**MỤC LỤ****C**

[Chương 1. Phần mở đầu 6](#_Toc194247610)

[1.1. Mục tiêu đề tài 6](#_Toc194247611)

[1.2. Phương pháp nghiên cứu 6](#_Toc194247612)

[1.3. Bố cục báo cáo 6](#_Toc194247613)

[Chương 2. API Gateway 7](#_Toc194247614)

[2.1. Lịch sử hình thành và phát triển 7](#_Toc194247615)

[2.1.1. Thời kỳ Monolithic Applications 7](#_Toc194247616)

[2.1.2. Ưu điểm của Monolithic Applications 7](#_Toc194247617)

[2.1.3. Nhược điểm của Monolithic Applications 8](#_Toc194247618)

[2.1.4. Sự Phát Triển của Monolithic Applications 8](#_Toc194247619)

[2.1.5. Sự Thoái Trào của Monolithic Applications 8](#_Toc194247620)

[2.2. Thời kỳ Microservices 9](#_Toc194247621)

[2.2.1. Ưu điểm của Microservices 10](#_Toc194247622)

[2.2.2. Nhược điểm của Microservices 10](#_Toc194247623)

[2.3. Sự hình thành của API Gateway 10](#_Toc194247624)

[2.3.1. Bối cảnh ra đời 10](#_Toc194247625)

[2.3.2. Vai trò và chức năng của API Gateway 11](#_Toc194247626)

[2.3.3. Các API Gateway phổ biến 12](#_Toc194247627)

[Chương 3. Ocelot 15](#_Toc194247628)

[3.1. Ocelot là gì 15](#_Toc194247629)

[3.2. Ưu điểm của Ocelot 15](#_Toc194247630)

[3.3. Nhược điểm 15](#_Toc194247631)

[Chương 4. Triển khai API Gateway bằng OCELOT 16](#_Toc194247632)

[4.1. Cài đặt Ocelot 16](#_Toc194247633)

[4.2. Rate Limiting 20](#_Toc194247634)

[4.3. Custom Rate Limiting 21](#_Toc194247635)

[4.4. Quality of Service 22](#_Toc194247636)

[4.5. SwaggerForOcelot 27](#_Toc194247637)

DANH MỤC TỪ VIẾT TẮT

JWT: JSON Web Token

API: Application Programming Interface

UI: User Interface

DANH MỤC HÌNH

[Hình 2.1: Mô hình MonoLithic 7](#_Toc194247643)

[Hình 2.2: Kiến trúc Microservices 9](#_Toc194247644)

[Hình 2.3: : Api Gateway trong mô hình Microservices 11](#_Toc194247645)

[Hình 4.1: : Cài đặt Package Ocelot cho project 16](#_Toc194247646)

[Hình 4.2: : Khởi tạo các lớp services 17](#_Toc194247647)

[Hình 4.3: Đổi port thành 5001 cho project Student 17](#_Toc194247648)

[Hình 4.4: Các thuộc tính để thực hiện định tuyến API 18](#_Toc194247649)

[Hình 4.5: Thiết lập file config tại program.cs 19](#_Toc194247650)

[Hình 4.6: Kết quả sau khi gọi API 19](#_Toc194247651)

[Hình 4.7: Cấu hình Rate limiting cho service Car 20](#_Toc194247652)

[Hình 4.8: Lỗi khi gọi quá nhiều lần tới API Gateway 21](#_Toc194247653)

[Hình 4.9: Cấu hình custom Rate limiting 21](#_Toc194247654)

[Hình 4.10: Custom Rate limiting 22](#_Toc194247655)

[Hình 4.11: Cài đặt Polly trong nuget 23](#_Toc194247656)

[Hình 4.12: Khai báo Polly trong file program.cs 23](#_Toc194247657)

[Hình 4.13: Thuộc tính QoSOptions 24](#_Toc194247658)

[Hình 4.14: Điều chỉnh service car 25](#_Toc194247659)

[Hình 4.15: Lỗi khi gọi tới honda 26](#_Toc194247660)

[Hình 4.16: Lỗi khi gọi tới yamaha 26](#_Toc194247661)

[Hình 4.17: Cài đặt MMLib.SwaggerForOcelot 27](#_Toc194247662)

[Hình 4.18: Khai báo SwaggerKey 28](#_Toc194247663)

[Hình 4.19: Khai báo SwaggerEndpoint 29](#_Toc194247664)

[Hình 4.20: Add service trong program.cs 29](#_Toc194247665)

[Hình 4.21: Kết quả swagger Ocelot 30](#_Toc194247666)

Danh Mục Bảng

[Bảng 2.1 Các Api Gateway phổ biến hiện nay 14](#_Toc194247638)

[Bảng 4.1: Bảng thuộc tính cơ bản trong Ocelot 18](#_Toc194247639)

[Bảng 4.2: Bảng ý nghĩa thuộc tính Rate Limiting 20](#_Toc194247640)

[Bảng 4.3: Thông số custom lỗi Rate limiting 22](#_Toc194247641)

[Bảng 4.4: Bảng thuộc tính QoSOptions 24](#_Toc194247642)

# Phần mở đầu

## Mục tiêu đề tài

Trong thời đại của kiến trúc microservices, việc quản lý luồng dữ liệu giữa các dịch vụ trở thành một thách thức quan trọng cho các lập trình viên. Các hệ thống cần một cách tiếp cận hiệu quả để điều phối request từ client đến các dịch vụ phù hợp, đồng thời cũng phải đảm bảo hiệu suất, bảo mật và khả năng mở rộng. API Gateway xuất hiện như một giải pháp mạnh mẽ để giải quyết vấn đề này.

Ocelot là một API Gateway mã nguồn mở dành cho nền tảng .NET, giúp đơn giản hóa việc quản lý request giữa các microservices. Với khả năng routing linh hoạt, hỗ trợ xác thực, cân bằng tải, caching, và tích hợp dễ dàng với Docker và Kubernetes, Ocelot trở thành lựa chọn phổ biến cho các hệ thống microservices trong hệ sinh thái .NET.

## Phương pháp nghiên cứu

**Phương pháp nghiên cứu tài liệu,** tham khảo các báo cáo, bài báo, sách vở,... về các công nghệ và ứng dụng của chúng trong lĩnh vực triển khai phần mềm.

## Bố cục báo cáo

Chương 1. Phần mở đầu: Nói về mục tiêu thực hiện bài báo cáo, phương pháp thực hiện báo cáo và bố cục báo cáo.

Chương 2. Giới thiệu về API Gateway: Cung cấp những thông tin về công nghệ, lịch sử phát triển, và sự ra đời của API Gateway

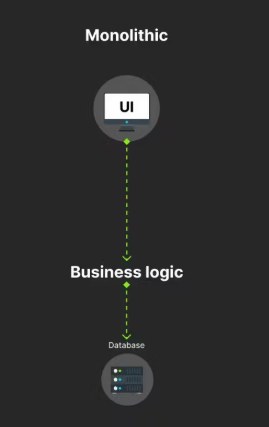
Chương 3. Ocelot: Giới thiệu về Ocelot, những tính năng quan trọng, ưu điểm và nhược điểm khi sử dụng Ocelot.

Chương 4. Triển khai API Gateway bằng Ocelot: Cài đặt và sử dụng Ocelot để định tuyến các API.

# API Gateway

## Lịch sử hình thành và phát triển

### Thời kỳ Monolithic Applications

****Monolithic Applications (Ứng dụng nguyên khối) là mô hình kiến trúc phần mềm phổ biến trước khi các kiến trúc Microservices ra đời. Trong mô hình này toàn bộ ứng dụng được xây dựng như một khối duy nhất, bao gồm cả phần giao diện người dùng (UI), logic xử lý nghiệp vụ và cơ sở dữ liệu [1].

Hình 2.1: Mô hình MonoLithic

### Ưu điểm của Monolithic Applications

Do không phải chia nhỏ hệ thống, các bộ phận phát triển có thể xây dựng và triển khai ứng dụng nhanh chóng, dễ dàng hơn [2]. Khi triển khai lên máy chủ, chỉ cần một tệp tin (JAR, WAR, EXE, v.v.) hoặc một ứng dụng duy nhất, giúp đơn giản hóa quá trình quản lý.

Toàn bộ ứng dụng nằm trong cùng một mã nguồn, giúp việc debug, logging và testing trở nên thuận tiện hơn. Ngoài ra, không cần điều phối nhiều dịch vụ trên các container hoặc máy chủ khác nhau, giúp quản lý tài nguyên đơn giản và hiệu quả hơn.

### ****Nhược điểm của Monolithic Applications****

Khi ứng dụng phát triển lớn hơn, việc thêm tính năng mới hoặc sửa đổi một phần nhỏ có thể ảnh hưởng đến toàn bộ hệ thống dẫn đến việc khó bảo trì khi mở rộng hệ thống. Một thay đổi nhỏ có thể yêu cầu xây dựng và triển khai lại toàn bộ ứng dụng, gây mất thời gian và có rủi ro hệ thống gặp sự cố trong quá trình khởi động lại [2].

Nếu có nhiều nhóm làm việc và phát triển trên cùng một mã nguồn lớn, xung đột mã (merge conflicts) là việc khó tránh khỏi và khó khăn trong quản lý phiên bản git sẽ xảy ra. Khi công nghệ thay đổi, rất khó để nâng cấp hoặc thay thế và áp dụng vào một phần của hệ thống mà không làm gián đoạn toàn bộ ứng dụng [3].

### Sự Phát Triển của Monolithic Applications

Trước năm 2010, hầu hết các hệ thống phần mềm doanh nghiệp, thương mại điện tử, và ứng dụng web đều được xây dựng theo mô hình Monolithic. Điều này xuất phát từ sự phổ biến của các nền tảng phát triển phần mềm như Java EE, ASP.NET, Ruby on Rails, PHP và các framework khác, vốn hỗ trợ mạnh mẽ cho kiến trúc nguyên khối.

Vào thời điểm đó, hạ tầng máy chủ đa phần dựa trên mô hình On-Premise (máy chủ vật lý đặt tại doanh nghiệp), và các hệ thống phần mềm thường được triển khai trên một hoặc một vài máy chủ lớn và được quản lý một cách thủ công. Vì vậy, một kiến trúc phần mềm gói gọn tất cả chức năng trong một ứng dụng duy nhất giúp đơn giản hóa việc triển khai, bảo trì và vận hành.

Tuy nhiên, khi quy mô ứng dụng tăng lên theo thời gian, kiến trúc Monolithic bắt đầu bộc lộ những yếu điểm. Việc thay đổi một phần nhỏ của hệ thống có thể yêu cầu phải xây dựng và triển khai lại toàn bộ ứng dụng. Điều này gây ra độ trễ cao, rủi ro lỗi hệ thống, và làm chậm tiến trình phát triển phần mềm, đặc biệt là đối với các doanh nghiệp công nghệ có nhu cầu cập nhật liên tục.

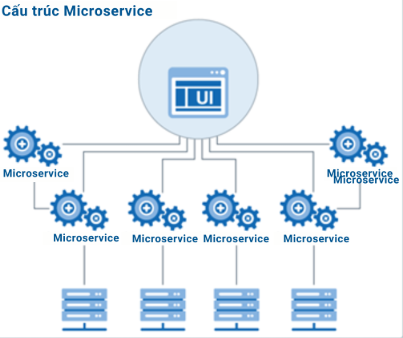
### Sự Thoái Trào của Monolithic Applications

Từ năm 2010 trở đi, với sự phát triển mạnh mẽ của điện toán đám mây và nhu cầu mở rộng linh hoạt, các doanh nghiệp công nghệ bắt đầu tìm kiếm các mô hình kiến trúc mới nhằm giải quyết những hạn chế của Monolithic Applications. Đây cũng là thời kỳ mà các công nghệ Containerization như Docker, Kubernetes dần trở nên phổ biến và đặc biệt là Microservices.

Các nền tảng như Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure giúp các doanh nghiệp dễ dàng mở rộng hệ thống mà không cần đầu tư hạ tầng vật lý.

Các phương pháp như CI/CD (Continuous Integration/Continuous Deployment) yêu cầu hệ thống có thể triển khai nhanh chóng, điều mà kiến trúc Microservices hỗ trợ tốt hơn. Bên cạnh đó, Công nghệ Containerization (Docker, Kubernetes) giúp cô lập và quản lý các dịch vụ một cách linh hoạt, mở đường cho việc triển khai Microservices một cách hiệu quả.

## Thời kỳ Microservices

Với kiến ​​trúc microservice, một ứng dụng được xây dựng dưới dạng các thành phần độc lập chạy từng quy trình ứng dụng dưới dạng các service riêng lẻ và độc lập. Các dịch vụ này giao tiếp với người dùng thông qua giao diện được xác định rõ bằng cách sử dụng API. Các dịch vụ được xây dựng cho các khả năng kinh doanh và mỗi dịch vụ thực hiện một chức năng duy nhất. Bởi vì chúng được chạy độc lập, mỗi dịch vụ có thể được cập nhật, triển khai và mở rộng để đáp ứng nhu cầu cho các chức năng cụ thể của phần mềm.

Hình 2.2: Kiến trúc Microservices

### Ưu điểm của Microservices

Mỗi dịch vụ có thể được mở rộng độc lập dựa trên nhu cầu của khách hàng hoặc nhu cầu thực tế vì vậy không cần phải mở rộng toàn bộ hệ thống, giúp giảm thời gian triển khai và hạn chế rủi ro khi nâng cấp hệ thống. Các services được chia nhỏ thành các dịch vụ, mỗi nhóm phát triển có thể làm việc độc lập trên mỗi dịch vụ mà không ảnh hưởng đến nhóm khác đặc biệt trong các dự án lớn với nhiều đội ngũ kỹ thuật [4].

Mỗi dịch vụ có thể sử dụng công nghệ, framework hoặc ngôn ngữ lập trình phù hợp với chức năng của nó giúp cho hệ thống có thể tận dụng được công nghệ mới. Nếu một dịch vụ bị lỗi, các dịch vụ khác vẫn có thể hoạt động bình thường, giảm thiểu tác động lên toàn bộ hệ thống. Ngoài ra, Microservices kết hợp rất tốt với Docker, Kubernetes, giúp triển khai linh hoạt trên các nền tảng đám mây như AWS, Google Cloud, Azure.

### Nhược điểm của Microservices

Việc phải quản lý nhiều dịch vụ nhỏ đòi hỏi hệ thống phải có cơ chế điều phối phức tạp hơn, có thể làm tăng độ trễ do các dịch vụ giao tiếp với nhau qua API hoặc message queue thay vì gọi trực tiếp trong cùng một ứng dụng như Monolithic.

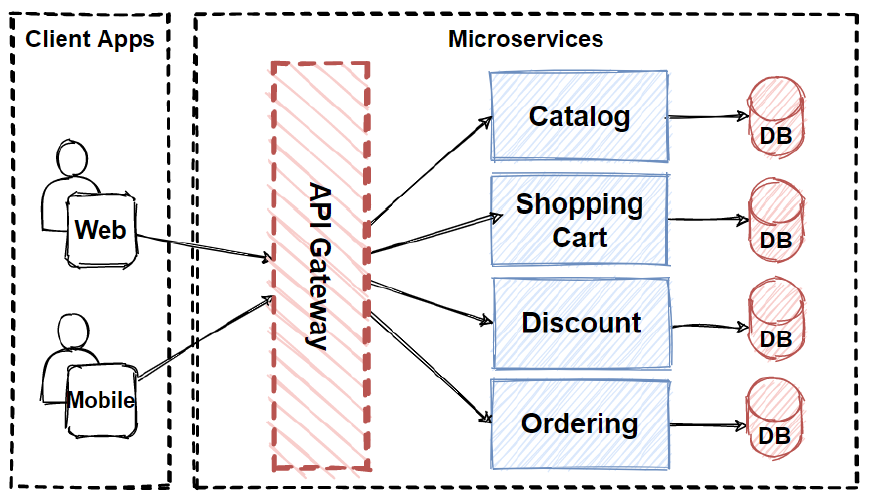
Mỗi dịch vụ có thể sử dụng cơ sở dữ liệu riêng, gây khó khăn trong việc duy trì tính nhất quán dữ liệu khi hoạt động trong môi trường database phân tán. Do kiến trúc Microservices đòi hỏi nhiều công nghệ phức tạp, các doanh nghiệp cần đội ngũ kỹ thuật có kinh nghiệm trong việc quản lý và thực thi Docker, Kubernetes, DevOps, API Management, Logging, Security [4].

## Sự hình thành của API Gateway

### Bối cảnh ra đời

Khi kiến trúc Microservices trở nên phổ biến, các hệ thống gặp nhiều vấn đề liên quan đến quản lý giao tiếp giữa các services, Client gửi nhiều request đến các dịch vụ khác nhau, làm tăng độ phức tạp trong quản lý kết nối và đẩy số lượng request tăng lên đáng kể.

Họ nhận thấy rằng cần có một cách tiếp cận mới để quản lý các request giữa client và hàng trăm dịch vụ nhỏ. Để giải quyết những thách thức này, API Gateway ra đời như một lớp trung gian giữa client và các services backend, giúp điều phối, tối ưu hiệu suất và quản lý hệ thống hiệu quả hơn.



Hình 2.3: : Api Gateway trong mô hình Microservices

### Vai trò và chức năng của API Gateway

#### Định tuyến request (Routing)

* API Gateway đóng vai trò như một người trung gian, nhận request từ client và quyết định chuyển tiếp request đến services thích hợp.
* Hỗ trợ định tuyến động (Dynamic Routing) dựa trên đường dẫn URL, header, query parameter hoặc nội dung của request [5].

1. **Load Balancing**

* Khi một service backend có nhiều instance chạy đồng thời để phục vụ request, API Gateway sẽ giúp phân phối tải đều giữa các instance, tránh tình trạng quá tải cho các instance đó.
* API Gateway hỗ trợ nhiều thuật toán cân bằng tải như: Round Robin, Least Connections, Weighted Load Balancing.

1. **Bảo mật API (Security)**

* xác thực danh tính của client trước khi cho phép truy cập vào hệ thống
* các phương thức xác thực phổ biến: OAuth2, JWT, API Key

1. **Rate Limiting**

* API Gateway cung cấp tính năng rate limiting nhằm mục đích giới hạn số lượng request mà một client có thể gửi đến services trong một khoảng thời gian nhất định để tránh quá tải.
* IP Whitelisting: Cho các địa chỉ IP cụ thể không bị ảnh hưởng từ rate limiting

Ngoài ra, còn các tính năng thú vị khác như tối ưu hoá hiệu suất của hệ thống thông qua việc Caching để lưu trữ phản hồi từ backend để giảm tải truy vấn lặp lại và cải thiện tốc độ phản hồi. Monitoring và Logging với mục đích thu thập dữ liệu request, như số lượng request, thời gian phản hồi, tỷ lệ lỗi.

### Các API Gateway phổ biến

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| API Gateway | Môi Trường | Ngôn ngữ | Ưu điểm | Nhược điểm |
| Ocelot | .NET, Clound | C# | - Tích hợp tốt với hệ sinh thái .NET - Cấu hình đơn giản với JSON hoặc code C# - Hỗ trợ đầy đủ tính năng: Routing, Load Balancing, Authentication, Caching, Service Discovery | - Chỉ dành cho .NET Core, không phù hợp với hệ thống đa nền tảng - Hiệu suất kém hơn Kong, NGINX khi xử lý lưu lượng lớn |
| Kong | Cloud, Kubernetes | Lua | - Hiệu suất cao, nhẹ, có thể mở rộng với plugin - Hỗ trợ API Gateway + Service Mesh - Hỗ trợ cả HTTP và gRPC - Hỗ trợ authentication, rate limiting, monitoring, logging mạnh mẽ | - Cấu hình phức tạp hơn Ocelot - Cần database (PostgreSQL, Cassandra) để quản lý metadata |
| Traefik | Kubernetes, Docker | Go | - Tích hợp tốt với Docker, Kubernetes, Consul - Tự động nhận diện service mới mà không cần cấu hình thủ công - Hỗ trợ TLS, Load Balancing, Middleware mạnh mẽ | - Không hỗ trợ đầy đủ authentication như Kong - Hạn chế về tính năng so với API Gateway khác |
| NGINX | Cloud | C | - Hiệu suất cao, có thể xử lý hàng triệu request/giây - Hỗ trợ caching, load balancing mạnh - Có thể mở rộng thành API Gateway với NGINX Plus | - Cấu hình phức tạp hơn so với các API Gateway khác - Phiên bản miễn phí có hạn chế về tính năng API Gateway |
| AWS API Gateway | Cloud (AWS) |  | |  | | --- | |  |  |  | | --- | | - Dịch vụ serverless , dễ dàng tích hợp với AWS Lambda, DynamoDB - Tự động mở rộng theo nhu cầu - Hỗ trợ monitoring bằng AWS CloudWatch | | - Chi phí cao nếu có nhiều request - Phụ thuộc vào AWS, không thể triển khai on-premise |
| Azure API Management | Cloud (Azure) |  | - Dịch vụ quản lý API toàn diện trên Azure - Hỗ trợ caching, security, rate limiting mạnh - Tích hợp với Azure Functions và Logic Apps | - Giá thành cao, khó triển khai on-premise - Cần hiểu rõ Azure ecosystem để tối ưu |
| Apigee (Google Cloud API Gateway) | Cloud (Google Cloud) |  | |  | | --- | |  |   - Dịch vụ mạnh mẽ, hỗ trợ đầy đủ tính năng API Gateway - Hỗ trợ quản lý API Lifecycle, security, monitoring | - Chi phí cao, phức tạp khi triển khai on-premise |
| Netflix Zuul | Cloud | Java | |  | | --- | |  |   - Được thiết kế cho hệ thống Microservices lớn - Hỗ trợ routing, load balancing, security - Tích hợp tốt với hệ sinh thái Spring Boot | - Hiệu suất kém hơn Kong, NGINX - Netflix đã dừng phát triển Zuul 1, thay thế bằng Zuul 2 và Spring Cloud Gateway |

Bảng 2.1 Các Api Gateway phổ biến hiện nay

# Ocelot

## Ocelot là gì

Ocelot là một API Gateway mã nguồn mở dành cho trong hệ sinh thái .NET, được thiết kế để hoạt động trong các hệ thống Microservices. Được phát triển trên nền tảng .NET Core, Ocelot tối ưu cho các ứng dụng ASP.NET Core vì được viết bằng C# và dễ dàng cấu hình để kết nối tới các services [6].

Ocelot cung cấp và hỗ trợ đầy đủ các tính năng của một API Gateway hiện đại như: Routing & Load Balancing, Authentication,Rate Limiting, Caching, Service Discovery, Logging & Monitoring.

## Ưu điểm của Ocelot

Ocelot giúp che dấu toàn bộ đường dẫn liên kết tới services bên trong của ứng dụng và các đường dẫn này không cần phải để lộ ra bên ngoài, nâng cao bảo mật và giảm sự phụ thuộc giữa client và ứng dụng, vì thế các Client giờ đây chỉ cần giao tiếp với gateway và gateway sẽ hướng dẫn cho client định tuyến đến services.

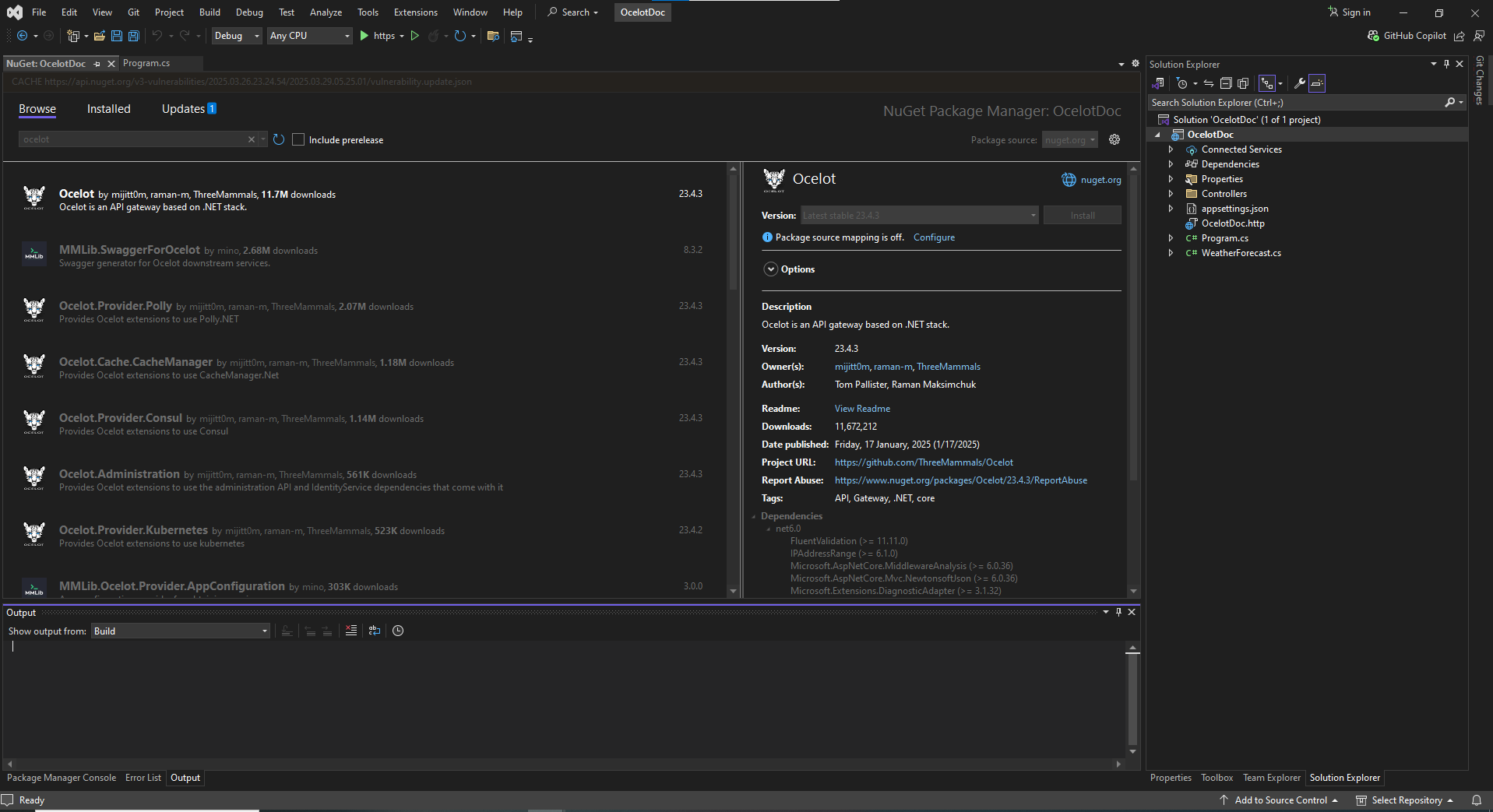
## Nhược điểm

API Gateway đóng vai trò như một lớp trung gian giữa client và hệ thống Microservices, giúp điều phối request và cung cấp các tính năng bảo mật, caching, logging. Tuy nhiên, nếu không được cấu hình đúng cách, nó có thể trở thành điểm nghẽn cổ chai**,** ảnh hưởng đến hiệu suất toàn hệ thống. Một API Gateway được thiết lập kém có thể gây lỗi trong quá trình routing, bảo mật hoặc caching, khiến các dịch vụ backend không thể hoạt động trơn tru. Đặc biệt, khi số lượng request tăng đột biến, nếu API Gateway không được tối ưu, nó có thể dẫn đến độ trễ cao, ảnh hưởng đến trải nghiệm người dùng.

Ngoài ra, vì tất cả request đều phải đi qua API Gateway trước khi đến backend, hệ thống có thể gặp rủi ro gián đoạn nếu API Gateway gặp sự cố và ngưng hoạt động. Do đó, việc tối ưu cấu hình và đảm bảo tính sẵn sàng cho API Gateway là rất quan trọng trong hệ thống Microservices.

# Triển khai API Gateway bằng OCELOT

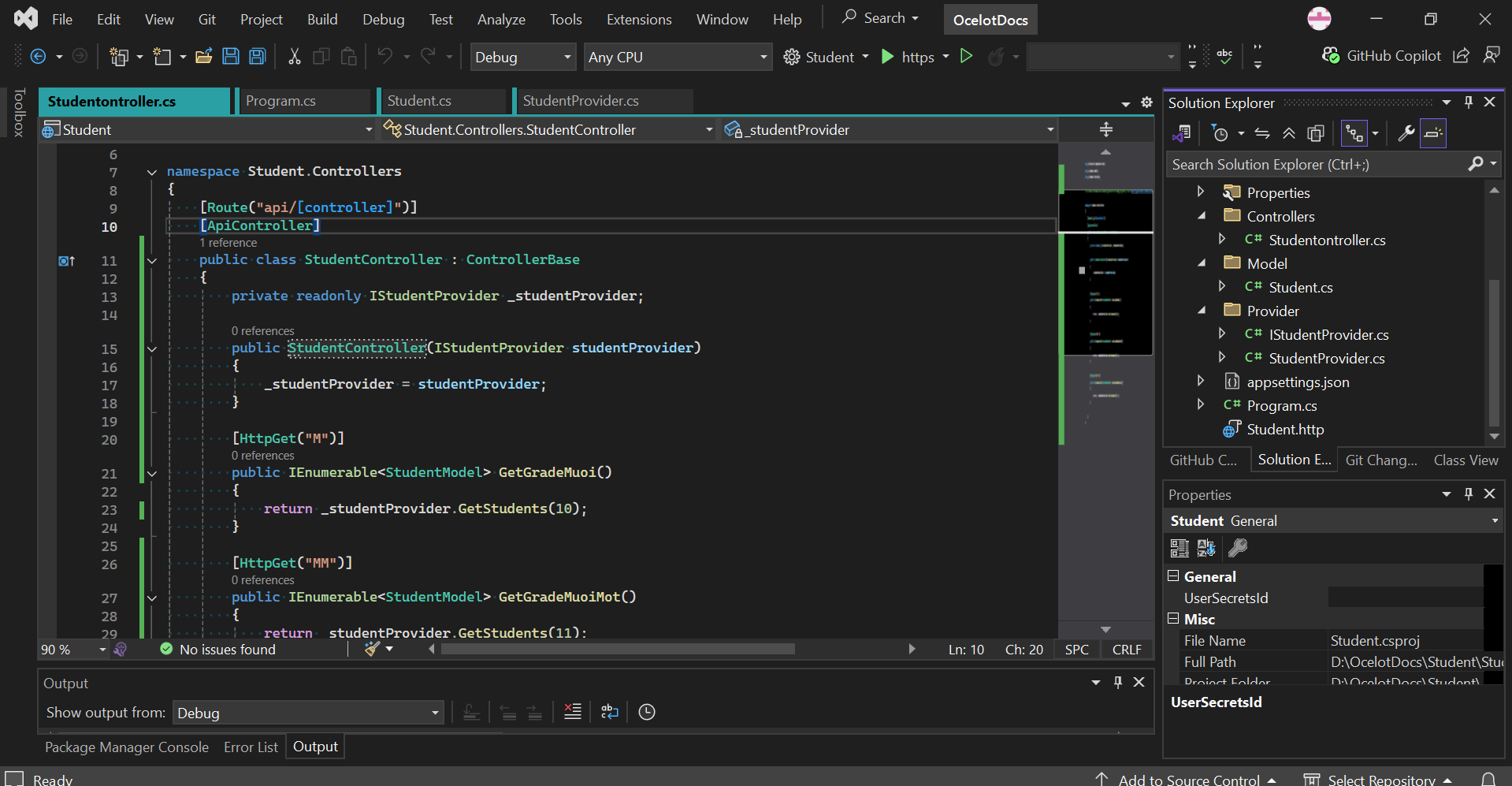
## Cài đặt Ocelot

Đầu tiên ta cần tạo ra project là Ocelot và cài đặt package Ocelot để cung cấp các chức năng cũng như cấu hình của ocelot cho project [7].

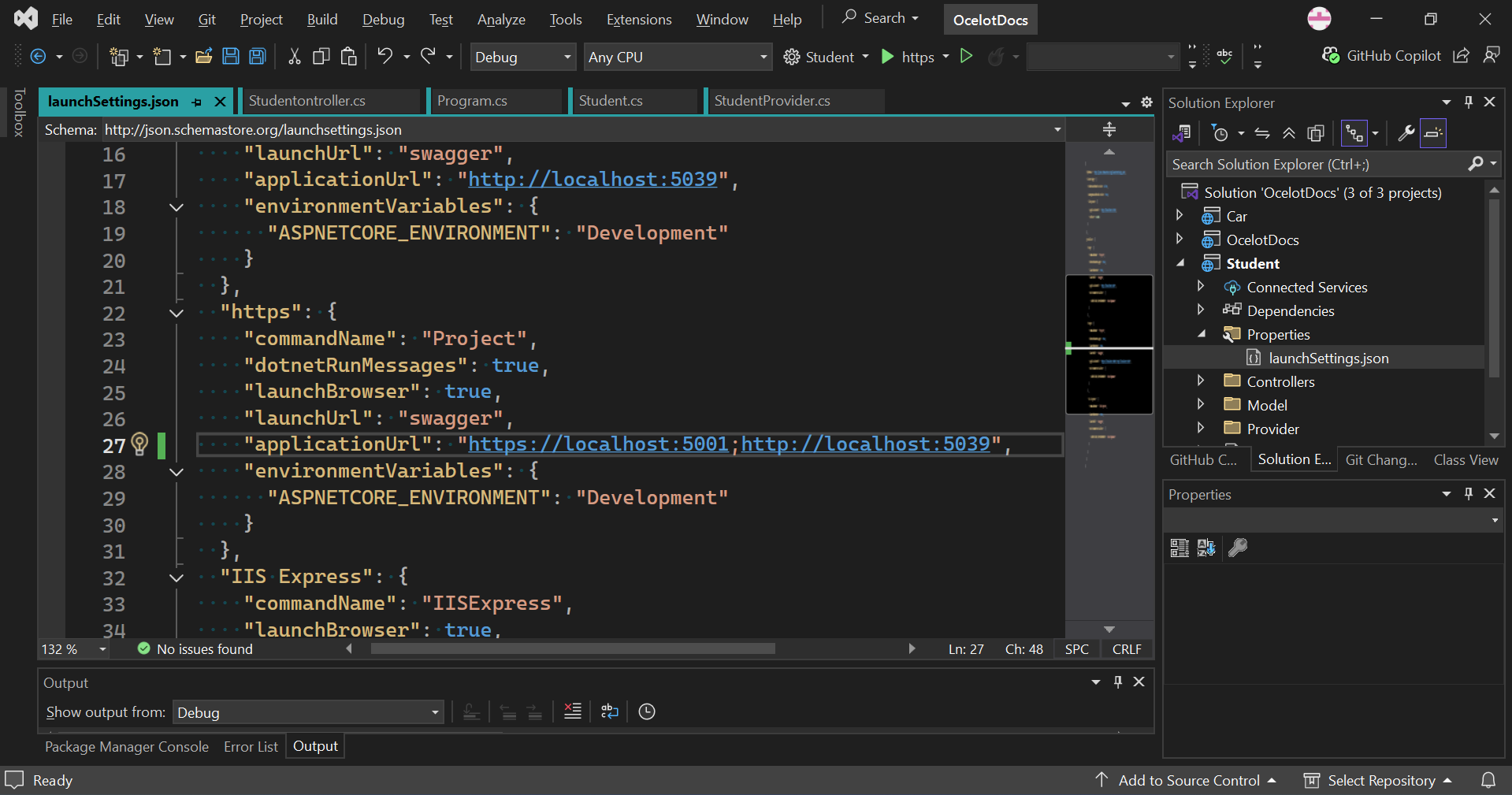
Hình 4.1: : Cài đặt Package Ocelot cho project

Sau khi đã cài đặt Package Ocelot cho project Ocelot, ta tiếp tục khởi tạo 2 project khác là Car, Student và tạo dữ liệu giả cho 2 project này. Với 2 project Car, Student không cần cài đặt package Ocelot.

Tại file launchSettings.json của các project, tiến hành đổi port thành 6001 cho giao thức https của project Car. Đối với project Student cũng đổi port thành 5001 cho giao thức https và port 5000 cho project Ocelot.

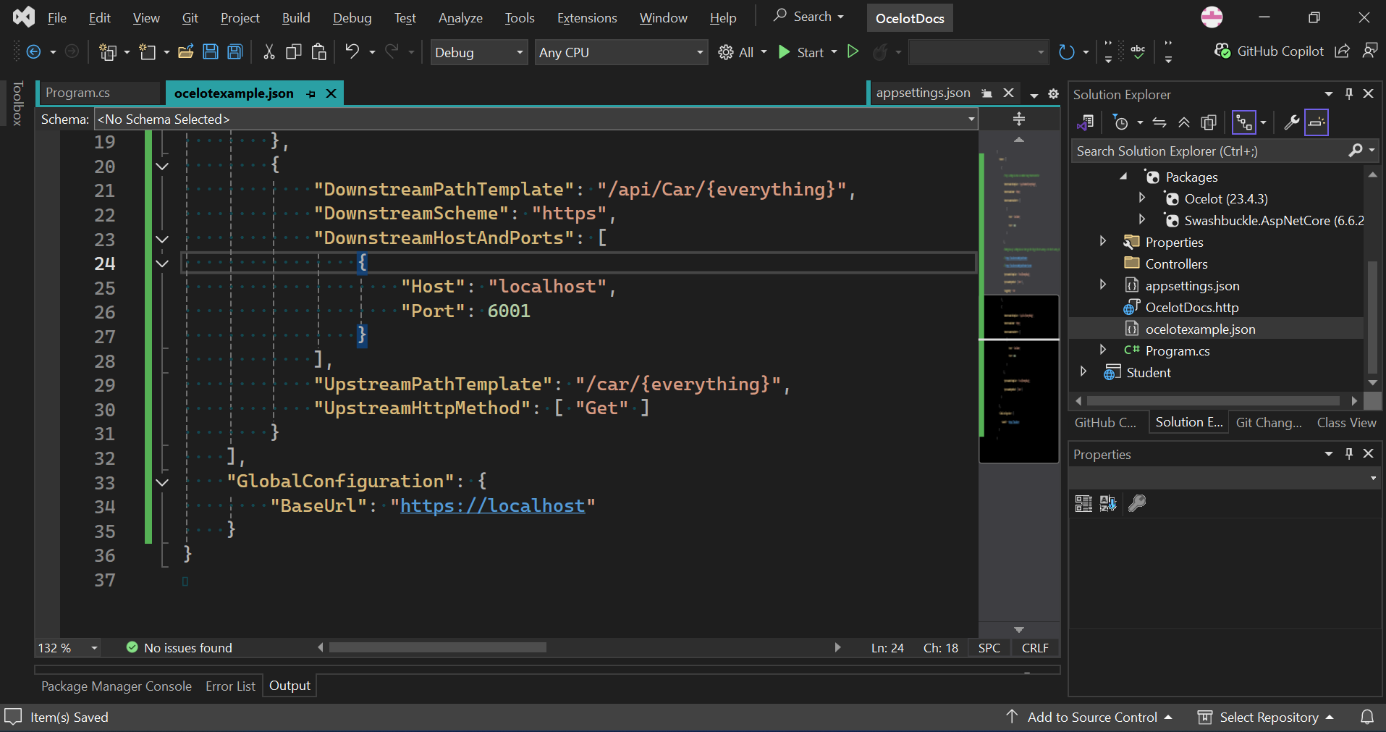
****

Hình 4.2: : Khởi tạo các lớp services

****

Hình 4.3: Đổi port thành 5001 cho project Student

Sau khi đã chuẩn bị 3 project thì quay lại với project Ocelot, tại project này khởi tạo file json với tên ocelotexample và thiết lập tại đây các thuộc tính sau:

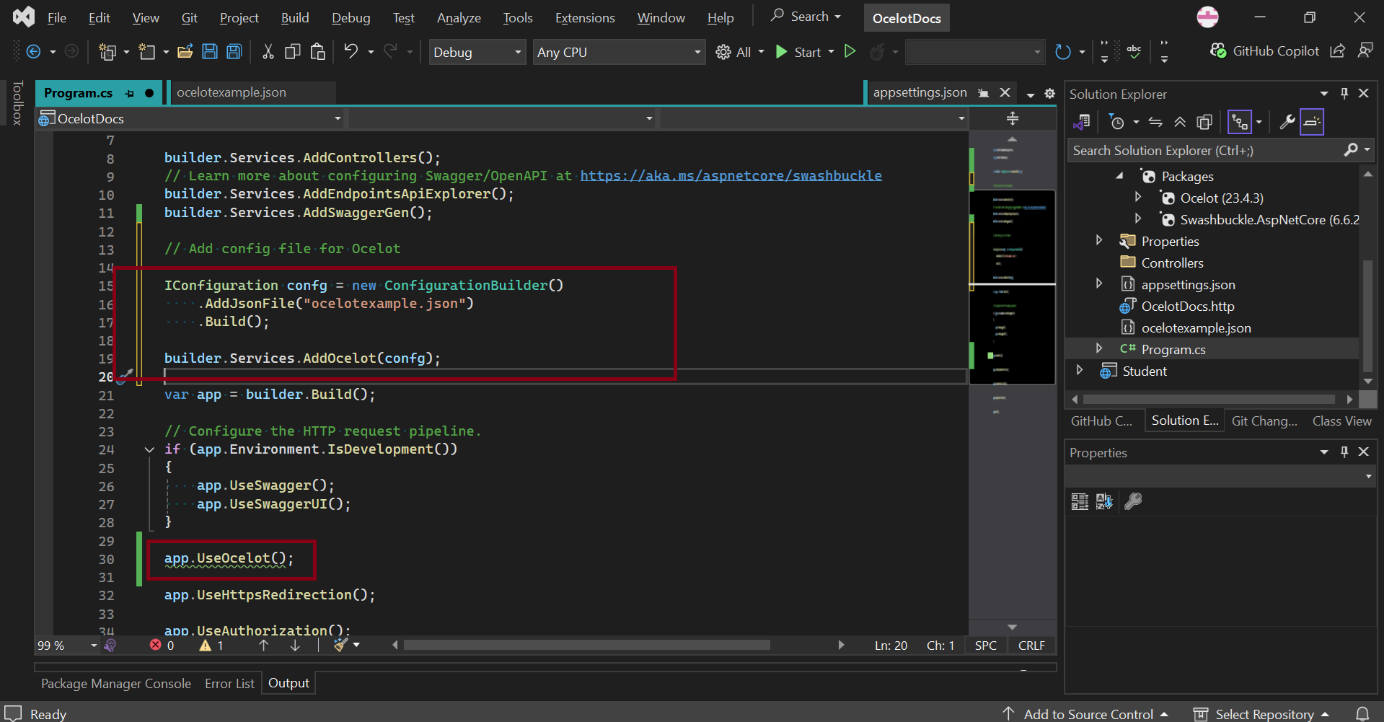


Hình 4.4: Các thuộc tính để thực hiện định tuyến API

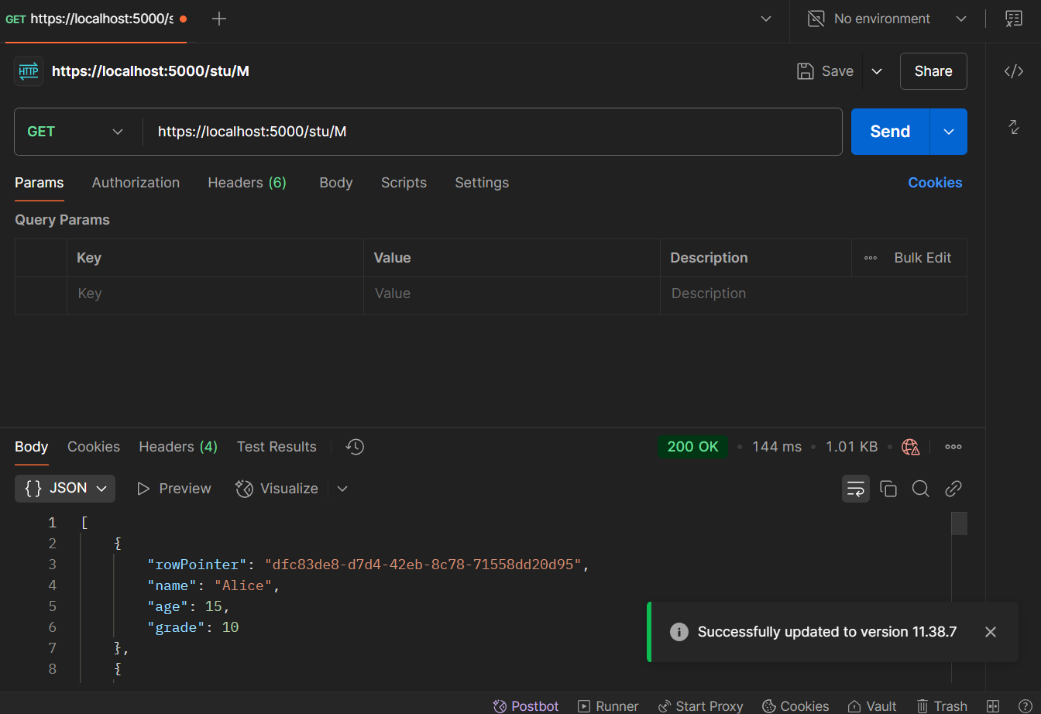
|  |  |
| --- | --- |
| Thuộc tính | Ý nghĩa |
| DownstreamPathTemplate | Khai báo đường dẫn api trong CarController, với {everything} ta có thể khai báo rõ ràng các đường dẫn ta muốn  Ví dụ: api/Car/M hoặc api/Car/MM |
| DownstreamScheme | Khai báo giao thức mà service được cài đặt |
| DownstreamHostAndPorts | Khai báo domain và port của service, ta có thể đặt nhiều domain và port nhằm mục đích Service Discovery và Load Balancing |
| UpstreamPathTemplate | Đường dẫn này là đường dẫn mà client gọi đến API Gateway và từ đó API Gateway sẽ chuyển tiếp request đến Service |
| UpstreamHttpMethod | Phương thức Restful mà API GateWay cho phép truy cập |
| GlobalConfiguration | Tại đây có thể thiết lập các thuộc tính chung dành cho toàn bộ config |

Bảng 4.1: Bảng thuộc tính cơ bản trong Ocelot

Khi hoàn thành thiết lập thành công file ocelotexample.json là đã gần như hoàn thành. Ở bước cuối cùng, ta cần khai báo cho builder biết là có file config mới và cần sử dụng trong file program.cs.

****

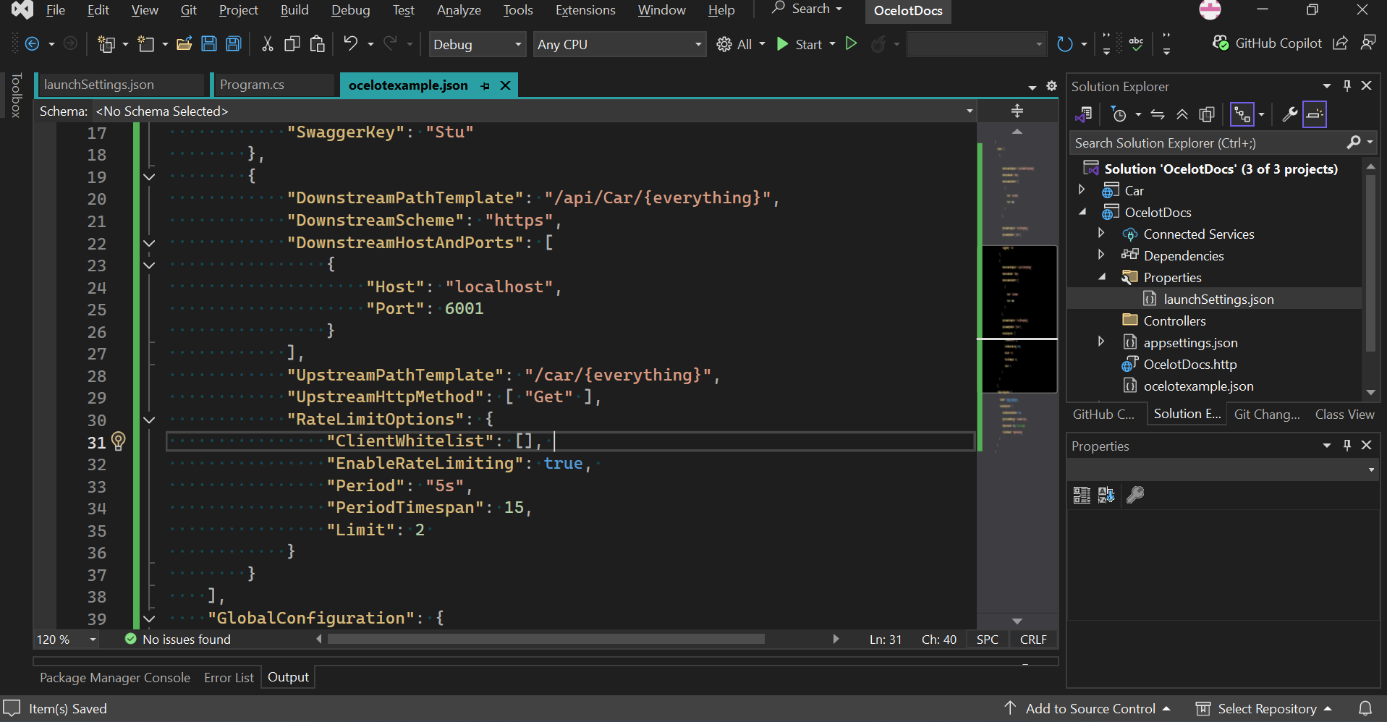
Hình 4.5: Thiết lập file config tại program.cs

Để Ocelot có thể hoạt động tốt nhất, ta cần phải chạy cùng lúc cả 3 services và đều chạy dưới giao thức HTTPS. Sau khi chạy và dùng Swagger của Ocelot gọi tới service chứa list student khối 10 với đường dẫn : https://localhost:5000/stu/M ta sẽ thu được kết quả.

Hình 4.6: Kết quả sau khi gọi API

## Rate Limiting

Rate limiting là chức năng Ocelot hỗ trợ nhằm mục đích giới hạn số lượng request mà các client gửi đến quá nhiều, vì vậy chức năng này làm tránh các services không bị quá tải.

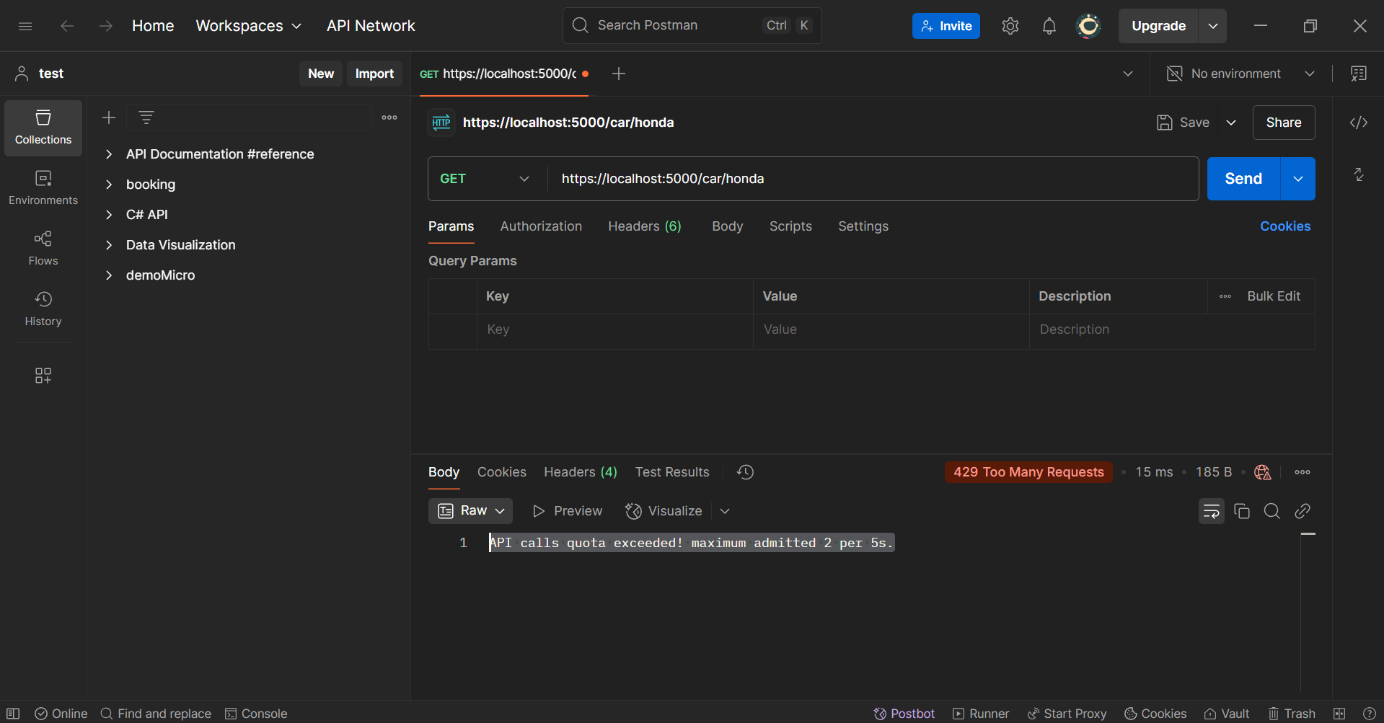
Để thực Rate limiting cho các services, cần kết hợp cấu hình JSON sau vào trong file ocelotexample.js [8].

Hình 4.7: Cấu hình Rate limiting cho service Car

|  |  |
| --- | --- |
| Thuộc tính | Ý nghĩa |
| ClientWhitelist | các client có IP này trong whitelist sẽ không phải bị ảnh hưởng limit request |
| EnableRateLimiting | bật hoặc tắt giới hạn hoạt động |
| Period | Tham số này xác định thời gian mà giới hạn số lần request được áp dụng. Nếu client đạt đến giới hạn request. Client phải đợi thời gian PeriodTimespan kết thúc trước khi đưa ra yêu cầu khác |
| PeriodTimespan | Thời gian mà client có thể thử lại sau khi đạt giới hạn request |
| Limit | Giới hạn số lần request |

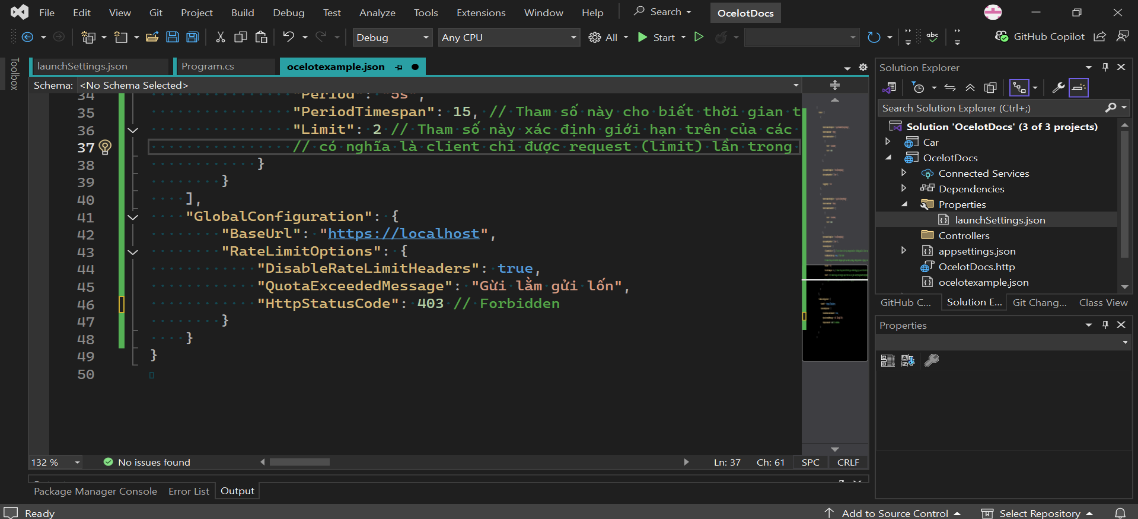
Bảng 4.2: Bảng ý nghĩa thuộc tính Rate Limiting

Với thông số thuộc tính Rate limiting của service car ở hình 4.6, có thể hiểu rằng trong vòng 5 giây (Period) client chỉ có thể gửi tối đa là 2 (limit) Request tới service car. Nếu vượt quá giới hạn thì sẽ phải đợi 15 giây (PeriodTimespan) thì mới có thể tiếp tục gửi request. Khi client phải chờ thời gian PeriodTimespan, hệ thống sẽ thông báo lỗi “API calls quota exceeded! maximum admitted 2 per 5s” kèm theo HttpStatusCode (Mặc định là 429)

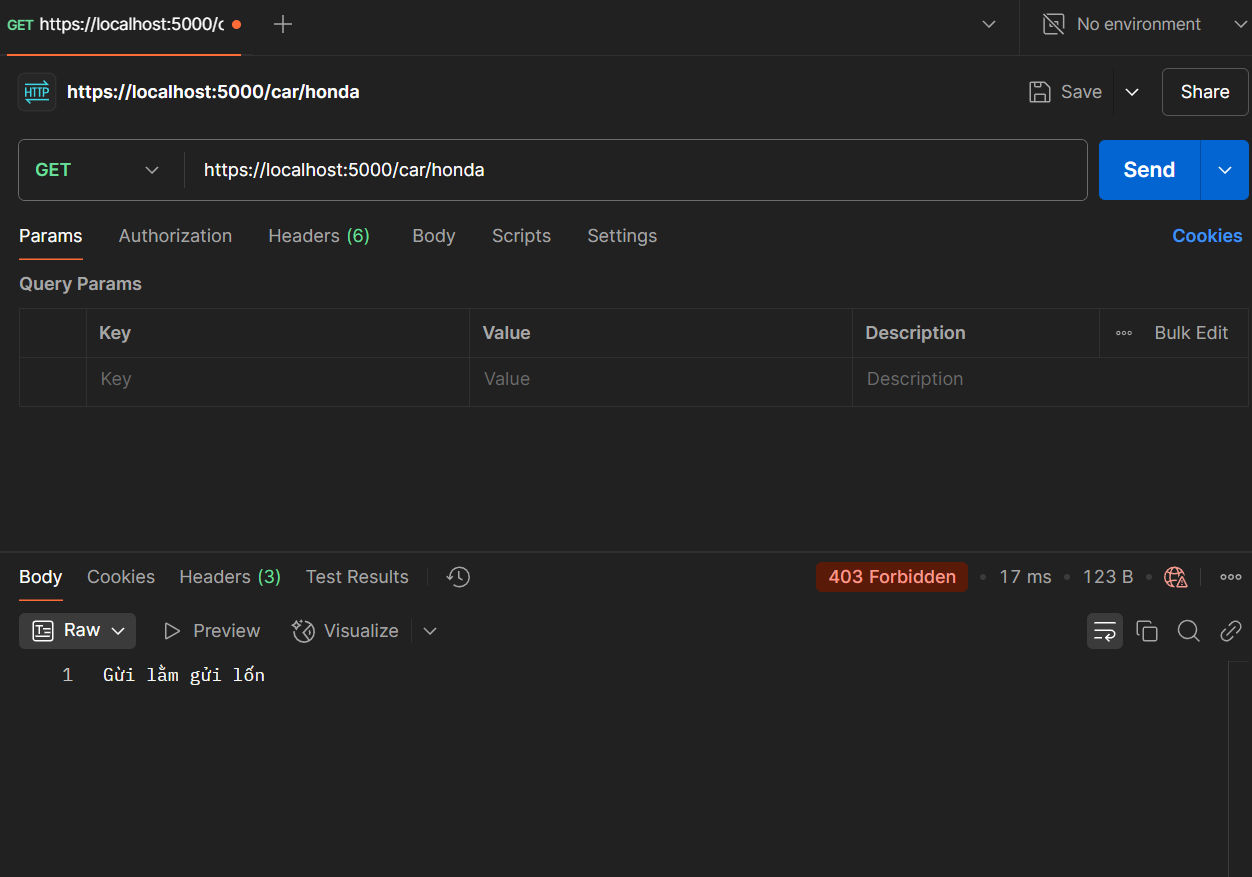


Hình 4.8: Lỗi khi gọi quá nhiều lần tới API Gateway

## Custom Rate Limiting

Ngoài ra Ocelot cũng hỗ trợ custom thông báo lỗi khi hiện thực Rate Limiting thông qua các cấu hình json trong GlobalConfiguration [8].

Hình 4.9: Cấu hình custom Rate limiting

Khi ta gọi vượt quá giới hạn số lần Request cho phép, ta sẽ được kết quả như mong đợi

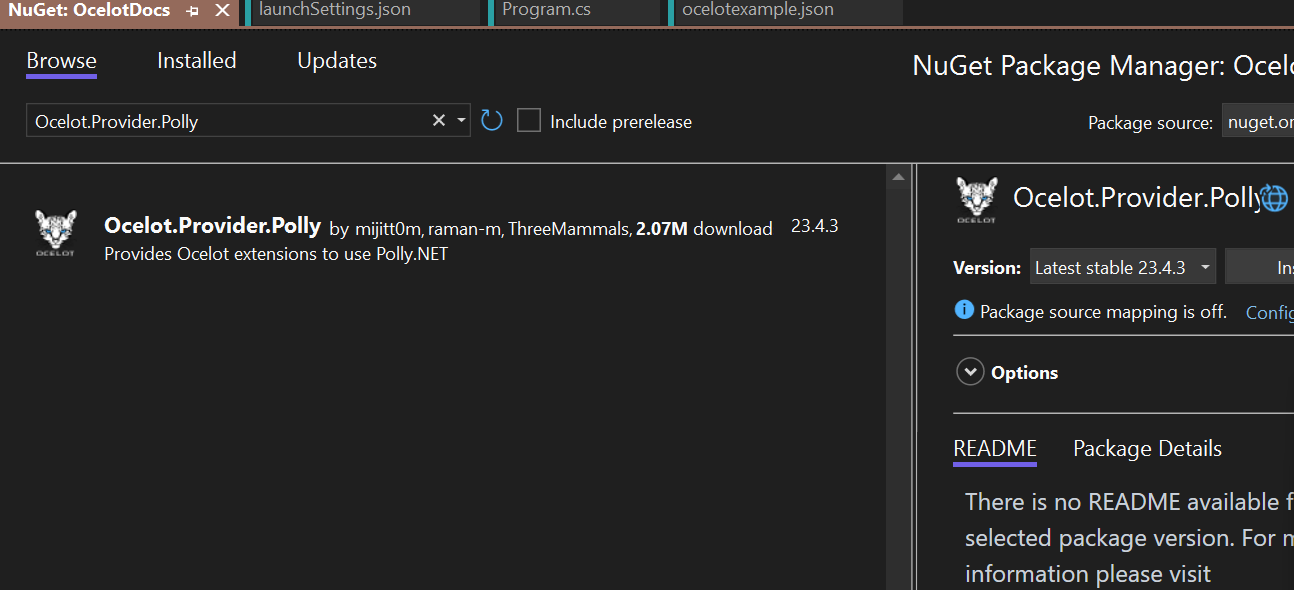
Hình 4.10: Custom Rate limiting

|  |  |
| --- | --- |
| Thông số | Ý nghĩa |
| DisableRateLimitHeaders | Xác định  X-Rate-Limit và Retry-After được gửi kèm trong header của response nếu là false, tắt là true. |
| QuotaExceededMessage | Cấu hình tin nhắn hiển thị cho Client khi vượt quá giới hạn cho phép |
| HttpStatusCode | Cấu hình lỗi trả về |

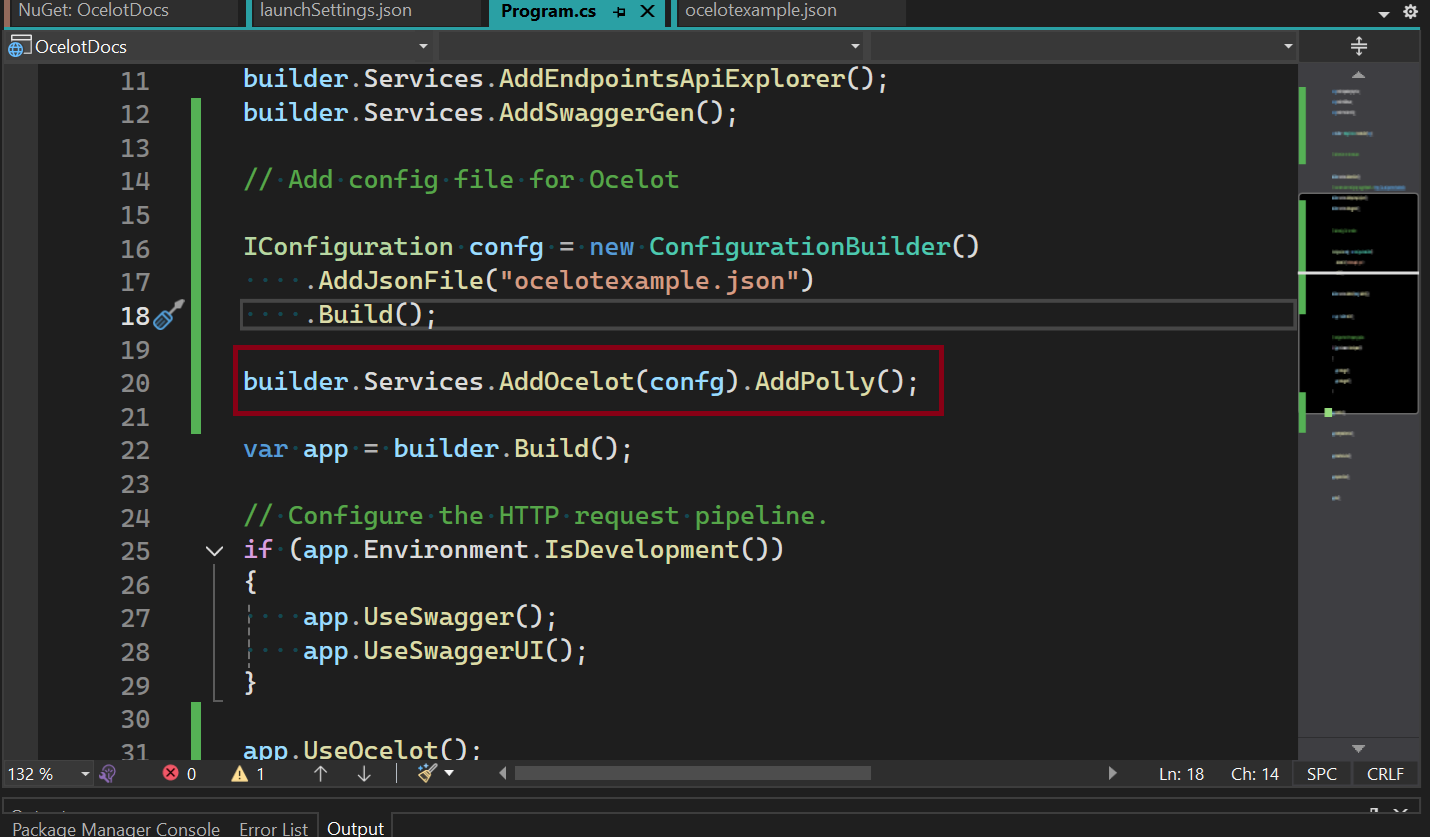
Bảng 4.3: Thông số custom lỗi Rate limiting

## Quality of Service

QoSOptions (Quality of Service Options) trong Ocelot giúp cải thiện tính ổn định của hệ thống bằng cách cung cấp cơ chế xử lý lỗi khi một dịch vụ không phản hồi đúng cách, hệ thống sẽ tự động ngắt request và quăng ra exception khi timeout [9].

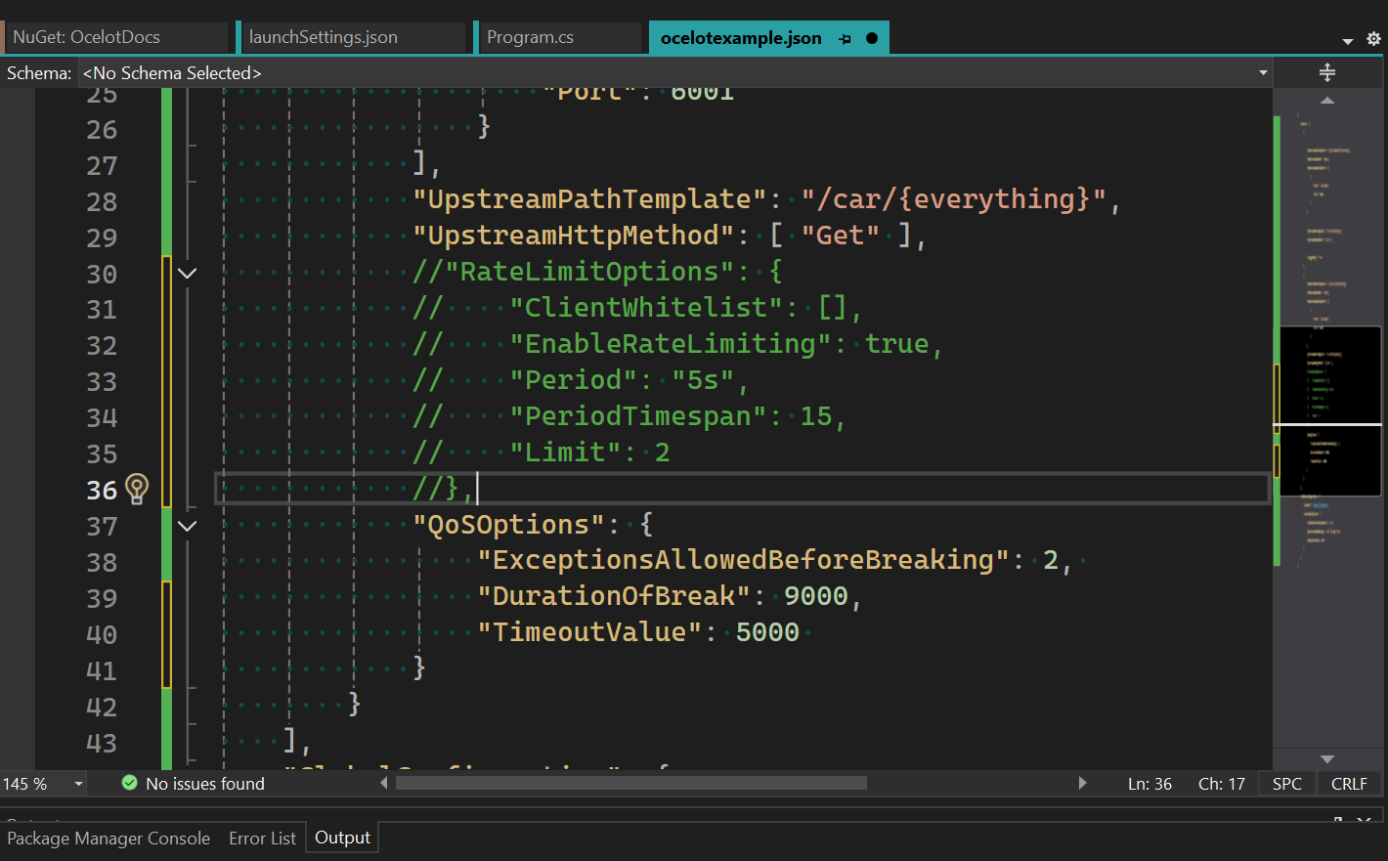
Để thực hiện tính năng này, ta cần cài đặt thêm package Ocelot.Provider.Polly trong nuget và thêm Polly trong file program.cs của project Ocelot.

Hình 4.11: Cài đặt Polly trong nuget



Hình 4.12: Khai báo Polly trong file program.cs

Để thực hiện ta cấu hình trong file ocelotexample.js các thuộc tính ExceptionsAllowedBeforeBreaking, DurationOfBreak, TimeoutValue và tắt đi cả 2 tính năng rate limiting trong Routes và GlobalConfiguration để thử nghiệm SoSOptions trực quan hơn.



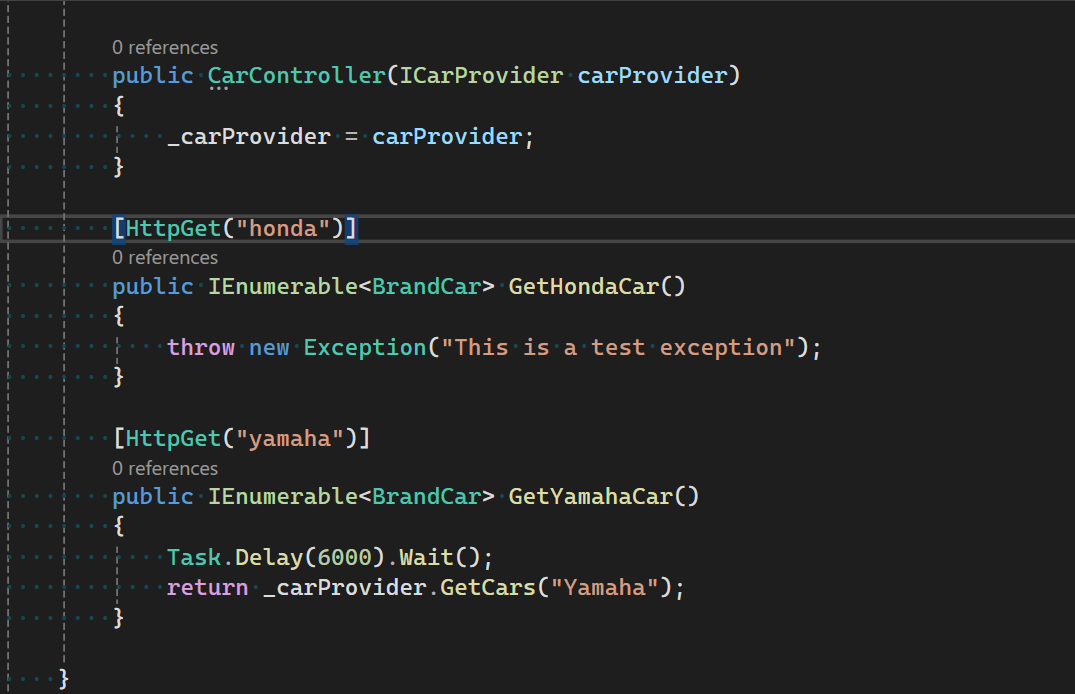
Hình 4.13: Thuộc tính QoSOptions

|  |  |
| --- | --- |
| Thuộc tính | Ý nghĩa |
| ExceptionsAllowedBeforeBreaking | số lần request thất bại trước khi bắt đầu chuyển sang trạng thái lỗi |
| DurationOfBreak | thời gian phải đợi sau khi chuyển sang trạng thái lỗi |
| TimeoutValue | Tự động ngắt request khi bị timeout |

Bảng 4.4: Bảng thuộc tính QoSOptions

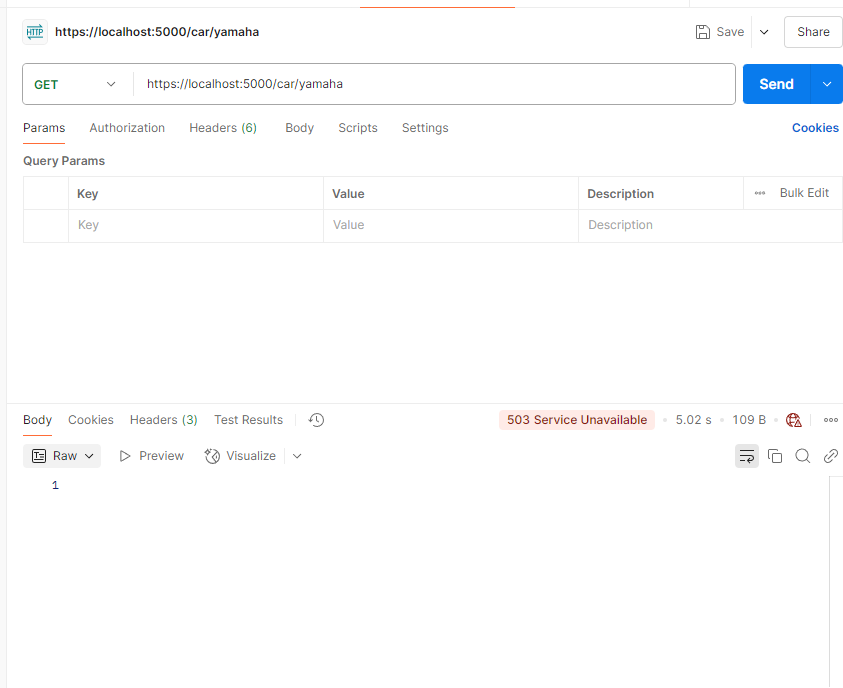
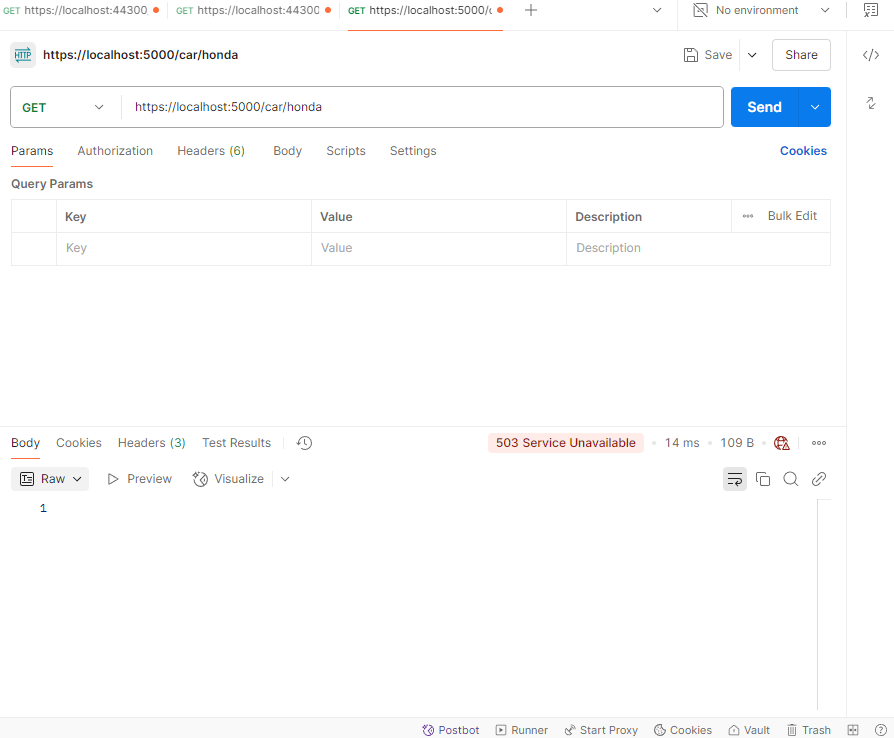
Với thông số trên, có thể hiểu rằng khi Client nhận kết quả trả về là Exception vượt quá 2 lần thì sẽ phải chờ 9 giây để có thể tiếp tục gửi tiếp các request. Hoặc khi Response trả về vượt quá 5 giây thì sẽ ngắt request và bị timeout.

Để rõ ràng về tính năng hơn, tại service car honda sẽ chỉ trả về lỗi và service Yamaha sẽ trả về thời gian lâu hơn 5 giây.



Hình 4.14: Điều chỉnh service car

Sau khi triển khai API Gateway với Ocelot, repsonse sẽ gặp lỗi **503 Service Unavailable** khi cố gắng gọi đến các dịch vụ Yamaha và Honda. Tại service yamaha ta có thể thấy thời gian response bên phải HTTPCode 503 là 5.02 giây – vượt quá 5 giây. Vì vậy sẽ ngắt kết nối request và quăng ra timeout ngay lập tức.

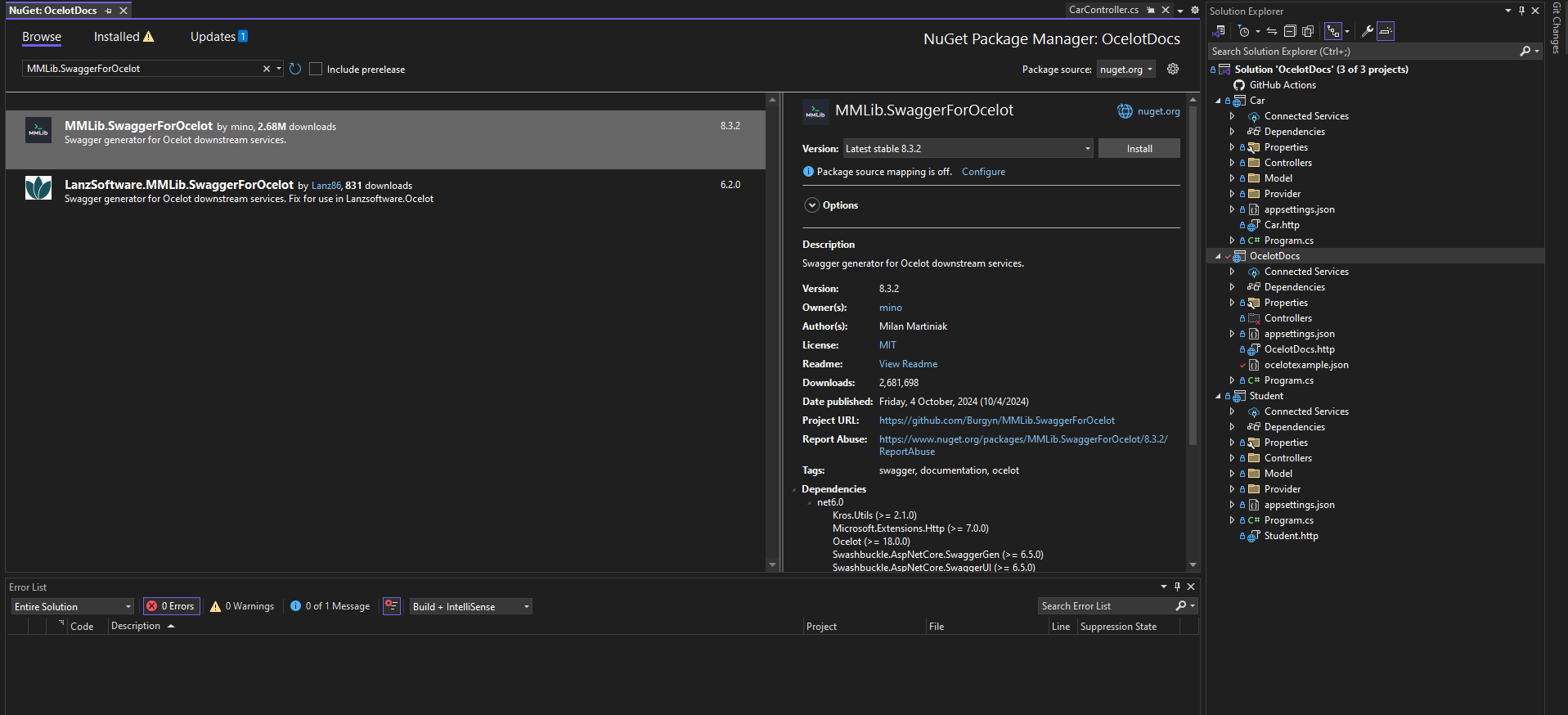


Hình 4.15: Lỗi khi gọi tới honda

Hình 4.16: Lỗi khi gọi tới yamaha

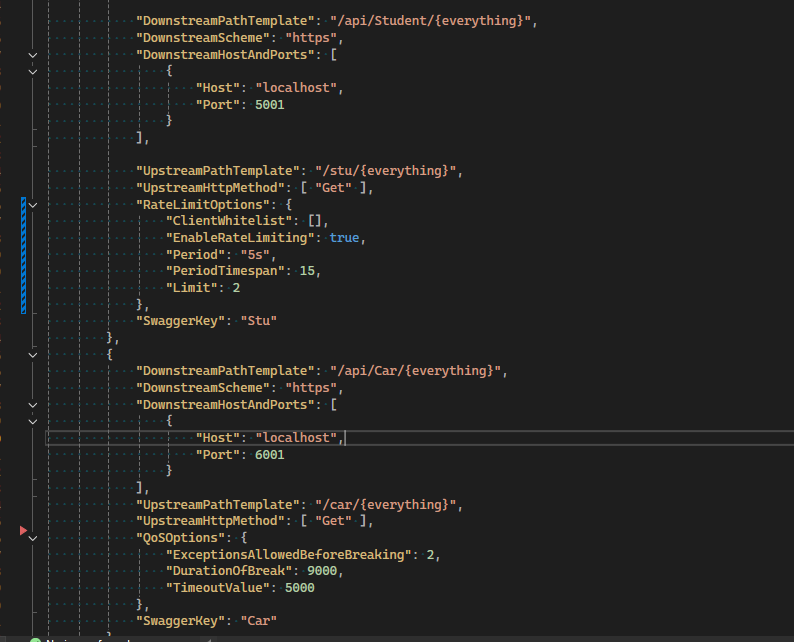
## SwaggerForOcelot

SwaggerForOcelot là một thư viện mở rộng giúp tích hợp Swagger UI với Ocelot API Gateway trong các ứng dụng .NET. Mặc định, Ocelot chỉ đảm nhận việc định tuyến các request từ API Gateway đến các dịch vụ downstream mà không cung cấp tài liệu API tập trung. SwaggerForOcelot giúp khắc phục vấn đề này bằng cách tổng hợp và hiển thị tài liệu Swagger từ nhiều dịch vụ microservice dưới một giao diện chung thông qua API Gateway [10].

Để có thể cấu hình và sử dụng, trước tiên phải cài đặt package MMLib.SwaggerForOcelot cho project Ocelot.

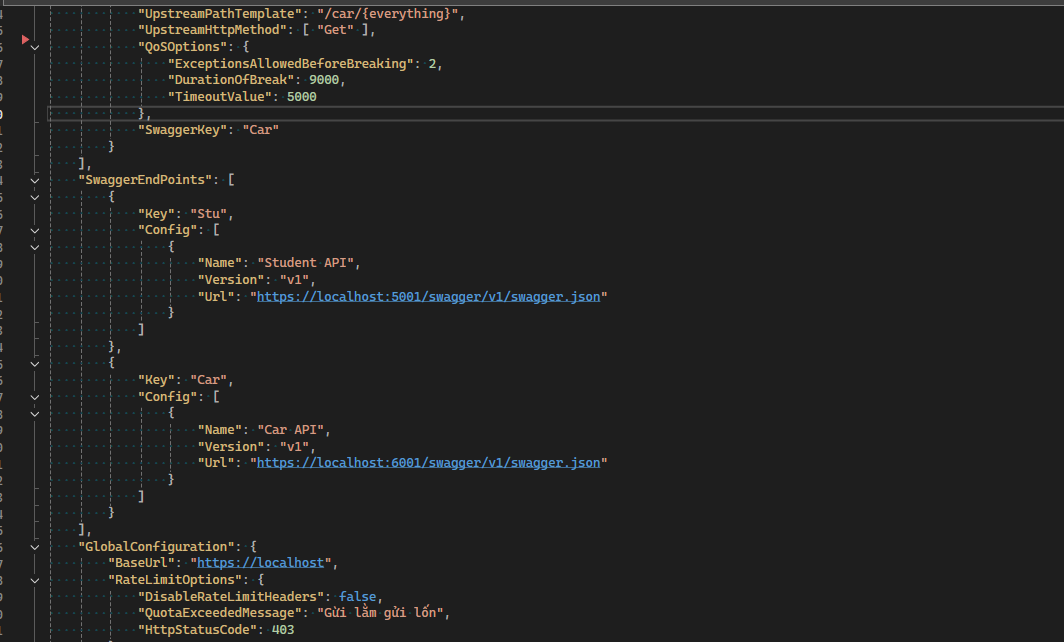
Hình 4.17: Cài đặt MMLib.SwaggerForOcelot

khi tích hợp SwaggerForOcelot, mỗi microservice sẽ có một Swagger Key để ánh xạ services API tương ứng. Swagger Key giúp Ocelot xác định API của từng dịch vụ downstream và hiển thị tài liệu Swagger một cách chính xác. Vì vậy ta khai báo "SwaggerKey" của Student là “Stu” và "SwaggerKey" của Car là “Car”.



Hình 4.18: Khai báo SwaggerKey

SwaggerForOcelot cần một tệp cấu hình riêng là SwaggerEndpoint, và kết hợp với SwaggerKey để xác định các dịch vụ Swagger cần hợp nhất. Ở đây SwaggerKey: "car" sẽ liên kết với tài liệu Swagger của CarService. SwaggerKey: "student" sẽ Liên kết với tài liệu Swagger của StudentService.



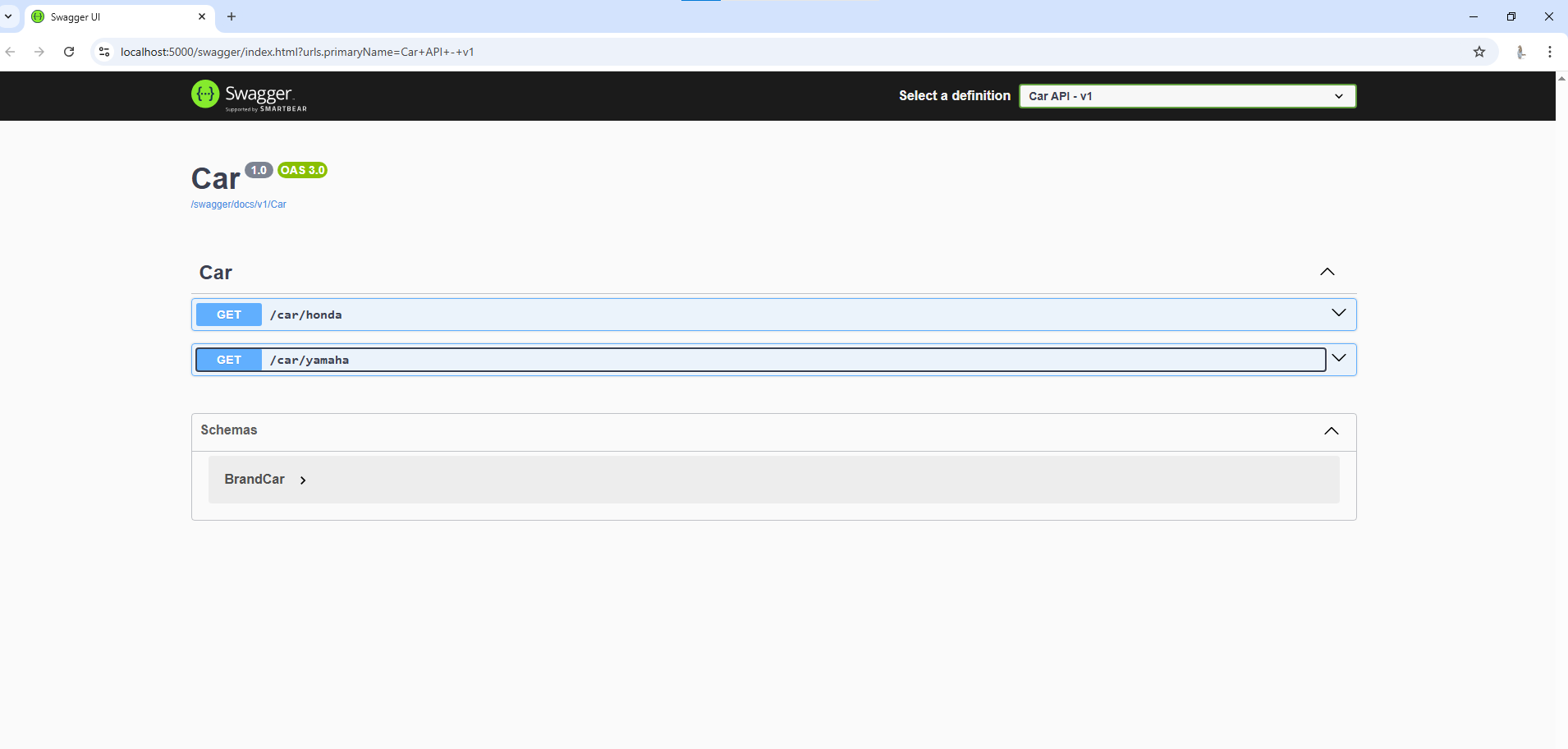
Hình 4.19: Khai báo SwaggerEndpoint

Sau khi đã thiết lập routes trong ocelot.json và định nghĩa Swagger Key trong SwaggerEndpoint, bước cuối cùng là cấu hình các services cần thiết trong Program.cs của dự án Ocelot API Gateway.



Hình 4.20: Add service trong program.cs

Sau khi cài đặt các bước thành công, đây là kết quả Swagger ta thu được từ Swagger Ocelot



Hình 4.21: Kết quả swagger Ocelot

TÀI LIỆU THAM KHẢO

|  |  |
| --- | --- |
| [1] | M. Tanner, "What Is a Monolithic Application? Everything You Need to Know," 13 September 2024. [Online]. Available: https://vfunction.com/blog/what-is-monolithic-application/. [Accessed 30 3 2025]. |
| [2] | A. Davis, "Monolithic vs Microservices Architecture: Pros, Cons and Which to Choose," 9 10 2023. [Online]. Available: https://www.openlegacy.com/blog/monolithic-application. [Accessed 30 3 2025]. |
| [3] | D. V. Nam, "Đôi nét về Microservice architecture và Monolithic architecture," 6 3 2018. [Online]. Available: https://viblo.asia/p/doi-net-ve-microservice-architecture-va-monolithic-architecture-XL6lAAvrlek. [Accessed 30 3 2025]. |
| [4] | J. Vietnam, "MONOLITH VÀ MICROSERVICES: SỰ KHÁC BIỆT GIỮA HAI KIẾN TRÚC WEBSITE," 16 6 2023. [Online]. Available: https://www.linkedin.com/pulse/monolith-v%C3%A0-microservices-s%E1%BB%B1-kh%C3%A1c-bi%E1%BB%87t-gi%E1%BB%AFa-hai-ki%E1%BA%BFn-tr%C3%BAc/. [Accessed 30 3 2025]. |
| [5] | "API Gateway là gì," 08 Aug 2024. [Online]. Available: https://200lab.io/blog/api-gateway-la-gi. [Accessed 30 March 2025]. |
| [6] | H. Đinh, "Triển khai API Gateway trong .NET Core với Ocelot - Giới thiệu," 25 June 2022. [Online]. Available: https://viblo.asia/p/trien-khai-api-gateway-trong-net-core-voi-ocelot-gioi-thieu-GrLZDop2Kk0. [Accessed 30 March 2025]. |
| [7] | T. O. team, "Getting Started," [Online]. Available: https://ocelot.readthedocs.io/en/latest/introduction/gettingstarted.html. [Accessed 30 March 2025]. |
| [8] | T. O. team., "Rate Limiting," [Online]. Available: https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html. [Accessed 30 March 2025]. |
| [9] | T. O. team., "Quality of Service," [Online]. Available: https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html. [Accessed 30 March 2025]. |
| [10] | Nuget, "Swagger For Ocelo," [Online]. Available: https://www.nuget.org/packages/MMLib.SwaggerForOcelot/. [Accessed 3 30 2025]. |