NAGELLA PRAJITHRAJ

Day 8: Reactive Spring - Real-Time Alerts and Notifications

Task 1: Apply Spring WebFlux to Develop a Non-Blocking, Reactive System for Sending Real-Time Traffic Alerts

Set Up Spring WebFlux:

Add dependencies for Spring WebFlux in your `pom.xml`.

xml

Copy code

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-r2dbc</artifactId>
    </dependency>
    <dependency>
        <groupId>io.r2dbc</groupId>
        <artifactId>r2dbc-postgresql</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-websocket</artifactId>
    </dependency>
</dependencies>
```

1.

Create a Reactive Controller for Traffic Alerts:

Implement a controller to handle real-time traffic alerts.

java

Copy code

```
@RestController
@RequestMapping("/alerts")
public class TrafficAlertController {

    private final TrafficAlertService alertService;
```

```
    @Autowired
    public TrafficAlertController(TrafficAlertService alertService) {
        this.alertService = alertService;
    }

    @GetMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<TrafficAlert> streamTrafficAlerts() {
        return alertService.getTrafficAlerts();
    }
}
```

2.

Implement a Reactive Service for Traffic Alerts:
Create a service class to provide reactive traffic alerts.
java
Copy code

```
@Service
public class TrafficAlertService {

    private final Flux<TrafficAlert> trafficAlerts;

    public TrafficAlertService() {
        this.trafficAlerts = Flux.interval(Duration.ofSeconds(1))
                .map(this::generateTrafficAlert)
                .share();
    }

    public Flux<TrafficAlert> getTrafficAlerts() {
        return trafficAlerts;
    }

    private TrafficAlert generateTrafficAlert(long interval) {
        // Generate a traffic alert based on interval or some logic
        return new TrafficAlert("Traffic congestion detected at
interval: " + interval);
    }
}
```

```
public class TrafficAlert {
    private String message;

    public TrafficAlert(String message) {
        this.message = message;
    }

    // Getters and setters
}
```

3.

Task 2: Use R2DBC for Integrating Reactive Data Updates to the Traffic Management System

Configure R2DBC Database Connection:

Add database connection properties in `application.properties`.

properties

Copy code

```
spring.r2dbc.url=r2dbc:postgresql://localhost:5432/trafficdb
spring.r2dbc.username=yourusername
spring.r2dbc.password=yourpassword
```

1.

Create a Reactive Repository for Traffic Data:

Define a repository interface for reactive data access.

java

Copy code

```
@Repository
public interface TrafficDataRepository extends
ReactiveCrudRepository<TrafficData, Long> {
}
```

2.

Implement a Service for Reactive Data Updates:

Create a service class to manage reactive data updates.

java

Copy code

```
@Service
public class TrafficDataService {
```

```
    private final TrafficDataRepository repository;

    @Autowired
    public TrafficDataService(TrafficDataRepository repository) {
        this.repository = repository;
    }

    public Flux<TrafficData> getAllTrafficData() {
        return repository.findAll();
    }

    public Mono<TrafficData> saveTrafficData(TrafficData trafficData)
{
        return repository.save(trafficData);
    }
}

@Data
@Table("traffic_data")
public class TrafficData {
    @Id
    private Long id;
    private String location;
    private String status;
    private LocalDateTime timestamp;
}
```

3.

Create Reactive Controller for Traffic Data:
Implement a controller to manage reactive traffic data.
java
Copy code

```
@RestController
@RequestMapping("/traffic-data")
public class TrafficDataController {

    private final TrafficDataService trafficDataService;

    @Autowired
```

```
    public TrafficDataController(TrafficDataService
trafficDataService) {
        this.trafficDataService = trafficDataService;
    }

    @GetMapping
    public Flux<TrafficData> getAllTrafficData() {
        return trafficDataService.getAllTrafficData();
    }

    @PostMapping
    public Mono<TrafficData> saveTrafficData(@RequestBody TrafficData
trafficData) {
        return trafficDataService.saveTrafficData(trafficData);
    }
}
```

4.

Task 3: Set Up WebSocket Channels for Broadcasting City-Wide Transportation Notifications and Updates

Configure WebSocket in Spring Boot:

Add WebSocket configuration class.

java

Copy code

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements
WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry)
{
        registry.addEndpoint("/ws").withSockJS();
```

```
    }
}
```

1.

Create WebSocket Controller:
Implement a controller to handle WebSocket messaging.
java
Copy code

```
@Controller
public class WebSocketController {

    private final SimpMessagingTemplate template;

    @Autowired
    public WebSocketController(SimpMessagingTemplate template) {
        this.template = template;
    }

    @MessageMapping("/sendNotification")
    @SendTo("/topic/notifications")
    public Notification sendNotification(Notification notification) {
        return notification;
    }

    public void broadcastNotification(Notification notification) {
        template.convertAndSend("/topic/notifications", notification);
    }
}

public class Notification {
    private String message;

    public Notification(String message) {
        this.message = message;
    }

    // Getters and setters
}
```

2.

Set Up WebSocket Client:
Create an HTML page with a WebSocket client for real-time notifications.
html
Copy code

```
<!DOCTYPE html>
<html>
<head>
    <title>WebSocket Notifications</title>
    <script
src="https://cdn.jsdelivr.net/sockjs/1.0.3/sockjs.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/stompjs@2.3.3/lib/stomp.min.js"></sc
ript>
    <script>
        var stompClient = null;

        function connect() {
            var socket = new SockJS('/ws');
            stompClient = Stomp.over(socket);
            stompClient.connect({}, function (frame) {
                console.log('Connected: ' + frame);
                stompClient.subscribe('/topic/notifications', function
(notification) {

showNotification(JSON.parse(notification.body).message);
                });
            });
        }

        function showNotification(message) {
            var notificationElement = document.createElement("div");
            notificationElement.innerText = message;

document.getElementById("notifications").appendChild(notificationEleme
nt);
        }
```

```
        window.onload = function() {
            connect();
        };
    </script>
</head>
<body>
    <h1>WebSocket Notifications</h1>
    <div id="notifications"></div>
</body>
</html>
```

3.

By implementing Spring WebFlux for reactive systems, R2DBC for reactive data updates, and WebSocket channels for real-time notifications, you can create a robust and scalable traffic monitoring system that provides real-time alerts and notifications to users. Adjust the code and configurations as needed to fit your specific application requirements.