01/02/2025

Communication-interservices

Identifier la communication entre les services, le protocole AMQP. Installer et exploiter RABBITMQ.

MAHDI KELLOUCH
ISMO TETOUAN

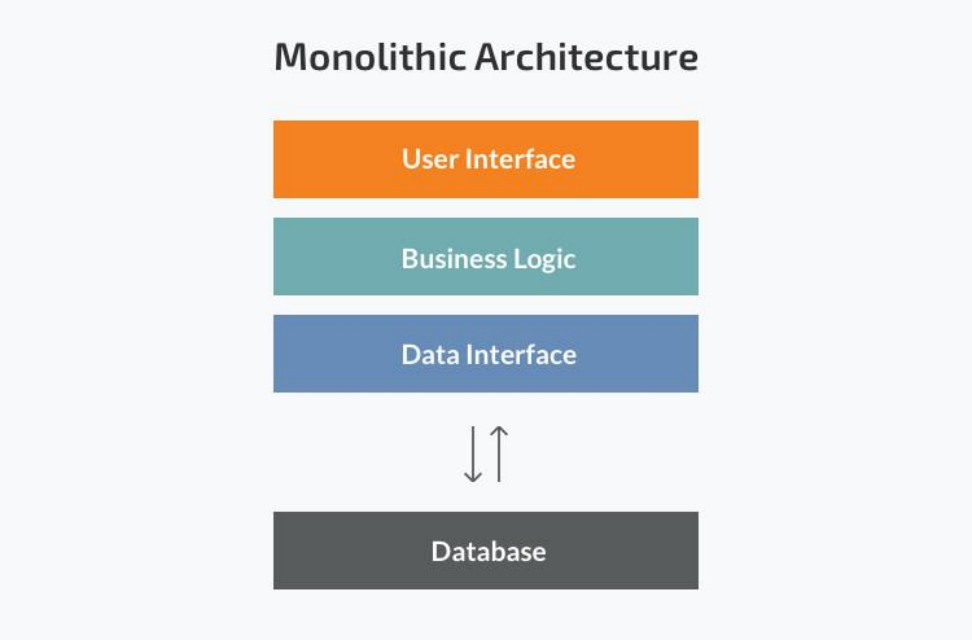
Table des matières

Communication-interservices

1. Microservices

1.1 Les applications monolithiques

Une application **monolithique** est une application qui est développée en un **seul bloc**, avec une même technologie et déployée dans un serveur d'application.



Les difficultés qu'on peut rencontrer avec l'architecture monolithique :

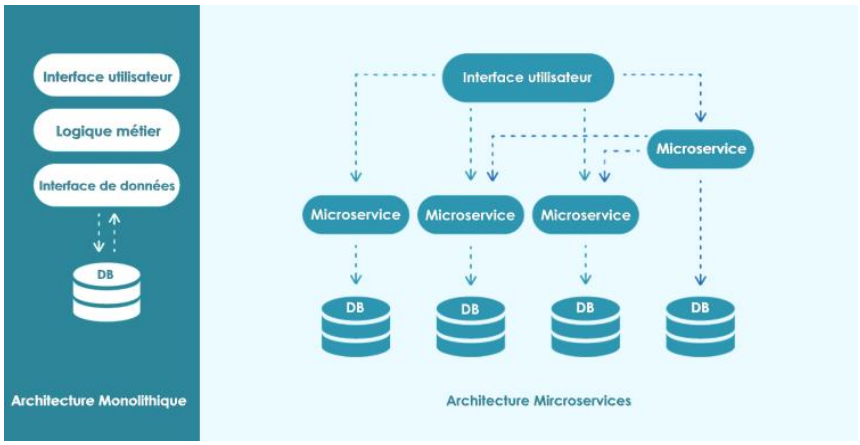
- **Complication du déploiement** : tout changement dans n'importe quel module de l'application nécessite le redéploiement de toute l'application et menace, par conséquent, son fonctionnement intégral.
- **Scalabilité non optimisée :** La seule façon d'accroître les performances d'une application conçue en architecture monolithique, suite à une augmentation de trafic par exemple, est de la redéployer plusieurs fois sur plusieurs serveurs. Or, dans la majorité des cas, on a besoin d'augmenter les performances d'une seule fonctionnalité de l'application. Mais en la redéployant sur plusieurs serveurs, on accroîtra les performances de toutes les fonctionnalités, ce qui peut être non-nécessaire et gaspiller les ressources de calcul.

➔ Ces deux difficultés principales ont donné naissance à l'architecture **microservices**

1.2 L'architecture microservice

Alors qu'une application monolithique est **une seule unité unifiée**, une architecture microservices la décompose en un ensemble de **petites unités indépendantes.**

Ces unités exécutent chaque processus d'application comme un service distinct. Ainsi, tous les services possèdent leur propre logique et leur propre base de données et exécutent les fonctions spécifiques.

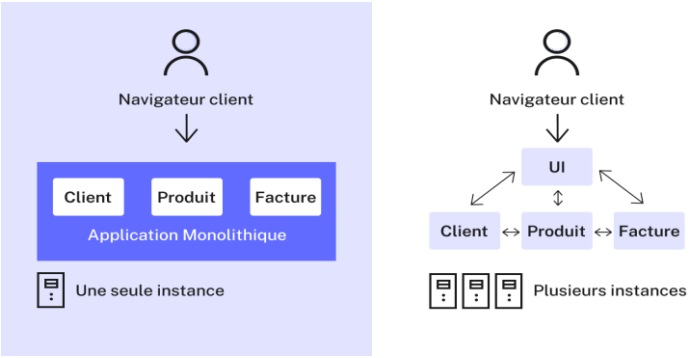


Exemple :

Prenons l'exemple d'un site de vente en ligne des produits : l'application est composée de plusieurs modules, à savoir : client, produit et facture.

Pour une application monolithique, ces modules seront déployés dans un seul serveur. En revanche, pour une architecture microservices, chacun de ces modules constituera un service à part avec sa propre base de données. Ces services peuvent être déployés dans des infrastructures différents (sur site, en cloud …)

Ainsi, un dysfonctionnement du service « facture », par exemple, n'arrêtera pas le fonctionnement de l'ensemble du système.



1.3 Agilité technique

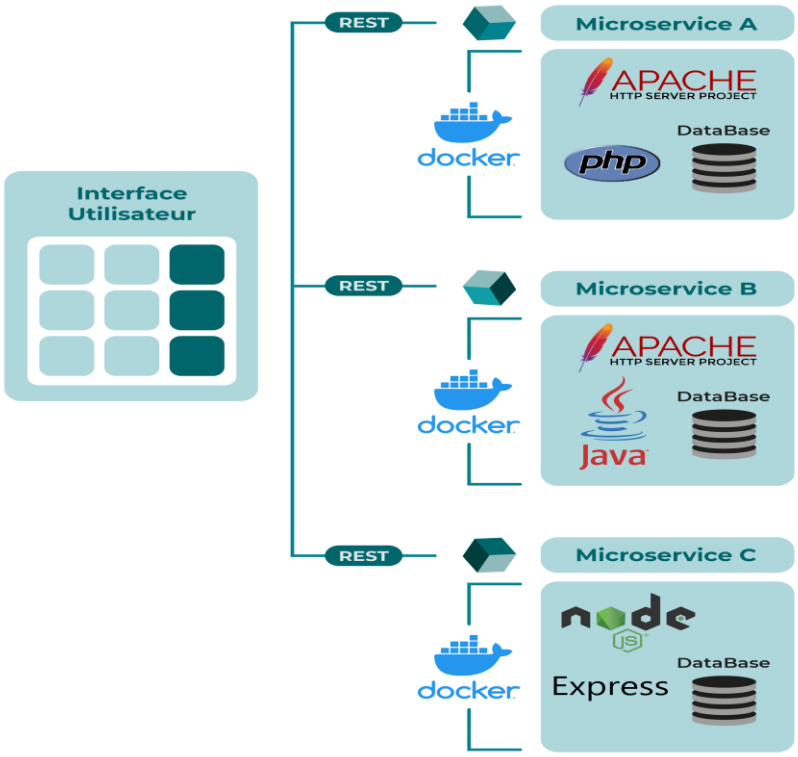
Le choix technologique dans une architecture Microservices est totalement ouvert. Il dépend majoritairement des besoins, de la taille des équipes et de leurs compétences et peut être adapté aux besoins spécifiques de chaque micro-service

Chaque microservice est parfaitement **autonome :** il a sa propre base de données, son propre serveur d'application (Apache, Tomcat, Jetty, etc.), ses propres librairies …

La plupart du temps, ces microservices sont chacun dans un container Docker, ils sont donc totalement indépendants y compris vis-à-vis de la machine sur laquelle ils tournent.

L'utilisation de conteneurs, permettra ainsi un déploiement continue et rapide des microservices.

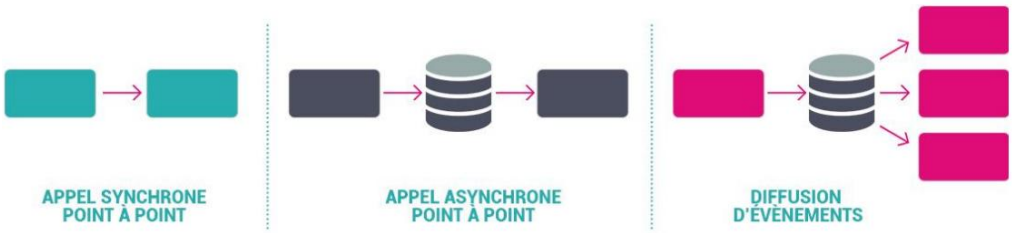
Voici un exemple simplifié d'une architecture microservice :



2. Communication entre services

On distingue trois types de modes de communication entre services :

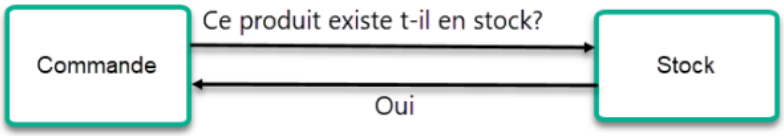
- Appel synchrone point à point
- Diffusion de messages asynchrones point à point
- Diffusion d'événements



2.1 Appel synchrone point à point

Un service appelle un autre service et attend une réponse.

➔ Ce type de communication est utilisé lorsqu'un service émetteur a besoin de la réponse pour continuer son processus

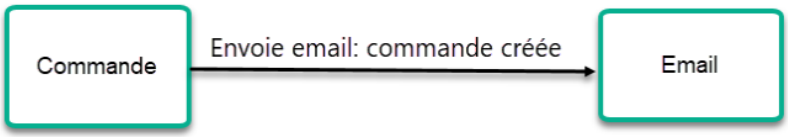


2.2 Diffusion de messages asynchrones point à point

Un service appelle un autre service et continue son processus

Le service émetteur n'attend pas de réponse : Fire and forget

➔ Ce type de communication est utilisé lorsqu'un service désire envoyer un message à un autre service



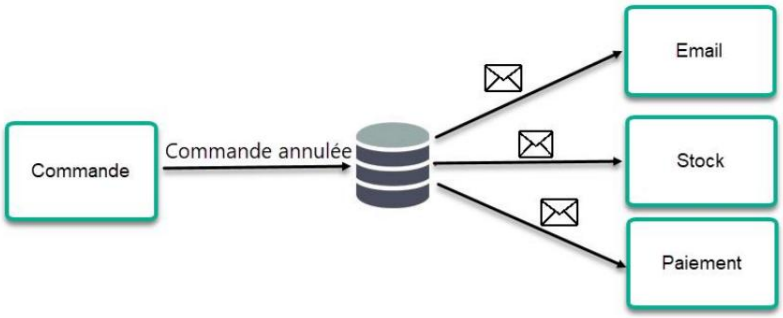
2.3 Diffusion d'événements

Quand un service désire envoyer une notification aux autres services.

Il n'a aucune idée des services écoutant cet événement (listen)

Il n'attend pas de réponse : Fire and forget

➔ Ce type de communication est utilisé quand un service veut notifier tout le système par un évènement



3. Le protocole AMQP

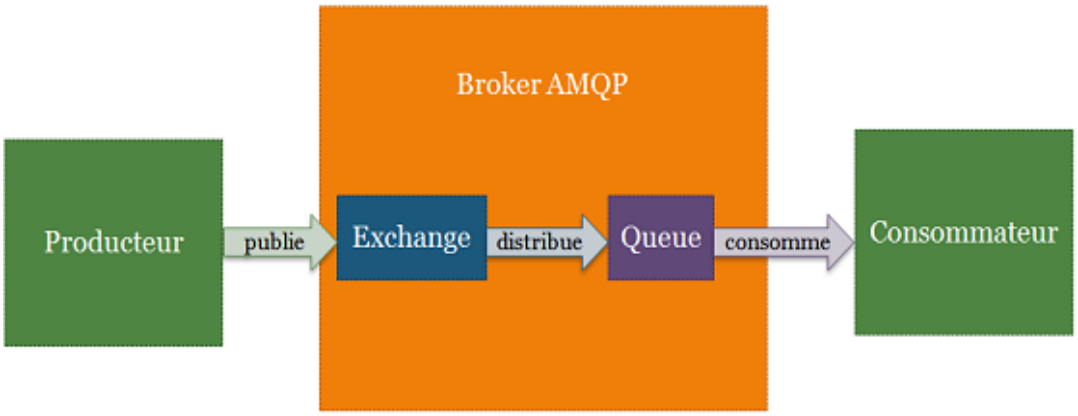
3.1 Définition

AMQP est l'abréviation de Advanced Message Queuing Protocol

AMQP est un protocole open source pour les systèmes de messagerie asynchrone par réseau. L'échange de messages est chiffré et interopérable entre les applications avec ce protocole.

3.2 Le modèle AMQP

La figure suivante qui représente le modèle AMQP souligne les principes de ce protocole.



Dans ce modèle, les applications client sont représentées par les termes producteur et consommateur :

- **Le rôle du producteur** est d'envoyer un message au broker à destination du consommateur, on dit qu'il publie.
- **Le rôle du consommateur** est de recevoir un message à partir du broker, on dit qu'il consomme.
- **Le rôle du broker** est de réceptionner les messages envoyés et de les livrer aux destinataires spécifiés.

4. RABBITMQ

4.1 Définition

RabbitMQ est un broker de messages se basant sur le standard AMQP afin d'échanger avec différents clients.

4.2 Installation

Nous allons utiliser dans nos applications l'image docker rabbitmq pour créer des conteneurs comportant le brocker RabbitMQ. Voici la commande pour créer un conteneur a partir de l'image rabbitmq :

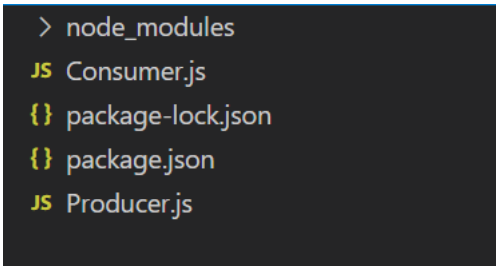
```
docker run --name rabbitmq -d -p 15672:15672 -p 5672:5672 rabbitmq:3-management
```

4.3 Première application

Dans un dossier, on lance les commandes suivantes :

```
npm init –y

npm install amqplib
```

Dans le même dossier, on crée les deux fichiers suivants : producer.js et consumer.js :



Puis, on écrit le programme du Producer.js :

```
const amqp =require('amqplib') ;

let channel, connection;
const queueName = "FirstQueue";
const Message = "Mon message";

// Connect to RabbitMQ
async function connectToRabbitMQ() {
    const amqpServer = "amqp://guest:guest@localhost:5672";
    connection = await amqp.connect(amqpServer);
    channel = await connection.createChannel();
    await channel.assertQueue(queueName);
 }

connectToRabbitMQ().then(() => {
    channel.sendToQueue(queueName,
        Buffer.from(Message)
    );
});

setTimeout(() => {
    connection.close();
}, 1000)
```

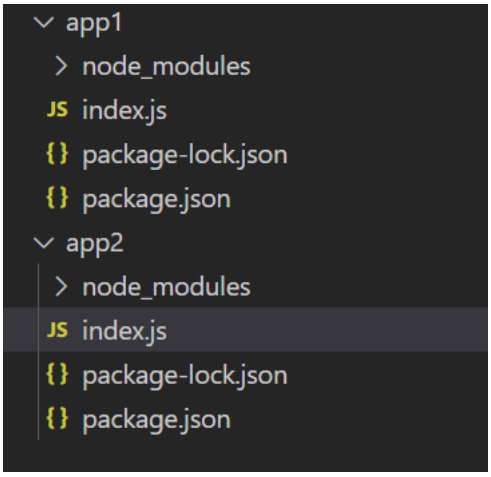
Voici le code du programme Consumer.js :

```
const amqp =require('amqplib') ;

let channel, connection;
const queueName = "FirstQueue";


// Connect to RabbitMQ
async function connectToRabbitMQ() {
    const amqpServer = "amqp://guest:guest@localhost:5672";
    connection = await amqp.connect(amqpServer);
    channel = await connection.createChannel();
    await channel.assertQueue(queueName);
 }

connectToRabbitMQ().then(() => {
    channel.consume(queueName, (data) => {
        console.log("Consumed from " + queueName + " - " +
data.content.toString());
```

```
        channel.ack(data);
      });
});
```

4.4 Micro-services avec express.js et RabbitMQ

Voici la structure du projet que nous allons créer dans cette section :



Les commandes suivantes seront lancées dans les deux dossiers app1 et app2 :

```
npm init –y

npm install express amqplib
```

Le programme de index.js du dossier App1 :

```
import express from 'express';
import amqp from 'amqplib';

const app = express()

const queueName1 = "testq1";
const queueName2 = "testq2";
let connection, channel;

async function connectToRabbitMQ() {
    const rabbitMQUrl = 'amqp://localhost';
    connection = await amqp.connect(rabbitMQUrl);
    channel = await connection.createChannel();
    await channel.assertQueue(queueName1);
    await channel.assertQueue(queueName2);
 }
 connectToRabbitMQ();

app.get('/:id', (req,res) => {
    switch(req.params.id){
        case "1":
            channel.sendToQueue(queueName1,Buffer.from("Message 1"));
        break;
        case "2":
            channel.sendToQueue(queueName2,Buffer.from("Message 2"));
```

```
        break;
    }

    res.sendStatus(200);
})

app.listen(3000)
```

Le programme de index.js du dossier App2 :

```
import amqp from 'amqplib';

const queueName1 = "testq1";
const queueName2 = "testq2";
let connection, channel;

async function connectToRabbitMQ() {
    const rabbitMQUrl = 'amqp://localhost';
    connection = await amqp.connect(rabbitMQUrl);
    channel = await connection.createChannel();
    await channel.assertQueue(queueName1);
    await channel.assertQueue(queueName2);
}
connectToRabbitMQ().then(() => {
    channel.consume(queueName1, (data) => {
        console.log("Consumed from " +queueName1+ " - " + data.content);
        channel.ack(data);
    });

    channel.consume(queueName2, (data) => {
        console.log("Consumed from " +queueName2+ " - " + data.content);
        channel.ack(data);
    });
});
```