

NESTJS RATE LIMITING SERIES



Framework: NestJS v11.0.5

- Packages:
@nestjs/throttler: ^6.4.0

Part 1

- WHY RATE LIMITING
- GLOBAL THROTTLING SETUP

Why Rate Limiting Matters

- **Protects your backend services**
Without rate limiting, a malicious user or buggy frontend could flood your API with requests and overwhelm your app or microservices.
- **Prevents abuse (e.g. brute-force attacks, scraping)**
Rate limiting blocks attackers trying to spam login forms or steal your data by scraping endpoints.
- **Ensures fair usage**
Prevents any single client from hogging your API resources.
- **Helps your system stay stable under load**
Good rate limiting makes your app more reliable even during peak traffic.

Why use @nestjs/throttler?

- Built and maintained by the NestJS team.
- Integrates cleanly with NestJS apps.
- Provides both global and per-route rate limiting.
- Works well with API gateways, microservices, and proxies.
- Supports custom strategies (e.g., rate limit by IP, user ID, or both).

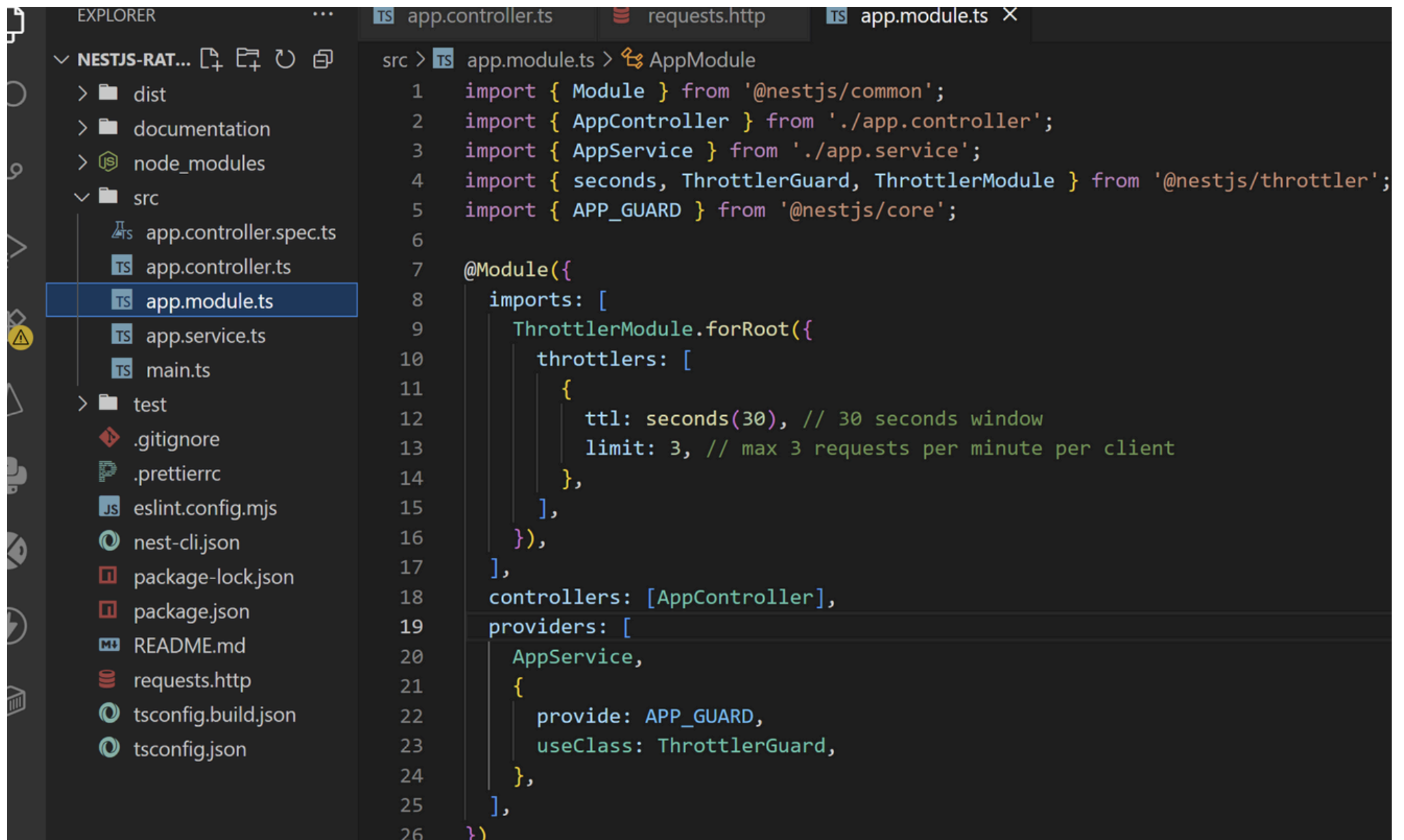
How to install

- `npm install @nestjs/throttler`

Next >> Global Rate Limiting

Global Rate Limiting

Add ThrottlerModule to AppModule



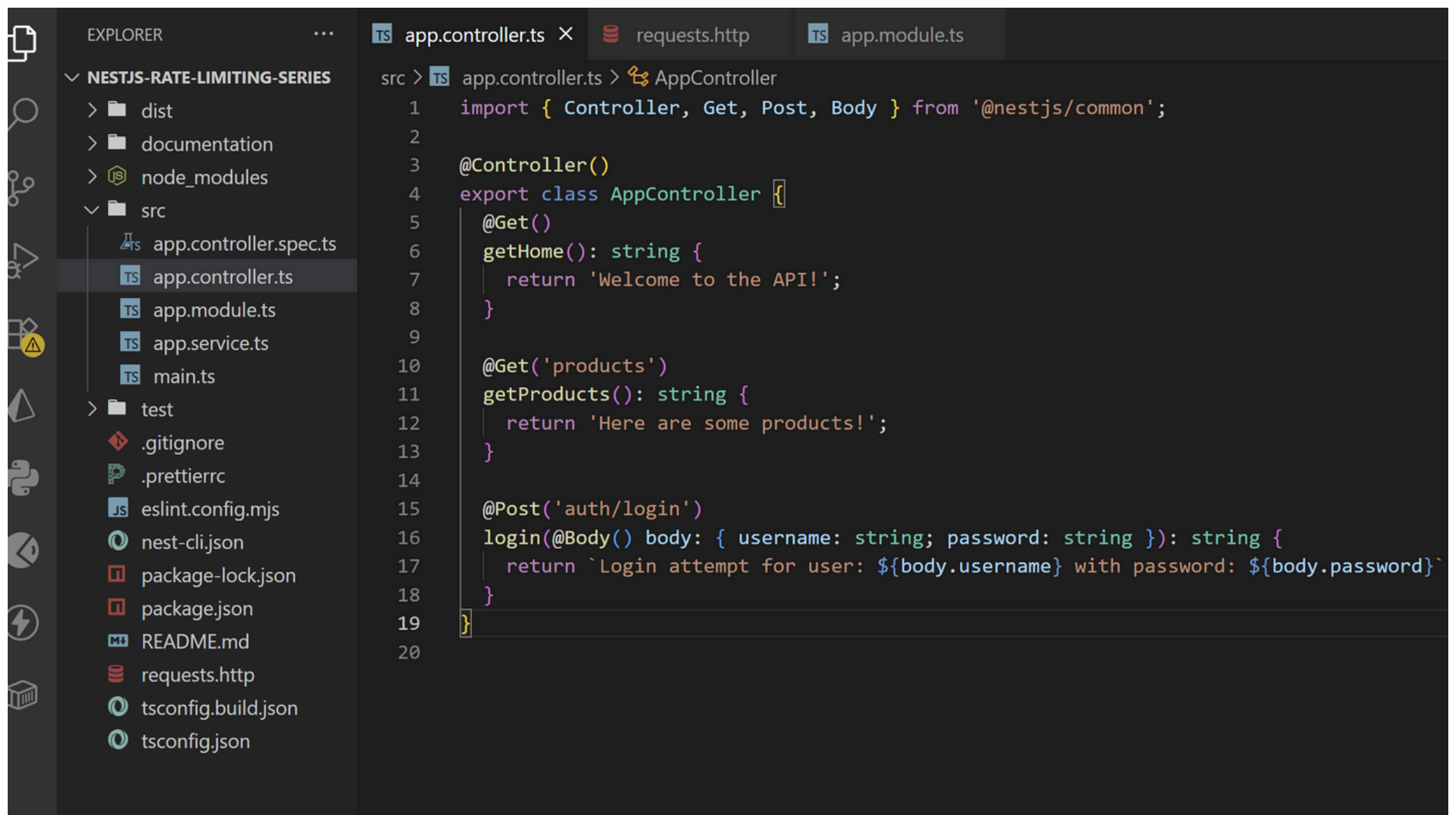
The screenshot shows a Visual Studio Code editor with the following components:

- EXPLORER (Left Panel):** Displays the file structure of a project named "NESTJS-RAT...". The "src" folder is expanded, showing files like `app.controller.spec.ts`, `app.controller.ts`, `app.module.ts` (which is selected), `app.service.ts`, and `main.ts`. Other files like `.gitignore`, `.prettierrc`, `eslint.config.mjs`, `nest-cli.json`, `package-lock.json`, `package.json`, `README.md`, `requests.http`, `tsconfig.build.json`, and `tsconfig.json` are also visible.
- Editor (Main Panel):** Shows the `app.module.ts` file. The code is as follows:

```
src > TS app.module.ts > AppModule
1  import { Module } from '@nestjs/common';
2  import { AppController } from './app.controller';
3  import { AppService } from './app.service';
4  import { seconds, ThrottlerGuard, ThrottlerModule } from '@nestjs/throttler';
5  import { APP_GUARD } from '@nestjs/core';
6
7  @Module({
8    imports: [
9      ThrottlerModule.forRoot({
10        throttlers: [
11          {
12            ttl: seconds(30), // 30 seconds window
13            limit: 3, // max 3 requests per minute per client
14          },
15        ],
16      }),
17    ],
18    controllers: [AppController],
19    providers: [
20      AppService,
21      {
22        provide: APP_GUARD,
23        useClass: ThrottlerGuard,
24      },
25    ],
26  })
```

Controller

Set up Controller



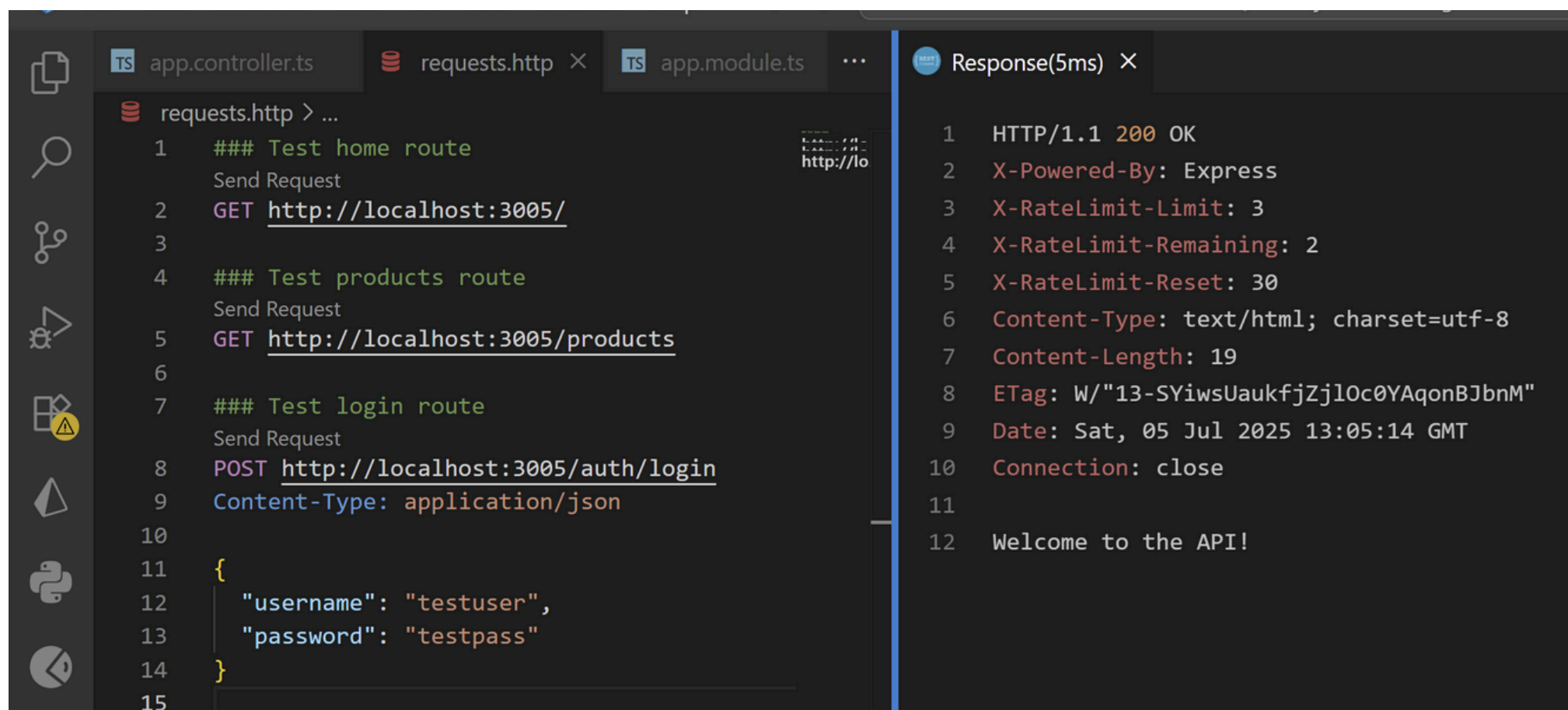
The screenshot shows the Visual Studio Code editor interface. On the left, the Explorer sidebar displays the project structure for 'NESTJS-RATE-LIMITING-SERIES'. The 'src' directory is expanded, showing files like 'app.controller.spec.ts', 'app.controller.ts', 'app.module.ts', 'app.service.ts', and 'main.ts'. The 'app.controller.ts' file is selected and open in the editor. The editor window has three tabs: 'app.controller.ts', 'requests.http', and 'app.module.ts'. The 'app.controller.ts' tab is active, showing the following TypeScript code:

```
src > TS app.controller.ts > AppController
1  import { Controller, Get, Post, Body } from '@nestjs/common';
2
3  @Controller()
4  export class AppController {
5    @Get()
6    getHome(): string {
7      return 'Welcome to the API!';
8    }
9
10   @Get('products')
11   getProducts(): string {
12     return 'Here are some products!';
13   }
14
15   @Post('auth/login')
16   login(@Body() body: { username: string; password: string }): string {
17     return `Login attempt for user: ${body.username} with password: ${body.password}`;
18   }
19 }
20
```

◆ Testing the Endpoints with requests.http

Initial test: successful request

We'll use a request.http file to send test requests directly from our editor



The screenshot shows a VS Code editor with three tabs: 'app.controller.ts', 'requests.http', and 'app.module.ts'. The 'requests.http' tab is active, displaying a file with test requests. The first request is a GET to 'http://localhost:3005/'. The second is a GET to 'http://localhost:3005/products'. The third is a POST to 'http://localhost:3005/auth/login' with a JSON body containing 'username: testuser' and 'password: testpass'. To the right, a 'Response(5ms)' panel shows the output of the first request: 'HTTP/1.1 200 OK', headers for 'X-Powered-By', 'X-RateLimit-Limit', 'X-RateLimit-Remaining', 'X-RateLimit-Reset', 'Content-Type', 'Content-Length', 'ETag', and 'Date', and a body message 'Welcome to the API!'.

```
requests.http > ...
1  ### Test home route
   Send Request
2  GET http://localhost:3005/
3
4  ### Test products route
   Send Request
5  GET http://localhost:3005/products
6
7  ### Test login route
   Send Request
8  POST http://localhost:3005/auth/login
9  Content-Type: application/json
10
11  {
12    "username": "testuser",
13    "password": "testpass"
14  }
15
```

```
1  HTTP/1.1 200 OK
2  X-Powered-By: Express
3  X-RateLimit-Limit: 3
4  X-RateLimit-Remaining: 2
5  X-RateLimit-Reset: 30
6  Content-Type: text/html; charset=utf-8
7  Content-Length: 19
8  ETag: W/"13-SYiwsUaukfjZjl0c0YAqonBJbnM"
9  Date: Sat, 05 Jul 2025 13:05:14 GMT
10 Connection: close
11
12 Welcome to the API!
```

The server responded with:

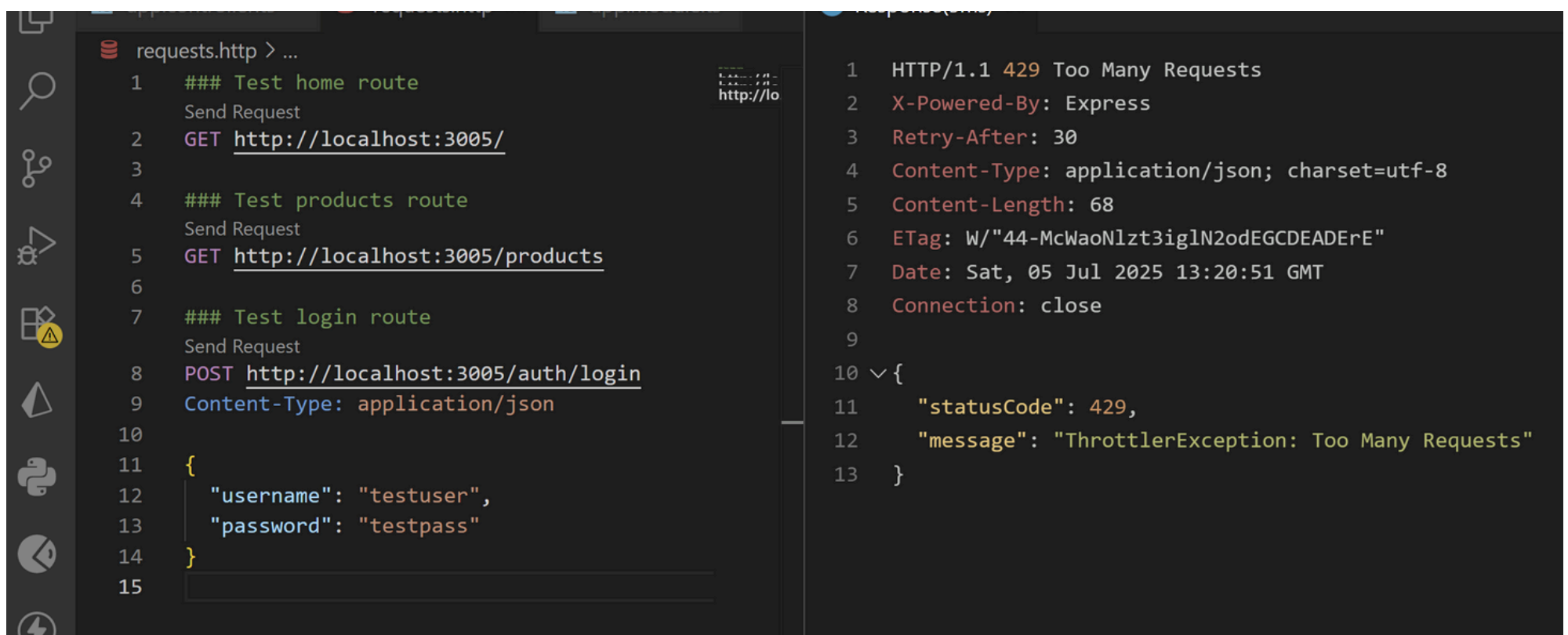
- 200 OK status — the request succeeded.
- X-RateLimit-Limit: 3 — shows our configured limit (3 requests per 30 seconds).
- X-RateLimit-Remaining: 2 — we've used 1 out of 3 allowed requests.
- X-RateLimit-Reset: 30 — time (in seconds) until the rate limit window resets.

◆ Testing the Endpoints with requests.http

Sending multiple requests

The first three requests returned 200 OK.

✗ The fourth request triggered our rate limiter, and we received a 429 Too Many Requests response.



```
requests.http > ...
1  ### Test home route
   Send Request
2  GET http://localhost:3005/
3
4  ### Test products route
   Send Request
5  GET http://localhost:3005/products
6
7  ### Test login route
   Send Request
8  POST http://localhost:3005/auth/login
9  Content-Type: application/json
10
11  {
12    "username": "testuser",
13    "password": "testpass"
14  }
15

1  HTTP/1.1 429 Too Many Requests
2  X-Powered-By: Express
3  Retry-After: 30
4  Content-Type: application/json; charset=utf-8
5  Content-Length: 68
6  ETag: W/"44-McWaoNlzt3iglN2odEGCDEADeR"
7  Date: Sat, 05 Jul 2025 13:20:51 GMT
8  Connection: close
9
10  {
11    "statusCode": 429,
12    "message": "ThrottlerException: Too Many Requests"
13  }
```

- request.http is a simple and effective way to test your API directly from your code editor.
- The X-RateLimit-* headers help you monitor rate limit status and remaining allowance.
- Our global rate limit is working: after 3 requests in 30 seconds, further requests are blocked as expected.

What our current setup does

- We've applied global rate limiting at the app level using `ThrottlerModule.forRoot()`.
- Each route is rate-limited individually:

If a client hits the limit on one endpoint, they can still access other endpoints without restriction during the same window.

- Example: Exceeding the limit on `/auth/login` doesn't block access to `/products`.

What's the challenge?

- ➡ Sometimes you want rate limits to apply across all routes for a client, not just per route.
- ➡ This helps stop attackers from spamming different endpoints to bypass limits

What's next in Part 2

- We'll explore how to override limits per route using `@Throttle()` and `@SkipThrottle()`.
- We'll also cover how to create a custom throttler that applies limits globally across all endpoints for a client!