CALC : Projet RabbitMQ

Tchadel Icard

8 janvier 2025

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Table des matières

1 Introduction

Les systèmes de messagerie instantanée occupent aujourd'hui une place centrale dans nos interactions personnelles et professionnelles. Des applications comme WhatsApp, Slack ou Microsoft Teams démontrent à quel point une communication rapide, fiable et en temps réel est devenue indispensable dans nos sociétés modernes. Cependant, concevoir une telle application pose des défis techniques complexes : garantir une faible latence, gérer une montée en charge efficace, maintenir une communication en temps réel même en cas de panne, et assurer une évolutivité à long terme.

Dans ce contexte, je me suis fixé pour objectif de développer une application de messagerie instantanée reposant sur une architecture *microservices* et utilisant `RabbitMQ` comme *middleware* de messagerie. Ce choix technologique répond à deux besoins fondamentaux : la gestion efficace des communications asynchrones et le passage à l'échelle d'un système distribué. Le projet sert également d'étude de cas pour explorer les bonnes pratiques dans le développement de systèmes distribués modernes.

Ce document présente les étapes clés de ce projet, de la définition des enjeux techniques et fonctionnels à la mise en œuvre d'une solution complète et évolutive.

2 Contexte et enjeux

Les systèmes de messagerie instantanée sont complexes à concevoir, car ils doivent répondre à des attentes élevées en matière de performance, de fiabilité et de flexibilité. Ces exigences sont encore accentuées par la diversité des scénarios d'utilisation : messages privés, discussions de groupe, notifications en temps réel, et intégration multiplateforme.

2.1 Contexte

Les outils de communication modernes doivent répondre à des besoins croissants :

— Temps réel : la latence perçue par les utilisateurs doit être réduite au minimum.
— Fiabilité : chaque message envoyé doit atteindre son destinataire, même en cas de panne temporaire d'un composant.
— Passage à l'échelle : le système doit pouvoir gérer un nombre croissant d'utilisateurs et de messages sans compromettre ses performances.
— Flexibilité : les fonctionnalités doivent pouvoir évoluer en fonction des besoins des utilisateurs, sans affecter les autres composantes du système.

Dans ce projet, nous avons opté pour une architecture *microservices* et l'intégration de `RabbitMQ` pour répondre à ces exigences. Ces choix ont été motivés par leur capacité à séparer les responsabilités fonctionnelles, à faciliter la communication entre services, et à offrir une base solide pour construire un système distribué.

2.2 Enjeux techniques

Pour répondre au cadre décrit, plusieurs défis techniques ont été identifiés :

1. Gestion des communications asynchrones : garantir que les messages sont livrés dans le bon ordre et sans perte, tout en maintenant une faible latence.
2. Conception modulaire : diviser les responsabilités en services indépendants, chacun étant autonome et capable d'évoluer ou d'être remplacé sans impact majeur sur l'ensemble.
3. Support de la montée en charge : ajouter dynamiquement des instances de services pour gérer des pics d'activité ou une croissance continue du nombre d'utilisateurs.
4. Convivialité pour l'utilisateur final : concevoir une interface intuitive et performante avec des mises à jour des messages en temps réel via WebSocket.

2.3 Positionnement du projet

Ce projet vise à appliquer ces principes dans le cadre d'une application de messagerie instantanée, tout en explorant les technologies modernes suivantes :

1. Golang : un langage performant, particulièrement adapté aux systèmes distribués et aux architectures concurrentes.
2. RabbitMQ : *middleware* assurant une gestion fiable des files de messages et du routage.
3. JavaScript avec React : création d'une interface utilisateur dynamique.

Ces technologies, combinées à une architecture bien conçue, permettent de construire un système évolutif et robuste, capable de répondre aux exigences modernes de la messagerie en temps réel.

3 Architecture

L'architecture de l'application repose sur trois services principaux : l'API, le service utilisateur, et le service message. Ces services interagissent de manière asynchrone à l'aide de RabbitMQ, tout en sauvegardant leurs données dans une base de données PostgreSQL pour garantir la persistance des informations critiques. Cette combinaison permet de répondre aux exigences de fiabilité, de modularité et de passage à l'échelle nécessaires à une application de messagerie instantanée.

Cette section détaille les rôles et interactions des trois services, ainsi que les mécanismes utilisés pour assurer une communication fluide et une gestion pérenne des données.

3.1 Service API

Le service API agit comme point d'entrée principal du système. Il est responsable de :

— L'authentification des utilisateurs : inscription (*register*) et connexion (*login*).
— La gestion des WebSockets : maintenir une connexion bidirectionnelle avec le *frontend* pour toutes les interactions pendant et après l'authentification.

3.1.1 Authentification asynchrone avec RabbitMQ

Les opérations `register` et `login` sont réalisées de manière asynchrone pour maximiser les performances et le passage à l'échelle. Lorsqu'une requête est envoyée par le *frontend* :

1. Une file d'attente temporaire est créée pour cet utilisateur, reliée à l'*exchange* `notification`.
2. La requête est publiée dans la file d'attente appropriée (`register` ou `login`) via l'*exchange* direct associé au service utilisateur.
3. Le service utilisateur traite la requête et renvoie une réponse via RabbitMQ, qui est transmise au *frontend* via le WebSocket par le service API.

3.1.2 Communication bidirectionnelle via WebSocket

Une fois connecté, toutes les communications entre le *frontend* et l'API se font exclusivement via WebSocket. Cela permet :

1. De maintenir un échange en temps réel entre le client et le serveur.
2. De recevoir instantanément les mises à jour (messages, notifications, etc.).

3.2 Service Utilisateur

Le service utilisateur est au cœur de la gestion des données liées aux utilisateurs. Ce service est conçu pour être à la fois indépendant et passer à l'échelle, et il sauvegarde toutes les informations dans une base de données PostgreSQL.

3.2.1 Rôles principaux

Le service utilisateur prend en charge les opérations suivantes :

1. Inscription et connexion : validation des données utilisateur et la création d'un token d'authentification (token JWT) pour le *frontend*.
2. Récupération de la liste des utilisateurs : permettre au *frontend* de construire la liste des contacts.
3. Récupération des informations de l'utilisateur courant (`getSelf`) : fournir les informations détaillées de l'utilisateur connecté.

3.2.2 Interaction avec RabbitMQ

Les échanges passent par un *exchange* direct dédié, avec plusieurs files d'attente pour des opérations spécifiques :

1. File d'attente `register` : reçoit les requêtes d'inscription.
2. File d'attente `login` : reçoit les requêtes de connexion.

3. File d'attente `getUsers` : requêtes de type RPC permettant de récupérer la liste complète des utilisateurs.
4. File d'attente `getSelf` : requêtes de type RPC pour récupérer les informations du profil de l'utilisateur connecté.

3.2.3 Persistance des données

Toutes les informations utilisateur sont stockées de manière pérenne dans une base de données PostgreSQL. Cela garantit :

1. La possibilité de retrouver les données même en cas de redémarrage des services.
2. Une flexibilité pour exécuter des requêtes complexes ou des analyses sur les données utilisateur.

3.3 Service Message

Le service message gère toutes les fonctionnalités liées à la messagerie, de l'envoi à la réception des messages, en passant par la gestion des conversations. Comme pour le service utilisateur, toutes les données sont sauvegardées dans une base de données PostgreSQL, garantissant la pérennité des messages échangés.

3.3.1 Rôles principaux

Le service message se charge de :

1. Envoyer des messages : routage des messages vers les files des destinataires appropriés via RabbitMQ.
2. Récupérer les messages d'une conversation : permettre au *frontend* d'afficher l'historique des messages.

3.3.2 Interaction avec RabbitMQ

Le service message utilise également un *exchange* direct dédié pour gérer les opérations suivantes :

1. File d'attente `getMessages` : requêtes de type RPC pour récupérer l'historique d'une conversation.
2. File d'attente `sendMessage` : requêtes de type RPC pour traiter l'envoi de messages.

3.3.3 Diffusion via un *exchange fanout*

Pour la diffusion des messages, un *exchange fanout* est utilisé :

1. Lorsqu'un utilisateur envoie un message, ce dernier est publié sur l'*exchange*.

2. Les destinataires potentiels reçoivent le message via leurs files d'attente respectives.
3. Le message est transmis uniquement via WebSocket à l'utilisateur connecté correspondant.

3.3.4 Persistance des messages

Tous les messages sont sauvegardés dans une base de données PostgreSQL pour :

1. Conserver un historique complet des conversations.
2. Assurer la disponibilité des messages en cas de reconnexion de l'utilisateur ou de reprise après une panne.

3.4 Passage à l'échelle et déploiement

L'architecture est conçue pour passer facilement à l'échelle grâce à l'utilisation combinée de RabbitMQ et de Docker :

1. Multiples instances : plusieurs instances de l'API, du service utilisateur et du service message peuvent être déployées en parallèle.
2. Équilibrage de charge : RabbitMQ distribue automatiquement les requêtes entre les instances disponibles.
3. Conteneurisation : chaque service est déployé dans un conteneur Docker, garantissant une isolation complète et facilitant les mises à jour ou le déploiement dans différents environnements.

3.5 Communication entre les services

Les flux de messages entre les différents composants sont illustrés dans les Figures 1 et 2.

Requêtes (Figure 1) : ce diagramme montre comment les requêtes sont envoyées depuis le *frontend*, via l'API, et routées vers les services utilisateur et message à travers RabbitMQ. Chaque opération, comme `login`, `register` ou `getMessages`, est associée à une file d'attente spécifique, garantissant une séparation claire des responsabilités et une gestion modulaire des échanges.

Réponses (Figure 2) : ce diagramme illustre le retour des réponses des services *backend* vers l'API, en passant par un *exchange* de notification dédié. Il met également en évidence l'utilisation d'un *exchange fanout* pour diffuser les messages d'une conversation à tous les participants concernés, tout en filtrant les messages côté *frontend* pour s'assurer qu'ils ne sont livrés qu'aux destinataires appropriés. **Réponses (Figure 2)** : ce diagramme illustre le retour des réponses des services *backend* vers l'API, en passant par un *exchange* de notification dédié. Il met également en évidence l'utilisation d'un *exchange fanout* pour diffuser les messages d'une conversation à tous les participants concernés, tout en filtrant les messages côté *frontend* pour s'assurer qu'ils ne sont livrés qu'aux destinataires appropriés.

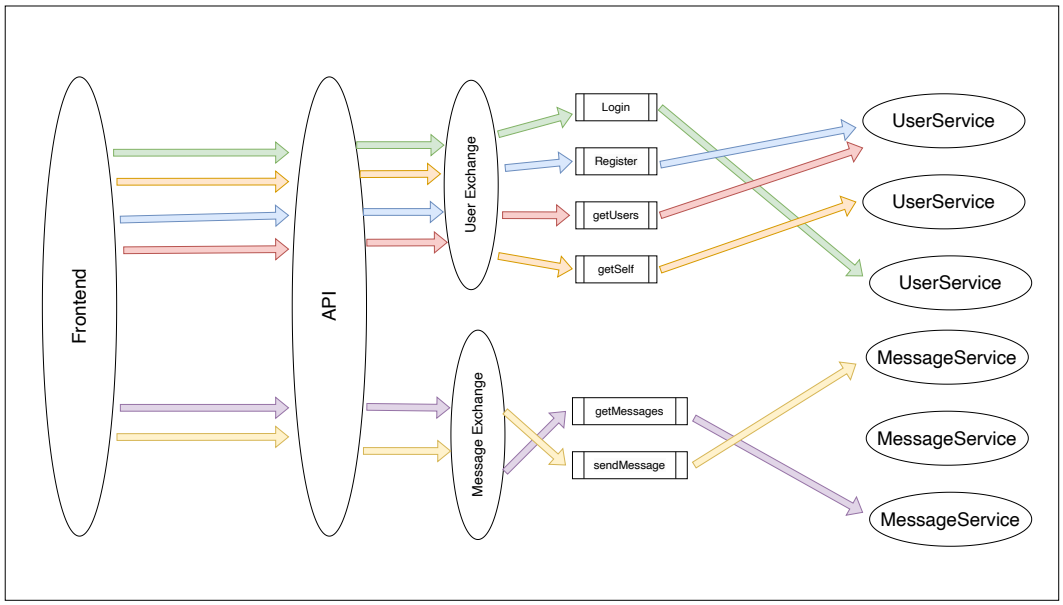

FIGURE 1 – Diagramme montrant les requêtes entre les services.

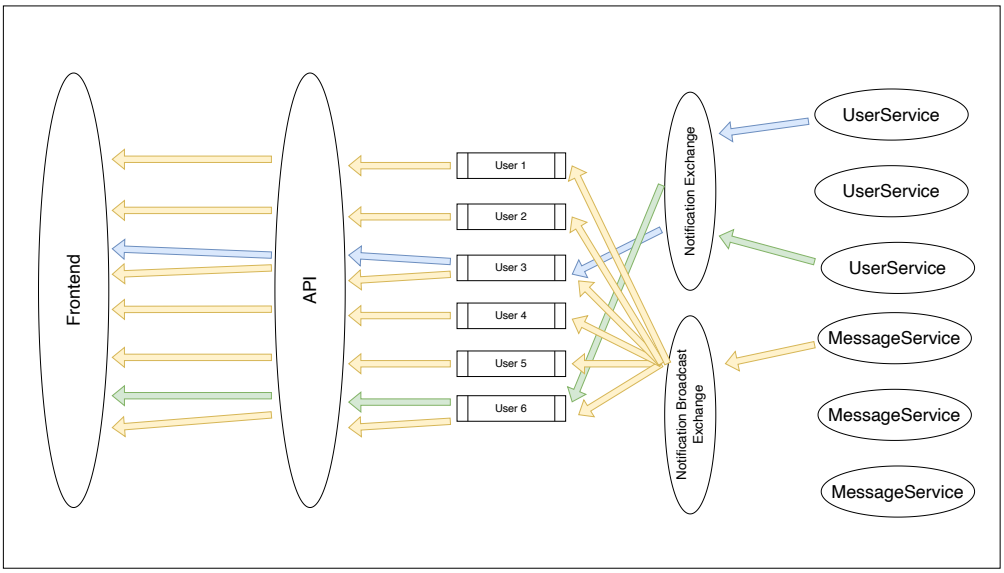

FIGURE 2 – Diagramme montrant les réponses entre les services.

Ces diagrammes mettent en lumière la structure organisée et performante de la communication asynchrone entre les services, rendue possible par RabbitMQ. Ils soulignent également la flexibilité de l'architecture, qui permet de gérer efficacement les requêtes et les réponses, même dans des environnements distribués à haute charge.

4 Utilisation

L'application de messagerie instantanée permet de communiquer en temps réel via des messages privés ou de groupe. Elle repose sur une architecture distribuée, utilisant RabbitMQ pour la gestion des messages et PostgreSQL pour la persistance des données. Voici les instructions pour utiliser l'application.

4.1 Lancer l'application

Pour démarrer l'application, suivez les étapes suivantes :

1. Clonez le dépôt Git :

```
git clone https://github.com/tchadelicard/instant-messaging-app
cd instant-messaging-app
```

2. Lancez les services avec Docker Compose :

```
docker compose up --build -d
```

3. Accédez aux services :
   — **Frontend (interface utilisateur)** : `http://localhost:3000`
   — **Backend API** : `http://localhost:8080`
   — **Interface RabbitMQ** : `http://localhost:15672` (utilisateur : `guest`, mot de passe : `guest`)

4.2 Noms des services Docker

Les services Docker définis dans le projet peuvent être contrôlés individuellement à l'aide des commandes Docker Compose. Voici leurs noms :

— **postgres** : service pour la base de données PostgreSQL.
— **rabbitmq** : service pour le gestionnaire de messages RabbitMQ.
— **frontend** : service pour l'interface utilisateur développée avec React.
— **api** : service pour le backend principal (API REST).
— **user-service-1** : service dédié à la gestion des utilisateurs.
— **message-service-1** : service dédié à la gestion des messages.

Commandes Docker utiles

— **Afficher les journaux d'un service :**

```
docker compose logs <nom_du_service>
```

Exemple pour le service `frontend` :

```
docker compose logs frontend
```

— **Redémarrer un service :**

```
docker compose restart <nom_du_service>
```

Exemple pour le service `api` :

```
docker compose restart api
```

— **Démarrer ou arrêter un service :**
  — Démarrer :

  ```
  docker compose up -d <nom_du_service>
  ```

  — Arrêter :

  ```
  docker compose stop <nom_du_service>
  ```

4.3 Commandes utiles

Voici quelques commandes pour gérer l'application via Docker Compose :

— **Démarrer l'application** :

```
docker compose up -d
```

— **Recréer les conteneurs après une mise à jour** :

```
docker compose up --build -d
```

— **Arrêter l'application** :

```
docker compose down
```

— **Afficher les journaux d'un service (exemple : frontend)** :

```
docker compose logs frontend
```

4.4 Démonstration

Un exemple de scénario d'utilisation de l'application est décrit ci-dessous :

1. L'utilisateur se connecte à l'application via l'interface frontend.
2. Il effectue une recherche d'utilisateur.
3. Les messages envoyés ou reçus sont traités en temps réel grâce à RabbitMQ.
4. Les données (utilisateurs et messages) sont sauvegardées dans PostgreSQL pour garantir leur persistance.

5 Résultats

L'application de messagerie instantanée a été déployée avec succès en utilisant une architecture distribuée. Elle intègre plusieurs services communiquant entre eux via RabbitMQ, garantissant la transmission des messages en temps réel avec fiabilité. Voici les principaux résultats obtenus.

5.1 Fonctionnalités principales validées

— **Inscription et connexion utilisateur** : les utilisateurs peuvent s'inscrire avec un nom d'utilisateur et un mot de passe, puis se connecter à l'application pour accéder à leurs messages.
— **Messagerie en temps réel** : les messages envoyés par un utilisateur sont transmis instantanément au destinataire grâce à RabbitMQ, assurant une faible latence et une fiabilité élevée.
— **Persistance des données** : toutes les informations relatives aux utilisateurs et messages sont stockées dans la base de données PostgreSQL, garantissant leur récupération en cas de redémarrage des services.

5.2 Résultats observés

Les tests effectués sur l'application ont confirmé les points suivants :

1. Les messages privés sont transmis en temps réel sans perte de données.
2. La base de données PostgreSQL assure une persistance fiable des informations, même en cas de redémarrage des conteneurs Docker.
3. RabbitMQ gère efficacement la file d'attente des messages, même sous une charge importante simulée par plusieurs utilisateurs simultanés.
4. L'interface utilisateur est fonctionnelle et intuitive, permettant une navigation fluide et rapide.

5.3 Démonstration

Un scénario de démonstration a été réalisé pour valider l'ensemble des fonctionnalités :
— **Étape 1** : un utilisateur s'inscrit, puis se connecte à l'application.
— **Étape 2** : l'utilisateur recherche un autre utilisateur et envoie un message privé.
— **Étape 3** : les messages sont vérifiés dans PostgreSQL pour confirmer leur persistance.
— **Étape 4** : la transmission des messages en temps réel est validée via les journaux de RabbitMQ.

6 Conclusion

L'application de messagerie instantanée développée dans ce projet illustre une architecture distribuée moderne, intégrant RabbitMQ pour la gestion en temps réel des messages et PostgreSQL pour la persistance des données. Cette architecture modulaire garantit la fiabilité et le passage à l'échelle du système. L'interface utilisateur, développée avec React et Tailwind CSS, offre une expérience fluide et intuitive, adaptée aux besoins des utilisateurs.

Ce projet a permis de mettre en œuvre des technologies robustes et d'acquérir des compétences pratiques en conception et déploiement d'applications distribuées. Les résultats obtenus démontrent la capacité du système à fonctionner de manière stable, même sous charge, tout en restant extensible pour de futurs développements. Le code source de l'application est disponible sur GitHub à l'adresse suivante : https://github.com/votre-repo/instant-messaging-app.