Integrating HiveMQ as an MQTT Broker in Java Spring Boot

Introduction

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol ideal for IoT and real-time applications. HiveMQ provides a cloud-based and self-hosted MQTT broker solution that integrates well with Java Spring Boot applications.

This tutorial will guide you through integrating HiveMQ as an MQTT broker in a Spring Boot application. We will cover:

- Configuring an MQTT client
- Publishing messages
- Subscribing to messages and processing them
- Setting up dependencies
- Testing the integration

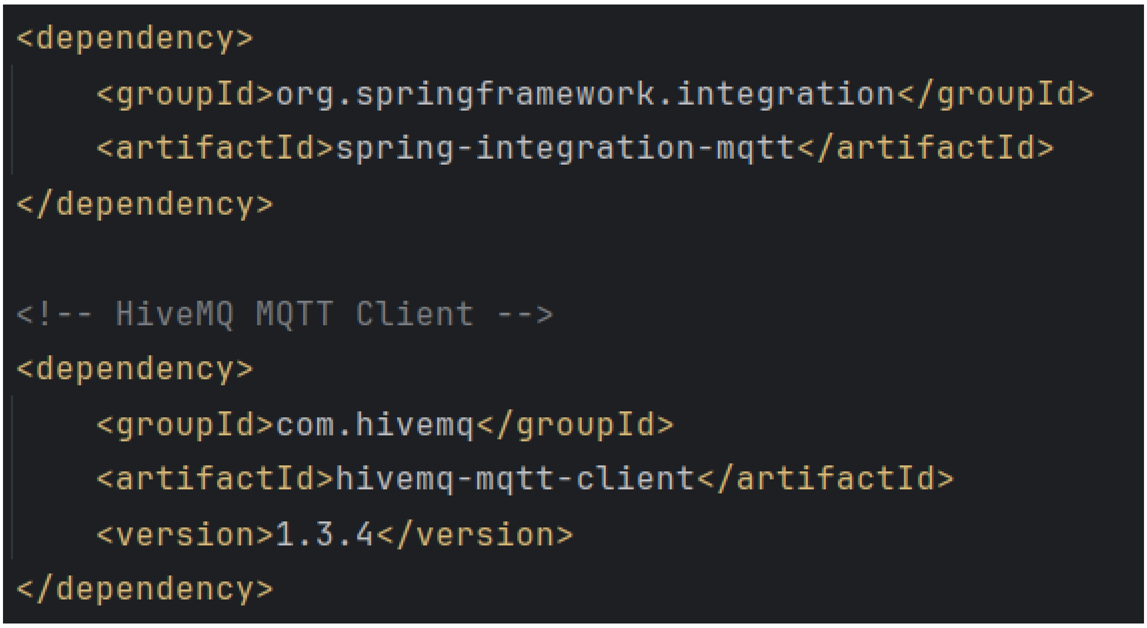
Prerequisites

Before proceeding, ensure you have:

- Java 17 or later installed
- Spring Boot set up in your project
- Maven or Gradle configured
- HiveMQ public broker details (or a local/private HiveMQ setup)
- Jackson library for JSON processing

Adding Dependencies

To use MQTT in Spring Boot, you need to add the required dependencies. If you're using Maven, include the following dependencies in your pom.xml:

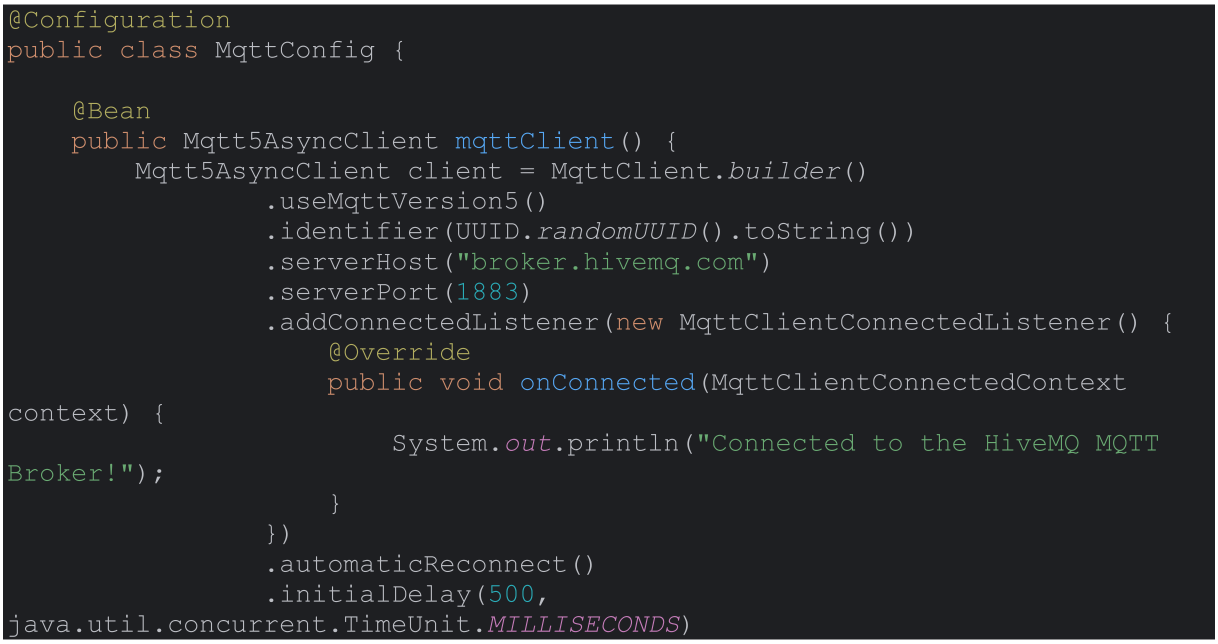


```xml
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-mqtt</artifactId>
</dependency>

<!-- HiveMQ MQTT Client -->
<dependency>
    <groupId>com.hivemq</groupId>
    <artifactId>hivemq-mqtt-client</artifactId>
    <version>1.3.4</version>
</dependency>
```

Setting Up the MQTT Client

First, we need to configure an MQTT client in Spring Boot. This will allow our application to connect to the HiveMQ broker.



```java
@Configuration
public class MqttConfig {

    @Bean
    public Mqtt5AsyncClient mqttClient() {
        Mqtt5AsyncClient client = MqttClient.builder()
                .useMqttVersion5()
                .identifier(UUID.randomUUID().toString())
                .serverHost("broker.hivemq.com")
                .serverPort(1883)
                .addConnectedListener(new MqttClientConnectedListener() {
                    @Override
                    public void onConnected(MqttClientConnectedContext
context) {
                        System.out.println("Connected to the HiveMQ MQTT
Broker!");
                    }
                })
                .automaticReconnect()
                .initialDelay(500,
java.util.concurrent.TimeUnit.MILLISECONDS)
```

```
                .maxDelay(1, java.util.concurrent.TimeUnit.MINUTES)
                .applyAutomaticReconnect()
                .buildAsync();
        return client;
    }
}
```

This configuration initializes an MQTT client that connects to HiveMQ, supports automatic reconnection, and assigns a unique identifier.

Publishing Messages

Next, let's create a service to publish messages to a specific MQTT topic.

```
@Service
public class MqttPublisherService {

    private final Mqtt5AsyncClient mqttClient;

    @Autowired
    public MqttPublisherService(Mqtt5AsyncClient mqttClient) {
        this.mqttClient = mqttClient;
    }

    public void publishMessage(String message) {
        // Check if the client is already connected
        if (!mqttClient.getState().isConnected()) {
            // Connect if not already connected
            mqttClient.connectWith()
                    .cleanStart(true)
                    .sessionExpiryInterval(500)
                    .send()
                    .whenComplete((connAck, throwable) -> {
                        if (throwable != null) {
                            System.out.println("Connection failed: " +
throwable.getMessage());
                        } else {
                            publish(message); // Call publish method after
successful connection
                        }
                    });
        } else {
            // Directly publish if already connected
            publish(message);
        }
    }

    private void publish(String message) {
        mqttClient.publishWith()
                .topic("SYSTEMCOMMUNICATION")
                .payload(message.getBytes())
                .send()
                .whenComplete((publishAck, pubThrowable) -> {
                    if (pubThrowable != null) {
                        System.out.println("Publish failed: " +
pubThrowable.getMessage());
                    } else {
                        System.out.println("Message published to topic
SYSTEMCOMMUNICATION");
                    }
                });
    }
}
```

This service checks if the client is connected before publishing messages and automatically attempts to reconnect if necessary.

Subscribing to Messages

Now, let's create a consumer service to subscribe to a topic and process incoming messages.

```
@Service
public class MqttMessageConsumerService {

    @Autowired
    private CardService cardService;

    @Autowired
    private WebSocketNotificationService notificationService;

    @PostConstruct
    public void subscribeToTopic() {
        Mqtt5AsyncClient client = MqttClient.builder()
                .useMqttVersion5()
                .identifier("consumer-id")
                .serverHost("broker.hivemq.com")
                .serverPort(1883) // Default MQTT port
                .automaticReconnectWithDefaultConfig() // Enable automatic
reconnect
                .buildAsync();

        client.connectWith()
                .cleanStart(true)
                .sessionExpiryInterval(500)
                .send()
                .whenComplete((connAck, throwable) -> {
                    if (throwable != null) {
                        System.out.println("Connection failed: " +
throwable.getMessage());
                    } else {
                        System.out.println("Connected successfully");
                        // Subscribe to a topic
                        client.subscribeWith()
                                .topicFilter("SYSTEMCOMMUNICATION") //
Change to your topic
                                .qos(MqttQos.AT_LEAST_ONCE)
                                .send()
                                .whenComplete((subAck, subThrowable) -> {
                                    if (subThrowable != null) {
                                        System.out.println("Subscribe
failed: " + subThrowable.getMessage());
                                    } else {
                                        System.out.println("Subscribed
successfully");
                                    }
                                });

                        // Set up callback for incoming messages

client.toAsync().publishes(MqttGlobalPublishFilter.ALL, publish -> {
                            try {
                                String message = new
String(publish.getPayloadAsBytes());
                                ObjectMapper objectMapper = new
ObjectMapper();
                                JsonNode jsonNode =
objectMapper.readTree(message);
                                String extractedMessage =
jsonNode.get("message").asText();

                                //any other logic can be added here to
process the payload
                                //in this case, we are saving the message
payload to the database and notifying listeners of this event though
websockets

                                Card card = new Card();
                                card.setContent(extractedMessage);
                                card.setTitle("MQTT Message");
                                cardService.save(card);
                                System.out.println("Received message: " +
message);

notificationService.sendCardSavedNotification(card);
                            } catch (Exception e) {
                                System.out.println("Failed to process
message: " + e.getMessage());
                            }
                        });
                    }
                });
    }
}
```

Testing the Integration

1. Run your Spring Boot application.
2. Ensure your publisher and consumer services are properly initialized.
3. Use the publish message method to send a message. In my case I created an endpoint for it

```
@RestController
@RequestMapping("/api/message")
public class MqttController {

    @Autowired
    private MqttPublisherService mqttPublisherService;

    @PostMapping("/send")
    public ResponseEntity<Map<String, String>> sendMessage(@RequestBody String message) {
        mqttPublisherService.publishMessage(message);

        Map<String, String> response = new HashMap<>();
        response.put("status", "success");
        response.put("message", "Successfully received!");

        return ResponseEntity.ok(response);
    }
}
```

4. Verify that MqttMessageConsumerService receives and processes the message

Conclusion

This tutorial demonstrated how to integrate HiveMQ as an MQTT broker in a Java Spring Boot application. We covered setting up dependencies, configuring an MQTT client, publishing messages, and subscribing to topics for message consumption.

By implementing these components, you can build real-time communication features for IoT applications, chat systems, and more using MQTT and HiveMQ.