

Projet web

Nest.js - GraphQL - RabbitMQ

Jérôme Commaret - Efrei - Juin 2025

Votre intervenant

15 ans d'expérience dans le digital

- 8 ans en gestion de projet
- 7 ans en développement web

Dans des grands groupes (Coca-Cola, GDF-SUEZ, Groupe Publicis), start-ups (LeLynx.fr), ou ESNs (SQLI, Digitas...)

Contributeur sur Void, un « Cursor Opensource » afin d'y ajouter Mistral.AI

Le projet

Une application de messagerie type « Messenger » de Facebook

- Création de profil utilisateur.
- Liste d'utilisateurs (tout les profils créés).
- Liste de conversations.
- Détails de Conversations.

Barème

- 8 points sur la présentation finale (slides, expression orale, explications)
- 12 points sur l'application (fonctionnelle, look and feel, et bonus)

	Slides	Expression	Explication	Fonctionnelle	Look And feel	Bonus
Présentation	4	2	2			
Application				4	4	4

C'est parti !

Faisabilité

Pourquoi Nest.js ?

- **Structure et Organisation** : Nest.js impose une architecture modulaire et organisée (basée sur les modules, contrôleurs, services), ce qui est crucial pour la maintenance et l'évolution d'un projet complexe.
- **TypeScript** : L'utilisation de TypeScript par défaut offre un typage fort, réduisant les erreurs à l'exécution et améliorant l'autocomplétion et la compréhension du code.
- **Écosystème** : Il intègre nativement des solutions pour la plupart des besoins d'un backend : ORM (TypeORM, Prisma), authentication (Passport), et bien sûr, GraphQL.

Pourquoi GraphQL ?

- **Efficacité des Données** : Contrairement à REST, GraphQL permet au client de demander précisément les données dont il a besoin, évitant le "sur-chargement" (over-fetching) ou le "sous-chargement" (under-fetching). C'est idéal pour une application de messagerie où différentes vues (liste de conversations, vue d'une conversation) nécessitent des données différentes.
- **Endpoint Unique** : Toutes les requêtes passent par un seul point d'entrée, ce qui simplifie la gestion de l'API.
- **Temps Réel avec les Subscriptions** : GraphQL inclut nativement un système de "Subscriptions" basé sur les WebSockets, parfait pour pousser les nouveaux messages aux clients en temps réel.

Pourquoi le Message Queuing (MQ) ?

- **Découplage** : Le service qui reçoit les messages (l'API) est découplé du service qui les traite et les distribue. Si le service de distribution tombe en panne, les messages ne sont pas perdus ; ils restent dans la file d'attente.
- **Scalabilité et Résilience** : On peut facilement ajouter plus de "workers" pour traiter les messages en cas de forte charge. Le système garantit que les messages seront livrés même en cas de pic de trafic ou de panne temporaire.
- **Asynchronisme** : Envoyer un message n'a pas besoin de bloquer la requête de l'utilisateur. L'API peut accepter le message, le mettre dans la file d'attente en quelques millisecondes et répondre immédiatement à l'utilisateur.

Configuration (1/2)

- Déterminez vos groupes
- Création de Github et envoi à l'adresse jerome.commaret@intervenants.efrei.net

Configuration (2/2)

1. Installer Node.js : <https://nodejs.org/en/download>

2. Nest.js `npm i -g @nestjs/cli` puis `nest new chat-app`
`cd chat-app`

3. RabbitMQ (dans un fichier docker-compose.yml à la racine du projet)

```
version: '3.8'
services:
  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq
    ports:
      - "5672:5672" # Port pour la communication AMQP
      - "15672:15672" # Port pour l'interface de management web
    environment:
      - RABBITMQ_DEFAULT_USER=user
      - RABBITMQ_DEFAULT_PASS=password
```

puis

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

4. Dépendances GraphQL

```
npm i @nestjs/graphql @nestjs/apollo graphql apollo-server-express
npm i @nestjs/microservices amqplib amqp-connection-manager
```


Architecture

1. Client (Web/Mobile) :

- Interagit avec l'API Nest.js via des requêtes et mutations GraphQL. Écoute les nouveaux messages via une souscription GraphQL.

2. API Nest.js / GraphQL :

- Expose les Queries (ex: getConversations), Mutations (ex: sendMessage) et Subscriptions (ex: onMessageSent).
- Quand une mutation sendMessage est reçue, elle ne distribue pas le message directement. Elle le publie dans une file d'attente RabbitMQ.

2. Worker/Listener Nest.js (Microservice)

- Ce service écoute les messages provenant de la file d'attente RabbitMQ.
- Lorsqu'il reçoit un message, il le sauvegarde en base de données, puis le publie via le PubSub de GraphQL pour que les clients abonnés le reçoivent

Schéma GraphQL (version simplifié)

```
type User {
  id: ID!
  username: String!
  createdAt: String!
}

type Message {
  id: ID!
  content: String!
  author: User!
  createdAt: String!
}

type Conversation {
  id: ID!
  participants: [User!]!
  messages: [Message!]!
}

# Les points d'entrée pour lire des données
type Query {
  me: User
  conversations: [Conversation!]!
  conversation(id: ID!): Conversation
}

# Les points d'entrée pour modifier des données
type Mutation {
  createUser(username: String!): User!
  sendMessage(conversationId: ID!, content: String!): Message!
}

# Les points d'entrée pour le temps réel
type Subscription {
  messageAdded(conversationId: ID!): Message
}
```

API : Nest.js - GraphQL (Résolveur pour les messages - simplifié)

```
import { Args, Mutation, Resolver, Subscription } from '@nestjs/graphql';
import { PubSub } from 'graphql-subscriptions';
import { Message } from '../entities/message.entity'; // Votre entité TypeORM/Prisma
import { MessageService } from '../message.service';

const pubSub = new PubSub(); // En production, utilisez un PubSub basé sur Redis ou autre

@Resolver(() => Message)
export class MessageResolver {
  constructor(private readonly messageService: MessageService) {}

  @Mutation(() => Message)
  async sendMessage(
    @Args('conversationId') conversationId: string,
    @Args('content') content: string,
  ): Promise<Message> {
    // 1. Créer le message et le passer au service
    const newMessage = await this.messageService.createMessage(conversationId, content);

    // 2. Publier l'événement pour les abonnés
    // En réalité, cette logique sera dans le listener RabbitMQ
    pubSub.publish('messageAdded', { messageAdded: newMessage });

    return newMessage;
  }

  @Subscription(() => Message, {
    filter: (payload, variables) =>
      payload.messageAdded.conversationId === variables.conversationId,
  })
  messageAdded(@Args('conversationId') conversationId: string) {
    return pubSub.asyncIterator('messageAdded');
  }
}
```

Intégration RabbitMQ

Etape 1: Configurer le client (src/rabbitmq/rabbitmq.module.ts)

```
import { Module } from '@nestjs/common';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { RabbitMQService } from '../rabbitmq.service';

@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'CHAT_SERVICE',
        transport: Transport.RMQ,
        options: {
          urls: ['amqp://user:password@localhost:5672'],
          queue: 'chat_queue',
        },
      },
    ]),
  ],
  providers: [RabbitMQService],
  exports: [RabbitMQService],
})
export class RabbitMQModule {}
```

Intégration RabbitMQ

Etape 1: Configurer le client dans un module (src/rabbitmq/rabbitmq.module.ts)

```
import { Module } from '@nestjs/common';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { RabbitMQService } from '../rabbitmq.service';

@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'CHAT_SERVICE',
        transport: Transport.RMQ,
        options: {
          urls: ['amqp://user:password@localhost:5672'],
          queue: 'chat_queue',
        },
      },
    ]),
  ],
  providers: [RabbitMQService],
  exports: [RabbitMQService],
})
export class RabbitMQModule {}
```


Intégration RabbitMQ

Etape 2: Créer le service (src/rabbitmq/rabbitmq.service.ts)

```
import { Inject, Injectable } from '@nestjs/common';
import { ClientProxy } from '@nestjs/microservices';

@Injectable()
export class RabbitMQService {
  constructor(@Inject('CHAT_SERVICE') private readonly client: ClientProxy) {}

  public sendMessage(pattern: string, data: any) {
    // 'emit' envoie un événement sans attendre de réponse
    this.client.emit(pattern, data);
  }
}
```

Intégration RabbitMQ

Etape 3: Créer le resolver (src/message/message.resolver.ts)

```
// ... imports, y compris RabbitMQService

// Dans la méthode sendMessage du resolver
@Mutation(() => Message)
async sendMessage(
  @Args('conversationId') conversationId: string,
  @Args('content') content: string,
): Promise<Message> {
  // Logique pour créer le message (sans le sauvegarder finalement)
  const messagePayload = { conversationId, content, authorId: 'currentUserId' };

  // Publier dans RabbitMQ au lieu de sauvegarder directement
  this.rabbitmqService.sendMessage('new_message', messagePayload);

  // On peut retourner une confirmation immédiate
  return { id: 'temp-id', content, ... };
}
```

Intégration RabbitMQ

Etape 3: Créer le listener (src/message/message.controller.ts)

```
import { Controller } from '@nestjs/common';
import { Ctx, EventPattern, Payload, RmqContext } from '@nestjs/microservices';

@Controller()
export class MessageController {

  // ... injecter le service de message, le pubsub...

  @EventPattern('new_message')
  async handleNewMessage(@Payload() data: any, @Ctx() context: RmqContext) {
    console.log(`Received message:`, data);


    // 1. Sauvegarder le message en BDD
    const savedMessage = await this.messageService.saveFinalMessage(data);

    // 2. Publier aux clients via GraphQL Subscription
    pubSub.publish('messageAdded', { messageAdded: savedMessage });

    // 3. Accuser réception du message à RabbitMQ pour qu'il le supprime de la file
    const channel = context.getChannelRef();
    const originalMsg = context.getMessage();
    channel.ack(originalMsg);
  }
}
```


CI/CD

Etape 1 - Dockerfile à la racine



```
# Stage 1: Build the application
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Create the final, smaller image
FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY package*.json ./
# Installez uniquement les dépendances de production
RUN npm install --omit=dev
CMD ["node", "dist/main"]
```

CI/CD

Etape 2 - Github Action

```
name: Deploy to Production

on:
  push:
    branches:
      - main # Déclenche sur un push sur la branche main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

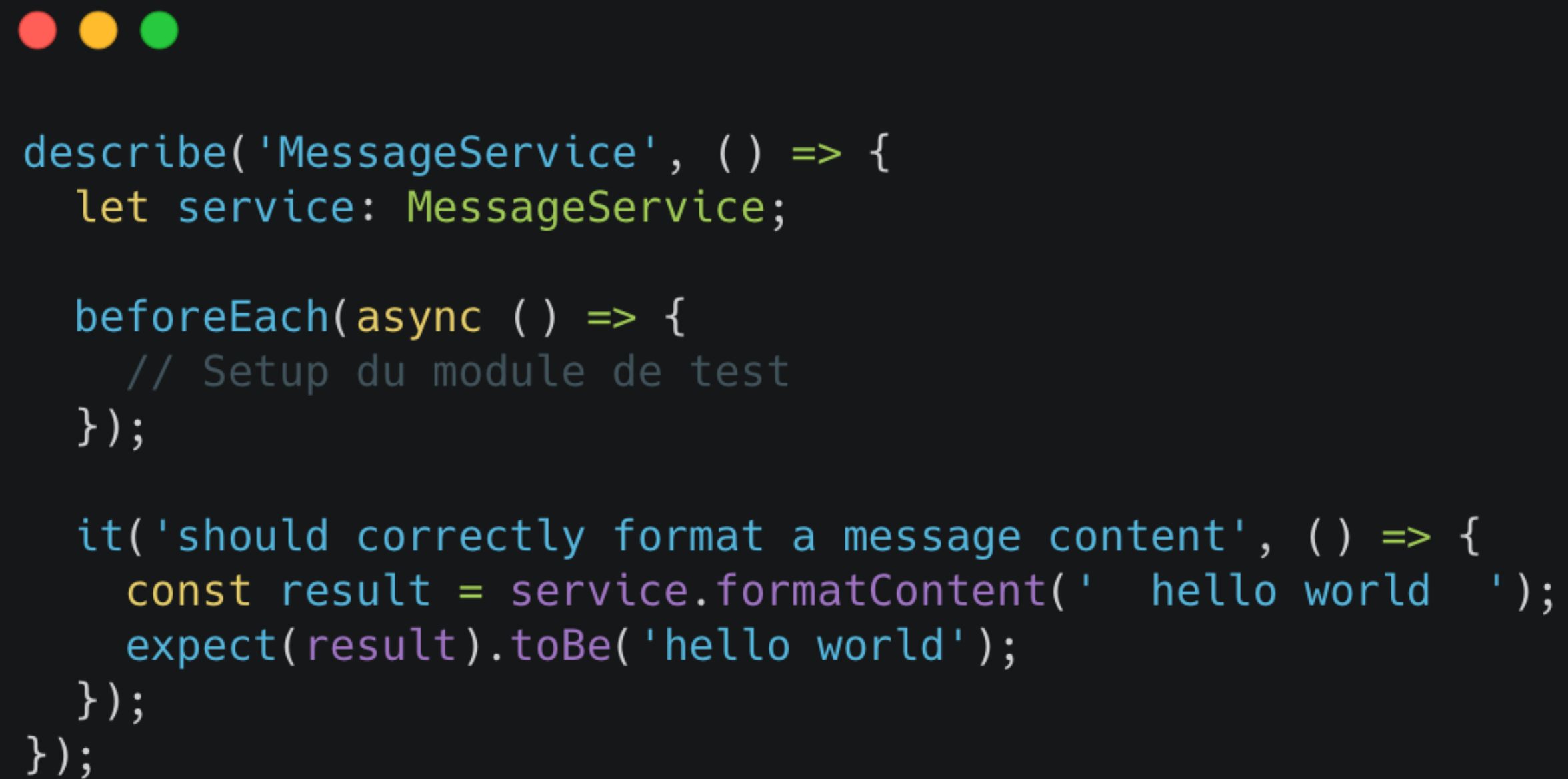
      - name: Build and push Docker image
        uses: docker/build-push-action@v4
        with:
          context: .
          push: true
          tags: your-dockerhub-username/chat-api:latest # Remplacez par votre repo

      # Les étapes de déploiement dépendent de votre hébergeur
      # Exemple pour Heroku :
      - name: Deploy to Heroku
        uses: akhileshns/heroku-deploy@v3.12.12
        with:
          heroku_api_key: ${ secrets.HEROKU_API_KEY }
          heroku_app_name: "your-heroku-app-name" # Remplacez
          heroku_email: "your-email@example.com"
```

Tests

Unitaire :

Quoi tester ? Des fonctions pures, des méthodes de service.



```
describe('MessageService', () => {  
  let service: MessageService;  
  
  beforeEach(async () => {  
    // Setup du module de test  
  });  
  
  it('should correctly format a message content', () => {  
    const result = service.formatContent('  hello world  ');  
    expect(result).toBe('hello world');  
  });  
});
```

Tests

Intégration:

Quoi tester ? Tester qu'une mutation `sendMessage` appelle bien le service `RabbitMQService`

Tests

End to end :

Postman : Créez une collection de requêtes pour tester votre API GraphQL. Vous pouvez écrire des scripts de test dans Postman pour valider les réponses.

Puppeteer : Simulez un utilisateur réel.

```
// Script Puppeteer (simplifié)
const browser = await puppeteer.launch();
const page = await browser.newPage();
await page.goto('http://localhost:3000'); // Votre client web
await page.type('#username', 'testuser');
await page.type('#message', 'Hello from Puppeteer!');
await page.click('#sendButton');
const sentMessage = await page.waitForSelector('.message-content');
expect(sentMessage).toContain('Hello from Puppeteer!');
await browser.close();
    expect(result).toBe('hello world');
  });
});
```

Tests de performances

Artillery :

Quoi tester ?

- Temps de réponse de la mutation `sendMessage.`,
- Latence de réception du message via la souscription.
- Utilisation CPU et mémoire du serveur API et de RabbitMQ.

```
config:
  target: "http://localhost:3000/graphql"
  phases:
    - duration: 60 # Durée du test en secondes
      arrivalRate: 20 # 20 nouveaux utilisateurs virtuels par seconde
  engines:
    graphql: {}
  scenarios:
    - name: "Send messages"
      engine: graphql
      flow:
        - query:
            mutation: |
              mutation SendMessage($content: String!) {
                sendMessage(conversationId: "some-conv-id", content: $content) {
                  id
                }
              }
            variables:
              content: "Hello from Artillery - {{ $loopCount }}"
```

Optimisations et Améliorations

Analysez les résultats des tests de performance pour trouver les goulots d'étranglement.

- **Problème N+1 dans GraphQL :**

- **Symptôme :** Si vous demandez une conversation avec 100 messages, et pour chaque message, son auteur, vous pourriez faire 101 requêtes à la base de données.
- **Solution :** Utilisez **DataLoader**. C'est un utilitaire qui regroupe les requêtes identiques qui se produisent dans un même "tick" de l'event loop en une seule requête (ex: `SELECT * FROM users WHERE id IN (1, 2, 3, ...)`).

- **Base de Données Lente :**

- **Solution :** Assurez-vous que vos colonnes utilisées pour les recherches (ex: `conversationId`) ont des **index**. Utilisez un outil d'analyse de requêtes (ex: `EXPLAIN ANALYZE` en PostgreSQL).

- **Scalabilité de RabbitMQ :**

- **Solution :** Si un seul nœud RabbitMQ ne suffit pas, vous pouvez créer un **cluster RabbitMQ** pour répartir la charge.

- **Mise en Cache :**

- **Solution :** Utilisez une solution comme **Redis** pour mettre en cache les données fréquemment accédées (profils utilisateur, conversations récentes) afin de réduire la charge sur la base de données principale.

Documentation

Un projet n'est complet que lorsqu'il est bien documenté.

- **README.md** : C'est la porte d'entrée de votre projet. Il doit contenir :
 - Une description claire du projet.
 - La liste des fonctionnalités clés.
 - Les technologies utilisées.
 - **Des instructions claires pour l'installation et le lancement :**
 1. git clone ...
 2. npm install
 3. cp .env.example .env (et expliquer les variables d'environnement)
 4. docker-compose up -d
 5. npm run start:dev
- **Documentation de l'API :**
 - GraphQL est auto-documenté grâce à l'introspection, mais vous pouvez utiliser des outils comme **Compodoc** pour générer une documentation web complète de votre code Nest.js.
 - Fournissez des exemples de requêtes pour les mutations et les souscriptions principales.