Day 8: Reactive Spring - Real-time Claim Status Updates

Task 1: Introduce Spring WebFlux for handling real-time claim status updates using reactive streams.

1. Setup Reactive Spring Boot Project

Start by creating a new Spring Boot project with WebFlux dependencies to handle reactive programming:

1.1. Project Setup

Use Spring Initializr to generate a new project with the following dependencies:

- **Dependencies**: Select Reactive Web and optionally Spring Data Reactive MongoDB or Spring Data Reactive R2DBC depending on your data storage needs.

1.2. Configure WebFlux

Ensure your main application class is set up to enable reactive programming:

java

```
package com.example.reactiveclaims;


import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication

public class ReactiveClaimsApplication {


  public static void main(String[] args) {

    SpringApplication.run(ReactiveClaimsApplication.class, args);

  }


}
```

2. Define Claim Entity

Create a Claim entity class with relevant fields for claim status updates:

java

```
package com.example.reactiveclaims.model;


import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;


@Document
public class Claim {


    @Id
    private String id;
    private String claimNumber;
    private String status;


    // Getters and setters
}
```

3. Implement Reactive Repository

If using MongoDB or R2DBC, implement a reactive repository for CRUD operations:

3.1. Reactive MongoDB Repository

java

```
package com.example.reactiveclaims.repository;


import com.example.reactiveclaims.model.Claim;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import org.springframework.stereotype.Repository;


@Repository
public interface ClaimRepository extends ReactiveMongoRepository<Claim, String> {
}
```

4. Create Reactive REST Controller

Define a reactive REST controller to handle real-time updates and queries:

4.1. Reactive Controller

java

```
package com.example.reactiveclaims.controller;
```

```
import com.example.reactiveclaims.model.Claim;

import com.example.reactiveclaims.repository.ClaimRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.*;

import reactor.core.publisher.Flux;

import reactor.core.publisher.Mono;


@RestController

@RequestMapping("/claims")

public class ClaimController {


    @Autowired

    private ClaimRepository claimRepository;


    @GetMapping

    public Flux<Claim> getAllClaims() {

        return claimRepository.findAll();

    }


    @PostMapping

    public Mono<Claim> createClaim(@RequestBody Claim claim) {

        return claimRepository.save(claim);

    }


    // Other endpoints for updating and deleting claims


}
```

5. Implement Real-time Updates with WebFlux

Use reactive streams to push real-time updates to clients:

5.1. Reactive Endpoints

java

```
package com.example.reactiveclaims.controller;


import com.example.reactiveclaims.model.Claim;
import com.example.reactiveclaims.repository.ClaimRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;


@RestController
@RequestMapping("/claims")
public class ClaimController {
    @Autowired
    private ClaimRepository claimRepository;
  @GetMapping
    public Flux<Claim> getAllClaims() {
        return claimRepository.findAll();
    }


    @PostMapping
    public Mono<Claim> createClaim(@RequestBody Claim claim) {
        return claimRepository.save(claim);
    }
   @GetMapping("/{id}")
    public Mono<Claim> getClaimById(@PathVariable String id) {
        return claimRepository.findById(id);
    }
    @PutMapping("/{id}")
    public Mono<Claim> updateClaim(@PathVariable String id, @RequestBody Claim claim) {
        return claimRepository.findById(id)
                .flatMap(existingClaim -> {
```

```
                existingClaim.setStatus(claim.getStatus());
                return claimRepository.save(existingClaim);
            });
    }
    @DeleteMapping("/{id}")
    public Mono<Void> deleteClaim(@PathVariable String id) {
        return claimRepository.deleteById(id);
    }
    @GetMapping(value = "/{id}/status", produces = "text/event-stream")
    public Flux<String> getClaimStatusUpdates(@PathVariable String id) {
        return claimRepository.findById(id)
            .flatMapMany(claim -> {
                return Flux.interval(Duration.ofSeconds(5)) // Emit every 5 seconds
                        .map(sequence -> claim.getStatus());
            });
    }
}
```

6. Testing and Deployment

- **Run Application**: Start your Spring Boot application (ReactiveClaimsApplication) to deploy the reactive endpoints.
- **Test Endpoints**: Use tools like Postman or curl to test CRUD operations and real-time updates (/claims/{id}/status) endpoint.
- **Monitor Reactivity**: Monitor reactive streams and performance using Spring Boot Actuator and logging frameworks.

Task 2: Configure R2DBC for reactive database connectivity to update claim status dynamically.

Configuring R2DBC (Reactive Relational Database Connectivity) for reactive database connectivity in Spring Boot allows you to handle database operations reactively, suitable for real-time applications like updating claim statuses dynamically. R2DBC provides non-blocking database access for relational databases, contrasting with traditional blocking JDBC connections. Here’s how you can set it up:

Task 2: Configure R2DBC for Reactive Database Connectivity

1. Add Dependencies

Start by adding necessary dependencies for R2DBC and the database driver (e.g., PostgreSQL, MySQL):

1.1. Maven Dependencies

Add dependencies to your pom.xml for R2DBC and the database driver. For example, using PostgreSQL:

```xml
<dependencies>
  <!-- R2DBC Postgres driver -->
  <dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-postgresql</artifactId>
    <version>0.9.0.RELEASE</version>
  </dependency>

  <!-- Spring Boot Starter Data R2DBC -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-r2dbc</artifactId>
  </dependency>

  <!-- Spring Boot Starter Webflux (if not already included) -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
  </dependency>
</dependencies>
```

1.2. Gradle Dependencies

For Gradle, include dependencies in build.gradle:

```groovy
dependencies {
  implementation 'io.r2dbc:r2dbc-postgresql:0.9.0.RELEASE'
  implementation 'org.springframework.boot:spring-boot-starter-data-r2dbc'
```

```
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
}
```

2. Configure Database Connection

Configure R2DBC connection properties in application.properties or application.yml:

2.1. Application Properties

properties

```
# PostgreSQL example
spring.r2dbc.url=r2dbc:postgresql://localhost:5432/yourdatabase
spring.r2dbc.username=db_username
spring.r2dbc.password=db_password
spring.r2dbc.pool.initial-size=2
```

2.2. Application YAML

yaml

Copy code

```
# PostgreSQL example
spring:
  r2dbc:
    url: r2dbc:postgresql://localhost:5432/yourdatabase
    username: db_username
    password: db_password
    pool:
      initial-size: 2
```

3. Define Entity and Repository

Define an entity and repository for interacting with the database using R2DBC:

3.1. Entity Class

Java

```
package com.example.reactiveclaims.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;
```

```
@Table("claims")
public class Claim {

    @Id
    private Long id;
    private String claimNumber;
    private String status;
}
```

3.2. Reactive Repository

Create a reactive repository interface extending ReactiveCrudRepository:

java

```
package com.example.reactiveclaims.repository;

import com.example.reactiveclaims.model.Claim;
import org.springframework.data.repository.reactive.ReactiveCrudRepository;

public interface ClaimRepository extends ReactiveCrudRepository<Claim, Long> {
}
```

4. Update Claim Status Reactively

Implement reactive endpoints to update claim status dynamically using R2DBC:

4.1. Reactive Controller

java

Copy code

```
package com.example.reactiveclaims.controller;

import com.example.reactiveclaims.model.Claim;
import com.example.reactiveclaims.repository.ClaimRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
```

```
@RestController
@RequestMapping("/claims")
public class ClaimController {
   @Autowired
   private ClaimRepository claimRepository;
  @GetMapping
   public Flux<Claim> getAllClaims() {
      return claimRepository.findAll();
   }

   @PostMapping
   public Mono<Claim> createClaim(@RequestBody Claim claim) {
      return claimRepository.save(claim);
   }    @GetMapping("/{id}")
   public Mono<Claim> getClaimById(@PathVariable Long id) {
      return claimRepository.findById(id);
   }
@PutMapping("/{id}")
   public Mono<Claim> updateClaimStatus(@PathVariable Long id, @RequestBody Claim
updatedClaim) {
      return claimRepository.findById(id)
            .flatMap(existingClaim -> {
               existingClaim.setStatus(updatedClaim.getStatus());
               return claimRepository.save(existingClaim);
            });
   }
}
```

5. Testing and Deployment

- **Run Application**: Start your Spring Boot application (ReactiveClaimsApplication) to deploy reactive endpoints using R2DBC.

- **Test Endpoints**: Use tools like Postman or curl to test CRUD operations and verify dynamic updates of claim statuses.
- **Monitor Reactivity**: Monitor reactive streams and performance using Spring Boot Actuator and logging frameworks.

Task 3: Implement WebSocket communication for real-time interaction between the client and the server.

Implementing WebSocket communication in Spring Boot allows for real-time interaction between clients and servers, suitable for applications requiring instant updates and notifications. Here's a guide to implement Task 3 using WebSocket in Spring Boot:

Task 3: Implement WebSocket Communication

1. Add Dependencies

Start by adding necessary dependencies for WebSocket support in your pom.xml or build.gradle:

1.1. Maven Dependencies

Add dependencies for WebSocket and Spring Boot Starter Web:

xml

Copy code

```
<dependencies>
  <!-- Spring Boot Starter Web for general web support -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Boot Starter WebSocket for WebSocket support -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
  </dependency>
</dependencies>
```

1.2. Gradle Dependencies

For Gradle, include dependencies in build.gradle:

groovy

Copy code

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-web'
   implementation 'org.springframework.boot:spring-boot-starter-websocket'
}
```

2. Configure WebSocket Endpoint

Create a WebSocket endpoint handler in your Spring Boot application:

2.1. WebSocket Handler

Create a handler class that extends TextWebSocketHandler to manage WebSocket connections and messages:

java

```
package com.example.reactiveclaims.websocket;

import org.springframework.stereotype.Component;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.TextWebSocketHandler;

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

@Component
public class ClaimStatusWebSocketHandler extends TextWebSocketHandler {

   private Set<WebSocketSession> sessions = new HashSet<>();

   @Override
   public void afterConnectionEstablished(WebSocketSession session) throws Exception {
      sessions.add(session);
   }
```

```
    @Override
    protected void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception {
        // Handle incoming messages (if needed)
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session, org.springframework.web.socket.CloseStatus closeStatus) throws Exception {
        sessions.remove(session);
    }

    // Method to send messages to all connected sessions
    public void broadcastClaimStatus(String status) {
        TextMessage message = new TextMessage(status);
        sessions.forEach(session -> {
            try {
                session.sendMessage(message);
            } catch (IOException e) {
                // Handle exception
            }
        });
    }
}
```

3. Configure WebSocket Endpoint

Configure WebSocket endpoint and message broker in Spring Boot:

3.1. WebSocket Configuration

Configure WebSocket endpoint and message broker in WebSocketConfig.java:

java

```
package com.example.reactiveclaims.config;
```

```
import com.example.reactiveclaims.websocket.ClaimStatusWebSocketHandler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Autowired
    private ClaimStatusWebSocketHandler webSocketHandler;

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(webSocketHandler, "/ws/claim-status")
                .setAllowedOrigins("*"); // Allow all origins (you may restrict as needed)
    }
}
```

4. Use WebSocket in Controller

Use WebSocket to send real-time updates from your controller:

4.1. Controller Integration

Inject the ClaimStatusWebSocketHandler into your controller and use it to broadcast updates:

java

```
package com.example.reactiveclaims.controller;
import com.example.reactiveclaims.websocket.ClaimStatusWebSocketHandler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/claims")
```

```
public class ClaimController {
    @Autowired
    private ClaimStatusWebSocketHandler webSocketHandler;
    @PutMapping("/{id}/status")
    public String updateClaimStatus(@PathVariable Long id, @RequestBody String status) {
        webSocketHandler.broadcastClaimStatus("Claim ID " + id + " status updated to " + status);
            return "Claim status updated successfully";
    }
}
```

5. Client-side Integration

Integrate WebSocket on the client-side (e.g., JavaScript) to receive updates from the server and handle WebSocket events.