
✅ High-Level Architecture (Auth via Request/Response)

🧩 In User Service:

- Expose a **message pattern handler** (`@MessagePattern('verify_jwt')`) to **validate JWT tokens** .
- This handler receives a token, validates it, and returns:
  - { isValid: boolean, user?: any }
- No event queue is needed — this uses **request-response** via RabbitMQ.

🧩 In Publication Service:

- Use a **custom** `AuthGuard` to intercept incoming requests and extract the JWT token.
- Use `ClientProxy.send()` to call the `verify_jwt` pattern from the User Service via RabbitMQ.
- Wait for the User Service to respond with validation results.
- If valid, attach `user` to the request and continue.

- If invalid or timeout, throw `UnauthorizedException`.

🔐 Auth Integration – Publication Service Side

🧩 Goal

Enable authentication in the **Publication Service** by verifying JWT tokens via RabbitMQ through a custom `AuthGuard`.

📦 Publication Service

📢 - Add this folder structure:

> publication_ms/src/

> → **auth/**

> → `auth.guard.ts`
>
> → `auth.module.ts`
>
> → `user.decorator.ts`

✅ `auth.guard.ts`

Responsible for intercepting incoming requests and validating JWT tokens via RabbitMQ communication.

```
// publication-service/src/auth/auth.guard.ts
import { Injectable, CanActivate, ExecutionContext, UnauthorizedException } from
'@nestjs/common';
import { Inject } from '@nestjs/common';
import { ClientProxy } from '@nestjs/microservices';
import { firstValueFrom } from 'rxjs';
import { timeout, catchError } from 'rxjs/operators';
```

```
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(
    @Inject('USER_SERVICE') private readonly userServiceClient: ClientProxy
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);

    if (!token) {
      throw new UnauthorizedException('Token non fourni');
    }

    try {
      const response = await firstValueFrom(
        this.userServiceClient.send({ cmd: 'verify_jwt' }, token).pipe(
          timeout(15000),
          catchError((err) => {
            console.error('[PUBLICATION_SERVICE] ❌ Erreur communication
microservice USER:', err);
            throw new UnauthorizedException('Le microservice ne répond pas');
          }),
        )
      );

      if (!response.isValid) {
        throw new UnauthorizedException('Token invalide');
      }

      request.user = response.user;
      console.log('[PUBLICATION_SERVICE] ✅ Token validé, utilisateur :',
response.user);
      return true;
    } catch (error) {
      throw new UnauthorizedException(error.message || 'Erreur
d\'authentification');
    }
  }

  private extractTokenFromHeader(request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : undefined;
  }
}
```

✅ auth.module.ts

Registers the client used to communicate with the RabbitMQ queue responsible for JWT validation.

```
// publication-service/src/auth/auth.module.ts
import { Module } from '@nestjs/common';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { AuthGuard } from './auth.guard';

@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'USER_SERVICE',
        transport: Transport.RMQ,
        options: {
          urls: ['amqp://user:password@localhost:5672'],
          queue: 'auth_queue', // Queue that listens for JWT validation requests
          queueOptions: {
            durable: true
          },
        },
      },
    ]),
  ],
  providers: [AuthGuard],
  exports: [AuthGuard, ClientsModule], // Export for use in other
modules/controllers
})
export class AuthModule {}
```

✅ `user.decorator.ts`

Custom decorator to easily access the authenticated user inside your controller methods.

```
// publication-service/src/auth/user.decorator.ts
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const CurrentUser = createParamDecorator(
  (_data: unknown, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    return request.user;
  },
);
```

🧩 Usage

✅ In `PublicationModule` (or your relevant feature module)

Import the `AuthModule` to make the guard available across your services and controllers:

```
// publication-service/src/publication.module.ts (example)
import { Module } from '@nestjs/common';
import { AuthModule } from './auth/auth.module'; // 👈 Import here

@Module({
  imports: [
    AuthModule,
    // other modules...
  ],
  // providers/controllers...
})
export class PublicationModule {}
```

✅ In Any Controller

Protect endpoints using `@UseGuards(AuthGuard)` and access the user with `@CurrentUser()`.

```
// Example usage in a controller
import { Controller, Post, UseGuards, Body } from '@nestjs/common';
import { AuthGuard } from '../auth/auth.guard';
import { CurrentUser } from '../auth/user.decorator';
import { CreatePublicationDto } from './dto/create-publication.dto';

@Controller('publications')
export class PublicationsController {

  @Post()
  @UseGuards(AuthGuard) // 🔐 Protect this route
  async create(
    @Body() createPublicationDto: CreatePublicationDto,
    @CurrentUser() user, // 👤 Access authenticated user
```

```
  ) {
    console.log('Creating publication for user:', user);
    // Do something with createPublicationDto and user
  }
}
```

✅ Final Notes

- 🧠 The `AuthGuard` allows centralized, reusable, and secure JWT validation without relying on local JWT parsing.
- 📬 All JWT validation is offloaded to a central queue (`auth_queue`) via RabbitMQ, improving separation of concerns.
- ✅ This pattern supports microservice-scaled environments where authentication logic stays in one place.