Relatório 02 - sensor-microservice

Felipe Gabriel Comin Scheffel - RA117306
Ciência da Computação - DIN - UEM
6920 - Sistemas Distribuídos
Prof. Raqueline Penteado

RabbitMQ

O RabbitMQ é um poderoso software de mensageria de código aberto, desenvolvido para facilitar a comunicação entre diferentes sistemas e aplicações distribuídas. Ele implementa o padrão de filas de mensagens, onde os produtores enviam mensagens para uma fila e os consumidores as recebem de lá.

O RabbitMQ opera com base em um modelo de troca de mensagens entre diferentes partes do sistema. As principais entidades em um ambiente RabbitMQ incluem:

- Produtores (Producers): São os componentes responsáveis por enviar mensagens para o RabbitMQ.
- Filas (Queues): São estruturas de dados que armazenam as mensagens enviadas pelos produtores. As mensagens são armazenadas de forma persistente até que sejam processadas pelos consumidores.
- Consumidores (Consumers): São os componentes que recebem mensagens das filas e as processam. Eles consomem as mensagens de acordo com a lógica de negócios implementada.
- Trocas (Exchanges): São responsáveis por receber mensagens dos produtores e decidir para qual fila elas devem ser roteadas. Existem diferentes tipos de trocas, como trocas de diretivas, trocas de tópicos e trocas de fanout, que determinam o comportamento de roteamento das mensagens.
- Vínculos (Bindings): Estabelecem a relação entre trocas e filas, especificando como as mensagens devem ser roteadas.

O RabbitMQ opera com o protocolo AMQP (Advanced Message Queuing Protocol), que define a forma como as mensagens são produzidas, entregues e consumidas.

No geral, o RabbitMQ fornece uma infraestrutura robusta e flexível para a troca de mensagens entre sistemas distribuídos. Ele permite que os desenvolvedores construam aplicativos escaláveis e confiáveis, facilitando a comunicação assíncrona entre diferentes componentes do sistema.

Descrição do sistema

A aplicação a ser implementada se trata de um sistema de gerenciamento de sensores de informações meteorológicas. A partir da simulação de sensores que enviam medições coletadas, o sistema deverá ser capaz de receber e armazenar os dados, para então processar e apresentar novas informações calculadas a partir das medições.

Nessa fase do projeto a aplicação será composta de dois microserviços: um deles será responsável pelos dados, com funcionalidades de cadastro, deleção e leitura, enquanto o outro será responsável pelo processamento e relatório.

Implementação

Arquitetura da aplicação

O sistema foi implementado em JavaScript no ambiente de runtime Node.JS. Foram utilizadas algumas bibliotecas, com destaque para o framework de desenvolvimento web `express`, que permite a criação rápida de APIs RESTful; o `mongoose`, para conexão com um banco de dados MongoDB; e a `amqplib`, para conexão e operações com o RabbitMQ, possibilitando a comunicação indireta.

Cada microserviço foi criado como um projeto Node separado. Ou seja, cada microserviço é executado em um processo separado. Ambos os processos funcionam como servidores web, expondo endpoints para uma API RESTful. Os servidores são configurados em `server.js`, enquanto a estrutura de cada serviço é definida nos diretórios `api`.

A imagem abaixo é a documentação gerada no Postman de ambas as APIs, incluíndo as requests e respostas possíveis.

Diferentemente da primeira fase do projeto, o microserviço `sensor-service` não irá mais manter uma lista em memória para armazenar as medições recebidas, mas sim realizar as operações de cadastro, deleção e leitura diratamente no banco de dados.

A comunicação passa a ser feita seguindo o modelo publicador-consumidor. O `sensor-service` é responsável por inserir toda nova medição na fila, e o `report-service` consome a fila para se manter atualizado.

Como o `report-service` não se comunica mais diretamente com o serviço de dados, ele passa a manter um máximo de 20 medições mais recentes em memória, para poder realizar calculos como a temperatura média.

Outros detalhes implementados foram: - Ao ser iniciado, o `sensor-service`publicas as leituras salvas em seu banco nas últimas 24h. Assim, caso o `report-service` também tenha sido desligado (e por consequência perdido as leituras que tinha em memória), ele pode ser atualizar novamente. - Ao consumir uma publicação, o `report-service` verifica se ela já não havia sido salva em sua memória. De

qualquer caso, é feito o ack da mensagem, já que ele a recebeu, apenas não é necessário tratá-la.

Persistência

A conexão com o banco de dados MongoDB é feita no arquivo `db/index.js`:

```
async function connectMongoose() {
  try {
    await mongoose.connect(connUri);
    console.log('Mongoose connected');
  } catch (e) {
    console.log(`Error connecting to mongoose: ${e}`);
  }
}

async function initialLoad() {
  await connectMongoose();
}

initialLoad();
```

E o modelo para a entidade SensorReading, que representa uma leitura de sensor, é definida no arquivo `db/model/sensorReading`:

```
const sensorReadingSchema = new mongoose.Schema({
  temperature: {
    type: Number,
    required: true,
  },
  relativeHumidity: {
    type: Number,
    required: true,
  },
  windSpeed: {
    type: Number,
    required: true,
  },
  timestamp: {
    type: Date,
    required: true,
    default: Date.now,
  },
});
```

As operações realizadas pelo `sensor-service` no banco são:

```
// POST /reading - salva nova leitura
```

```
const reading = new SensorReading({
    temperature,
    relativeHumidity,
    windSpeed,
    timestamp
    });

await reading.save();

// DELETE /readings - excluí todas as leituras
await SensorReading.deleteMany({});

// GET /readings - retorna todas as leituras
readings = await SensorReading.find();

// GET /last-reading
reading = await SensorReading.findOne({}, null, { sort: { timestamp: -1 } });
```

Comunicação

Com o servidor do RabbitMQ rodando, ambos os serviços irão se conectar ao servidor e criar um canal para a fila `readings_queue`.

```
const rabbitmqUrl = process.env.RABBITMQ_CONN;
const queueName = process.env.RABBITMQ_QNAME;

async function start(api, db) {
    const connection = await amqp.connect(rabbitmqUrl);
    const channel = await connection.createChannel();

    await channel.assertQueue(queueName, { durable: true });
    // ...
}
```

O `sensor-service` realiza publicações ao receber uma requisição POST com um nova leitura, ou assim que é iniciado, publicando as leituras ddas últimas 24h persistidas no banco.

```
channel.sendToQueue(queueName, Buffer.from(JSON.stringify(reading)), { persistent: true });
```

O `report-service` "assina" ao canal ao ser iniciado, se declarando como consumidor para qualquer publicação que seja feita.

```
function connectQueue(channel) {
  channel.consume(queueName, msg => {
    const reading = JSON.parse(msg.content.toString())
    if (!(readings.some(r => r._id === reading._id))) {
      readings.unshift(reading);
```

```
      console.log(`Consumed reading with timestamp ${reading.timestamp}`);
    }
    if (readings.length > 20)
      readings.pop();
    channel.ack(msg);
  });
}

module.exports = (app, channel) => {
  connectQueue(channel);

  // ...
}
```