Workshop: Event-sourced system through Reactive Streams

Introduction

In this workshop we'll be creating an Event-sourced system through Reactive Streams. We find two main concepts in that title: "Event-sourced" and "Reactive Streams"

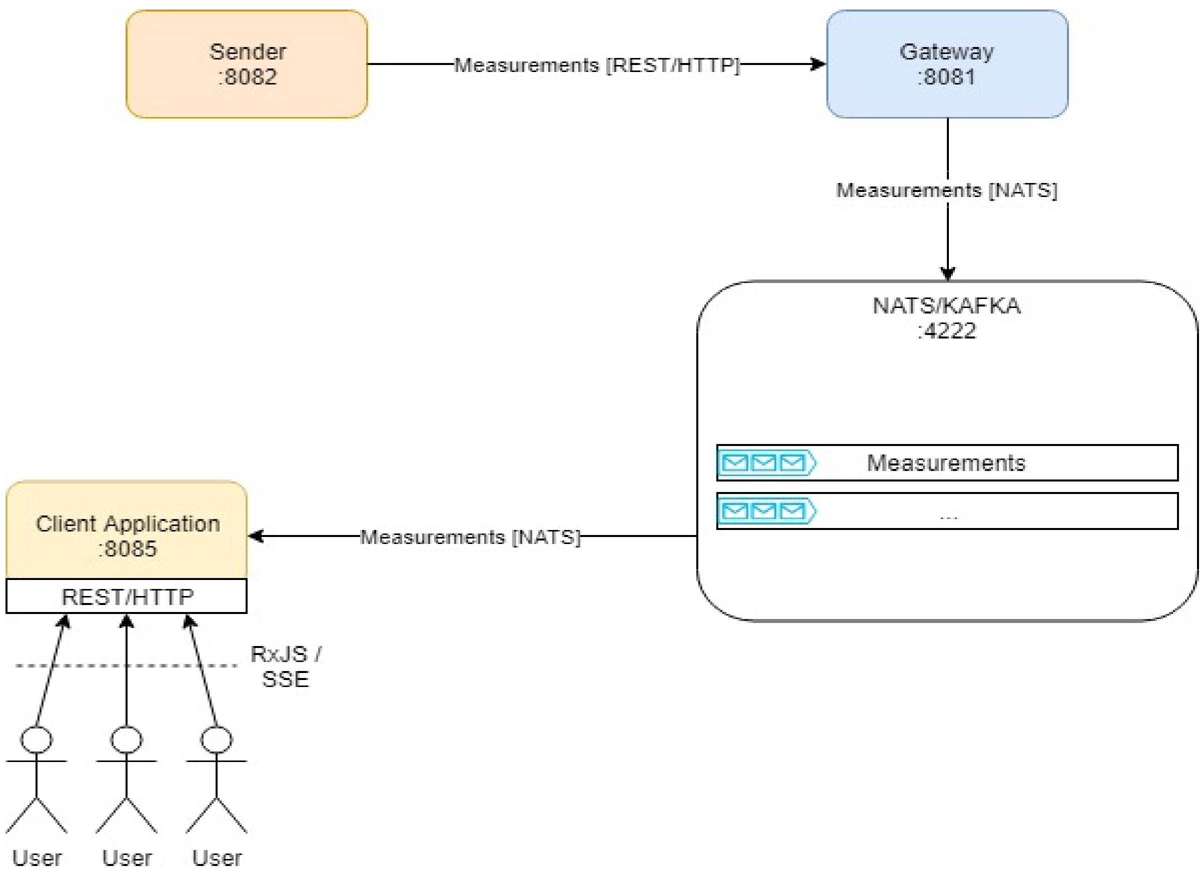
Event Sourcing is the idea of saving events for later use. Typically used in CQRS (Command Query Responsibility Separation) architectures, we don't save the current state of a system but rather the events that lead to that current state. Instead of storing a "bank account" which we update every time it changes which requires the table to be locked, we can simply store the events that happened to it. When combining these "ATMAddMoney", "ATMRetrieveMoney" and "TransferMoney" in the correct order, we should be able to get to the same state. This has severe advantages in terms of both speed and scalability. Because it does not require "locks" as we see in traditional database usage, we can achieve very high levels of scalability. When applying concepts of eventual consistency, we can even survive partitioning of our data or elevate it to scale even further.

Reactive Streams (also named Reactive Pipelines) enabled through Reactive Programming enable programming based on a Stream of events. It brings a message driven architecture on the level of the code inside an application. Events or messages could come from anywhere – user input through a keyboard, a messaging system, the resultset of a database query, etc. Because these Streams are handled in a message-driven manner, this means that the application does not have to block or wait at any point, enabling a far better use of resources.

Of course these concepts create an amazing synergy when combined together. We can scale many applications next to each other that react to outside events like user interactions and that will add events to the Event Store (like a user utilizing an ATM creating an ATMRetrieveMoney event). On the other side, there can be other applications using Reactive Event Streams that react to the new events added to the event store. Consider for example updating a local database inside an ATM with a new balance, or adding an additional fee to the user in case his balance goes negative.

This kind of Reactive Event-Driven architecture is extremely scalable. No locks have to happen on an architectural level and thanks to Reactive Streams data will just flow through the pipelines and add value to the users. Elasticity and Resilience are possible as well, because in case events are processed too slow, additional servers can be added which will just pick up additional work.

Architectural overview



Our Event-Driven Reactive System will consist out of a couple of different parts. We have the Sender which will emulate 10 devices sending a user defined number of measurements to the Gateway over HTTP. The Gateway accepts these values and sends them to the NATS server with the Subject “measurements”. NATS will add these measurements to an event-log with the given Subject.

On the other side we have our Client Application, which has a Reactive Stream of the Measurements Subject. Both existing and newly added Measurement Events will be streamed to the Client Application this way.

The client application will then filter out the Measurements which are in fact Anomalies and hold these in a local cache for as long as these are relevant. Users can get a (filtered) list of Anomalies from the Client Application. When new Anomaly Events are detected, these will be streamed to the User as well.

This is a very scalable architecture. In case we have more Senders (or in reality, Smart Meters), we can simply add more Gateways to capture these Measurements. The NATS (or Kafka) installation can be easily extended and clustered as well. In case we have more Users that want to have Anomaly Event information to stream in, more Client Applications can be added as well.

NATS

NATS Server is a simple, high performance open source messaging system for cloud native applications, IoT messaging, and microservices architectures. It enables messaging at a huge scale, with millions of messages per seconds.

NATS Streaming is a data streaming service powered by NATS. It enables Message/Event persistence and streaming towards applications. Through it we can create an event-store where events will be distributed from to other systems, and even has replay capabilities.

Kafka and Kafka-Streaming offer very similar capabilities. For this workshop, NATS can be completely substituted with Kafka, however only a solution for a NATS system is supplied.

Reactor-NATS-Streaming

For this exercise a work-in-progress version of **‘reactor-nats-streaming’** is added. This will take care of the integration between Project Reactor and NATS. In case you prefer to do this exercise with Kafka or RabbitMQ, you can refer to one of the following dependencies :

Kafka

```
<dependency>
    <groupId>io.projectreactor.kafka</groupId>
    <artifactId>reactor-kafka</artifactId>
    <version>1.1.0.RELEASE</version>
</dependency>
```

RabbitMQ

```
<dependency>
    <groupId>io.projectreactor.rabbitmq</groupId>
    <artifactId>reactor-rabbitmq</artifactId>
    <version>1.0.0.M3</version>
</dependency>
```

Project setup and structure

NATS Setup

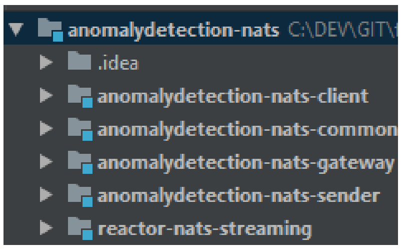
You can download the NATS Streaming server from https://nats.io/download/nats-io/nats-streaming-server/ After downloading just run nats-streaming-server.exe to start up the NATS server. By default NATS uses an in-memory event-store, but there are storage alternatives available as well.

Project setup

Clone the workshop project from https://github.com/vanseverk/anomalydetection-nats and into your favourite IDE.

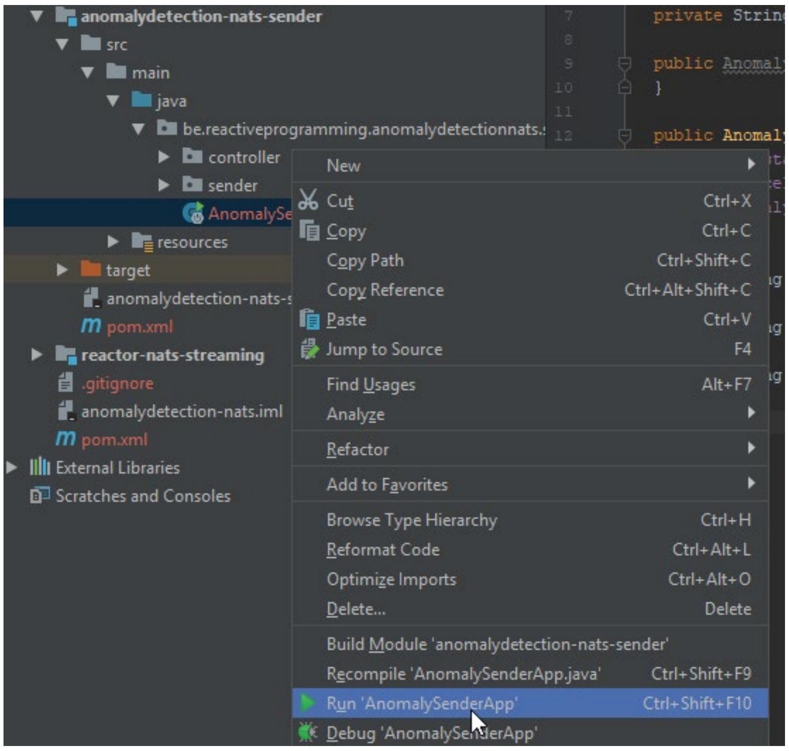
Modules

The project is split up into different modules, that map to the different parts of the architecture we saw in the architectural overview. These can be run on the same development PC but in case we want a more scalable architecture, we'd spread them to different servers.



```
anomalydetection-nats C:\DEV\GIT\
  .idea
  anomalydetection-nats-client
  anomalydetection-nats-common
  anomalydetection-nats-gateway
  anomalydetection-nats-sender
  reactor-nats-streaming
```

The different modules all have a "runner" class available that use Spring Boot to run the application. If you're not sure what Spring is, be sure to look at the "Spring in 5 minutes" document added to the workshop. Simple run them like a Java application in your IDE when you need to start them up.



Workshop

From Sender to Gateway

We'll start building our application at its logical start, at the boundary between the Sender and Gateway applications.

The Sender application is already fully implemented, so we'll start by taking a look at it. There are a number of TODO README steps that will offer additional information on what's going on in this application.

From Gateway to NATS

Afterwards we'll implement the sendMeasurement method, which will help us send a measurement to the NATS server. We'll be sending it as part of a Reactive Pipeline; the Gateway will only return a success message through its webcontroller after NATS acknowledges the measurement, but the server doesn't have to actively wait for it and can simply continue receiving new web requests.

Follow the TODO steps further to get to this point.

From NATS to Client

Now we get to the "main part" of this workshop. Our MeasurementEvents can stream from the gateway to the NATS server. This means our next step will be filtering and transforming to the user.

Let's start by doing exactly that. Follow the TODO's further to implement the last bits of our Reactive Streaming application, leading from our event store to our user.