**Step-by-Step Integration of ET (Event-Driven Technologies) with Solace**

In today’s distributed and real-time application landscape, messaging middleware plays a vital role in ensuring seamless communication between microservices and external systems. **Solace PubSub+** is a powerful event broker that supports multiple messaging protocols and patterns like publish/subscribe, queuing, request/reply, and streaming.

In this blog, we’ll walk through the **step-by-step process of integrating Solace in an event-driven microservices architecture**, particularly focusing on **Java Spring Boot applications**.

**🧩 What is Solace?**

**Solace PubSub+** is an advanced message broker that enables event mesh architecture. It supports protocols like:

* MQTT
* AMQP
* JMS
* REST
* WebSocket
* Native Solace API (Java, C, etc.)

Use cases include:

* Real-time data streaming
* Event-driven microservices
* Hybrid cloud communication

**🔧 Prerequisites**

Before starting integration:

* Java 11+ installed
* Maven or Gradle project setup
* Spring Boot app scaffolded
* Solace PubSub+ Broker running (either local or Cloud)

**🔌 Step-by-Step Integration of Solace with Java Spring Boot**

**Step 1: Setup Solace PubSub+**

You have two options:

1. **Cloud Version**:
   * Go to Solace Cloud Console
   * Create a new messaging service
   * Note down the connection details
2. **Local Docker Version**:

bash

CopyEdit

docker run -d -p 8080:8080 -p 55555:55555 -p 8008:8008 \

-p 1883:1883 -p 5672:5672 -p 9000:9000 \

--name=solace \

solace/solace-pubsub-standard

**Step 2: Add Solace Dependencies**

**If using Maven:**

xml

CopyEdit

<dependency>

<groupId>com.solace.spring.boot</groupId>

<artifactId>solace-spring-boot-starter</artifactId>

<version>4.0.0</version>

</dependency>

**If using Gradle:**

groovy

CopyEdit

implementation 'com.solace.spring.boot:solace-spring-boot-starter:4.0.0'

**Step 3: Configure application.yml or application.properties**

yaml

CopyEdit

solace:

java:

host: tcp://<host>:<port>

msgVpn: <your-vpn-name>

clientUsername: <username>

clientPassword: <password>

Replace the placeholders with actual credentials from Solace Cloud or Docker broker.

**Step 4: Create Publisher (Producer) Component**

java

CopyEdit

import com.solace.messaging.publisher.DirectMessagePublisher;

import com.solace.messaging.publisher.OutboundMessage;

import com.solace.messaging.MessagingService;

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

import org.springframework.stereotype.Service;

@Service

public class SolacePublisher {

private final DirectMessagePublisher publisher;

public SolacePublisher(MessagingService messagingService) {

this.publisher = messagingService.createDirectMessagePublisherBuilder().build().start();

}

public void sendMessage(String topic, String payload) {

OutboundMessage message = publisher.getMessageBuilder().build(payload);

publisher.publish(message, topic);

}

}

**Step 5: Create Subscriber (Consumer) Component**

java

CopyEdit

import com.solace.messaging.receiver.InboundMessage;

import com.solace.messaging.receiver.DirectMessageReceiver;

import com.solace.messaging.MessagingService;

import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;

@Service

public class SolaceSubscriber {

private final MessagingService messagingService;

public SolaceSubscriber(MessagingService messagingService) {

this.messagingService = messagingService;

}

@PostConstruct

public void init() {

DirectMessageReceiver receiver = messagingService

.createDirectMessageReceiverBuilder()

.withSubscriptions("sample/topic")

.build()

.start();

receiver.receiveAsync((InboundMessage message) -> {

String payload = message.getPayloadAsString();

System.out.println("Received message: " + payload);

});

}

}

**🔄 Test Integration**

You can test by triggering a REST endpoint or schedule a message publisher.

java

CopyEdit

@RestController

@RequestMapping("/solace")

public class SolaceController {

@Autowired

private SolacePublisher solacePublisher;

@PostMapping("/publish")

public ResponseEntity<String> publish(@RequestParam String msg) {

solacePublisher.sendMessage("sample/topic", msg);

return ResponseEntity.ok("Message sent!");

}

}

Call:

bash

CopyEdit

curl -X POST "http://localhost:8080/solace/publish?msg=HelloSolace"

**📊 Common Use Cases**

| **Use Case** | **How Solace Helps** |
| --- | --- |
| Event-driven microservices | Decouples services using topic-based pub/sub |
| Hybrid cloud connectivity | REST/MQTT bridges cloud and on-prem apps |
| Real-time dashboards | Stream data updates via WebSocket |
| IoT and MQTT apps | Native MQTT support for device communication |

**🛠️ Troubleshooting Tips**

| **Problem** | **Solution** |
| --- | --- |
| Connection refused | Check firewall, port, or Solace is running |
| Messages not received | Verify topic/subscription matching |
| Auth errors | Check username/password/VPN config |

**🧠 Best Practices**

* Use hierarchical topic naming (domain/service/event)
* Apply access control per topic or queue
* Use guaranteed messaging (persistent) for critical data
* Monitor via Solace PubSub+ Manager or Cloud Console

**🧾 Conclusion**

Solace provides a powerful foundation for building scalable, event-driven systems. By integrating it into your Spring Boot or microservice architecture, you can unlock the benefits of real-time messaging, high performance, and robust delivery guarantees.

If you're building event-driven systems, Solace is definitely a solution worth exploring.