Simulate processing try { Thread.sleep(10000); // Simulate time-consuming processing }
catch (InterruptedException e) { Thread.currentThread().interrupt(); } isProcessing =
false; return ResponseEntity.ok("Service A processed request."); } @GetMapping("/status")
public ResponseEntity<String> status() { return isProcessing ? ResponseEntity.ok("Service
A is currently processing, please wait.") : ResponseEntity.ok("Service A is idle."); }
@Bean public RestTemplate restTemplate() { return new RestTemplate(); } }
```

Logic for Checking Status Before Processing:

In Service B (mirroring for Service A), before starting the actual processing in the `/process` endpoint, you would do:

```java
String serviceAStatus = restTemplate.getForObject("http://localhost:8081/serviceA/status",
String.class); if(serviceAStatus.contains("processing")) { return
ResponseEntity.status(HttpStatus.CONFLICT).body("Service A is currently processing, please
wait."); } isProcessing = true; // Proceed with processing isProcessing = false;
```

Notes:

- **Scalability**: As mentioned, this approach with static flags only works reliably for single-instance setups. For scaling to multiple instances, you would need a centralized system to manage the state.
- **Database or Persistent Store**: If using a database or a system like Redis, instead of hitting an endpoint to check the status, each service would check the shared persistent store's state.
- **Asynchronous Processing**: If the processing is asynchronous, you would need additional logic to handle the setting and unsetting of the `isProcessing` flag appropriately.
- **Real-time Communication**: For a more responsive solution, consider using real-time communication (like WebSocket) between services for status updates, but this adds complexity.

Error Handling: Add appropriate error handling, especially for network calls between services.

This solution should meet your requirements. If there are any more details or specific situations you need to address, please let me know!