1- روتینگ(Routing)

- Ocelotبه شما امکان می‌دهد درخواست‌های ورودی (upstream) را به سرویس‌های مناسب در بک‌اند (downstream) هدایت کنید. این روتینگ بر اساس الگوهایی است که در فایل پیکربندی مشخص می‌شود و می‌توانید تعیین کنید که هر مسیر به کدام سرویس داخلی منتقل شود.

2- بالانس بار(Load Balancing)

- در صورتی که چندین نمونه از یک سرویس در حال اجرا باشند، Ocelotمی‌تواند ترافیک را بین آن‌ها توزیع کند. این قابلیت موجب افزایش دسترسی‌پذیری و کارایی سیستم می‌شود. روش‌های مختلفی برای بالانس بار پشتیبانی می‌شوند، از جمله Round Robin و Least Connection.

3- حفاظت از درخواست‌ها(Rate Limiting & Throttling)

- Ocelotاز محدودسازی تعداد درخواست‌ها در بازه‌های زمانی خاص پشتیبانی می‌کند. این ویژگی برای جلوگیری از سوءاستفاده یا حملات DDoS مفید است و به شما امکان می‌دهد نرخ درخواست‌ها را برای هر کاربر یا API محدود کنید.

4- ذخیره‌سازی موقت(Caching)

- Ocelotامکان کش کردن پاسخ‌ها را فراهم می‌کند، به این معنا که برخی از پاسخ‌های API برای مدت زمان مشخصی در حافظه نگه داشته می‌شوند. این ویژگی کمک می‌کند تا بار روی سرورهای بک‌اند کاهش یابد و زمان پاسخ‌دهی بهبود یابد.

5- تایید هویت و مجوزدهی(Authentication & Authorization)

- Ocelotاز احراز هویت JWT و OAuth2 پشتیبانی می‌کند و به شما امکان می‌دهد از کاربران بخواهید قبل از دسترسی به سرویس‌ها اعتبارسنجی شوند. همچنین می‌توانید سطوح دسترسی مختلفی برای هر درخواست تنظیم کنید.

6- تبدیل درخواست و پاسخ(Request & Response Transformation)

- Ocelotبه شما اجازه می‌دهد که درخواست‌ها و پاسخ‌ها را قبل از رسیدن به مقصد تغییر دهید. برای مثال، می‌توانید داده‌های درخواست یا پاسخ را تغییر دهید، هدرهای جدید اضافه کنید، یا برخی اطلاعات غیرضروری را حذف کنید.

7- پشتیبانی از پروتکل‌های مختلف

- Ocelotاز پروتکل‌های HTTP و HTTPS پشتیبانی می‌کند و می‌تواند درخواست‌های HTTP را به سرویس‌های gRPC یا پروتکل‌های دیگر هدایت کند. این ویژگی به سازگاری سرویس‌های مختلف کمک می‌کند.

8- مسیرهای دینامیک(Dynamic Routing)

- Ocelotاین امکان را دارد که با استفاده از پارامترهای ورودی مسیرها را دینامیک کند. این ویژگی کمک می‌کند تا بتوانید مسیرهای متغیر و پویا برای درخواست‌ها تعریف کنید و از آن‌ها برای روتینگ استفاده کنید.

9- مانیتورینگ و گزارش‌گیری(Logging & Monitoring)

- Ocelotبه شما این امکان را می‌دهد که گزارش‌های مربوط به درخواست‌ها و پاسخ‌ها را ضبط کنید. این قابلیت برای عیب‌یابی، شناسایی مشکلات و بهبود کارایی سیستم بسیار مفید است.

10- تجمیع درخواست‌ها(Request Aggregation)

- Ocelotمی‌تواند چندین درخواست را به یک درخواست تجمیع کند و پاسخ‌ها را به صورت یکپارچه به کاربر بازگرداند. این ویژگی در سناریوهایی که نیاز به جمع‌آوری داده‌ها از چندین سرویس وجود دارد بسیار کاربردی است.

11- پشتیبانی از Canary Testing

- Ocelotاین امکان را فراهم می‌کند که نسخه‌های مختلف از یک سرویس را در محیط تولید آزمایش کنید. برای مثال، می‌توانید درصد مشخصی از ترافیک را به یک نسخه جدید هدایت کرده و بازخورد کاربران را بسنجید.

Routing

روتینگ (Routing) در Ocelot به این معنی است که درخواست‌های ورودی از سمت کاربران یا سرویس‌های دیگر را به سرویس‌های مناسب در بک‌اند هدایت کنیم. فرض کنید یک کاربر می‌خواهد به اطلاعات کاربران دسترسی پیدا کند و به جای اینکه مستقیم به سرویس کاربران وصل شود، درخواستش را به API Gateway بفرستد. در اینجا Ocelot به عنوان یک واسطه عمل کرده و درخواست کاربر را به سرویس مناسب هدایت می‌کند.

چطور روتینگ کار می‌کند؟

برای اینکه Ocelot بداند هر درخواست باید به کجا برود، از قوانین روتینگ استفاده می‌کنیم که در یک فایل به نام ocelot.jsonتعریف می‌شود. این فایل به Ocelot می‌گوید، اگر مثلاً درخواستی به مسیر gateway/users/آمد، این درخواست باید به سرویس اصلی کاربران در مسیر api/users/منتقل شود.

یک مثال ساده

فرض کنید دو سرویس داریم:

- **سرویس کاربران که اطلاعات کاربران را برمی‌گرداند و روی پورت 5001در حال اجراست.**
- **سرویس سفارشات که لیست سفارشات را برمی‌گرداند و روی پورت 5002در حال اجراست.**

در فایل ocelot.json، اینطور روتینگ را تعریف می‌کنیم:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/users",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/gateway/users",
      "UpstreamHttpMethod": [ "GET", "POST" ]
    },
    {
      "DownstreamPathTemplate": "/api/orders",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5002
        }
```

```
    ],
    "UpstreamPathTemplate": "/gateway/orders",
    "UpstreamHttpMethod": [ "GET", "POST" ]
  }
 ]
}
```

توضیح بخش‌های مختلف این فایل

- **DownstreamPathTemplate: این بخش مشخص می‌کند که درخواست بعد از عبور از API Gateway باید به کدام مسیر در سرویس بک‌اند ارسال شود.**
- **DownstreamHostAndPorts: اینجا آدرس و پورت سرویسی که درخواست باید به آن برود، مشخص می‌شود. مثلاً برای سرویس کاربران، آدرس localhost:5001است.**
- **UpstreamPathTemplate: این بخش مسیر ورودی است که کاربر یا کلاینت به آن درخواست ارسال می‌کند. در این مثال، وقتی کاربر به /gateway/users درخواست می‌فرستد، Ocelot آن را به مسیر /api/users هدایت می‌کند.**
- **UpstreamHttpMethod: این بخش مشخص می‌کند که چه نوع درخواست‌هایی مثل GET و POST قابل قبول هستند.**

چرا این روش مفید است؟

با این ساختار، همه درخواست‌ها از طریق یک نقطه (API Gateway) مدیریت می‌شوند و کاربران نیازی ندارند که بدانند سرویس‌ها در کجا و روی چه پورتی در حال اجرا هستند. هر تغییری هم در سرویس‌های بک‌اند، بدون تغییر برای کاربران و تنها با ویرایش فایل ocelot.json انجام می‌شود.

خلاصه

روتینگ در Ocelot باعث می‌شود که درخواست‌های کاربران به سرویس‌های مناسب هدایت شوند و همه چیز از یک نقطه کنترل شود. این روش مدیریت درخواست‌ها را آسان می‌کند و انعطاف بیشتری در تغییرات به ما می‌دهد.

بالانس بار (Load Balancing)

بالانس بار (Load Balancing) یکی از ویژگی‌های مهم در Ocelot است که کمک می‌کند ترافیک بین چندین نمونه (instance) از یک سرویس توزیع شود. هدف این ویژگی این است که اگر یک سرویس درخواست‌های زیادی دریافت کرد، ترافیک به طور مساوی بین همه سرورهایی که آن سرویس را ارائه می‌دهند تقسیم شود تا هیچ سروری بیش از حد مشغول نشود و عملکرد کلی سیستم بهتر شود.

چرا بالانس بار مهم است؟

فرض کنید یک سرویس در پروژه ما به شدت محبوب است و تعداد زیادی درخواست از آن دریافت می‌شود. اگر این سرویس فقط روی یک سرور اجرا شود، ممکن است به سرعت کند شود یا حتی از کار بیفتد. اما اگر چندین نسخه از این سرویس (مثلاً روی چند سرور مختلف) داشته باشیم، Ocelotمی‌تواند درخواست‌ها را بین این نسخه‌ها پخش کند و به هر نسخه تنها بخشی از درخواست‌ها برسد.

چطور بالانس بار در Ocelot کار می‌کند؟

برای اینکه Ocelot بتواند ترافیک را بین چند نمونه از یک سرویس توزیع کند، باید در فایل پیکربندی (ocelot.json) مشخص کنیم که سرویس مربوطه چند نمونه دارد و هر کدام در کجا قرار دارند. سپس یک الگوریتم بالانس بار انتخاب می‌کنیم تا Ocelot طبق آن ترافیک را توزیع کند.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/users",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5001 },
        { "Host": "localhost", "Port": 5002 },
        { "Host": "localhost", "Port": 5003 }
      ],
      "UpstreamPathTemplate": "/gateway/users",
      "LoadBalancerOptions": {
        "Type": "RoundRobin"
      }
    }
  ]
}
```

یک مثال از تنظیمات بالانس بار

فرض کنید سرویس کاربران را داریم که سه نمونه از آن روی پورت‌های 5001، 5002و 5003اجرا شده‌اند. تنظیمات زیر را در فایل ocelot.jsonقرار می‌دهیم:

توضیح بخش‌های مختلف این تنظیمات

- **DownstreamHostAndPorts:در اینجا ما سه نمونه از سرویس کاربران داریم، که Ocelot می‌تواند درخواست‌ها را بین آن‌ها توزیع کند.**
- **LoadBalancerOptions:این بخش برای تنظیمات بالانس بار است. در اینجا نوع بالانس بار را مشخص می‌کنیم.**

الگوریتم‌های بالانس بار موجود درOcelot

Ocelotاز چندین الگوریتم برای توزیع ترافیک پشتیبانی می‌کند:

1. **Round Robin:این الگوریتم به ترتیب، درخواست‌ها را بین سرورها تقسیم می‌کند. مثلاً اولین درخواست به سرور 5001، دومین درخواست به سرور 5002، و سومین درخواست به سرور 5003 می‌رود و سپس دوباره به 5001 برمی‌گردد.**
2. **Least Connection:این الگوریتم درخواست‌ها را به سروری که کمترین تعداد اتصال فعال را دارد ارسال می‌کند. این روش برای سرویس‌هایی که بار غیر یکنواخت دارند مناسب‌تر است.**
3. **No Load Balancer:می‌توان هیچ بالانس باری انتخاب نکرد، در این صورت تنها به اولین سرور موجود درخواست فرستاده می‌شود.**

چرا از بالانس بار استفاده کنیم؟

- **افزایش دسترسی‌پذیری :اگر یک سرور از کار بیفتد، Ocelotمی‌تواند درخواست‌ها را به سرورهای دیگر بفرستد.**

- **بهبود عملکرد :با توزیع بار، از ایجاد ترافیک سنگین روی یک سرور خاص جلوگیری می‌شود و پاسخگویی بهتر می‌شود.**
- **انعطاف‌پذیری :می‌توان به سادگی سرورها را اضافه یا حذف کرد و تغییرات زیادی در معماری اعمال نکرد.**

خلاصه

بالانس بار در Ocelot باعث می‌شود درخواست‌ها به طور مساوی بین سرورهای مختلف یک سرویس توزیع شوند. این ویژگی کمک می‌کند تا هم سیستم سریع‌تر عمل کند و هم اگر یک سرور از دسترس خارج شد، کاربر متوجه نشود و درخواست به سرور دیگری منتقل شود.

RateLimitat

حفاظت از درخواست‌ها (Rate Limiting & Throttling) در Ocelot به شما کمک می‌کند تعداد درخواست‌های ورودی به سیستم را کنترل کنید تا از مصرف بیش از حد منابع توسط یک کاربر یا در یک بازه زمانی جلوگیری کنید. این ویژگی به خصوص برای جلوگیری از حملات DDoS و سوءاستفاده‌های احتمالی ازAPI ها مفید است.

چرا Rate Limiting مهم است؟

تصور کنید که یک سرویس محبوب دارید و تعداد زیادی کاربر به آن دسترسی پیدا می‌کنند. اگر برخی از کاربران درخواست‌های زیادی در مدت زمان کوتاهی ارسال کنند، ممکن است سرویس تحت فشار قرار بگیرد و عملکرد آن برای دیگر کاربران کند شود. با Rate Limiting می‌توانید تعداد درخواست‌های مجاز برای هر کاربر را محدود کنید تا سرویس همواره در دسترس و با کیفیت باقی بماند.

چطور Rate Limiting در Ocelot کار می‌کند؟

درOcelot ، شما می‌توانید تنظیمات Rate Limiting را برای هر مسیر API در فایل ocelot.jsonتعریف کنید. به این ترتیب، مشخص می‌کنید که یک کاربر در بازه‌های زمانی مشخصی تنها تعداد محدودی درخواست می‌تواند ارسال کند. اگر تعداد درخواست‌های کاربر بیشتر از مقدار تعیین‌شده شود، Ocelotبه او پاسخ خطا می‌دهد و درخواستش را مسدود می‌کند.

یک مثال ازRate Limiting

فرض کنید می‌خواهیم کاربرها بتوانند در هر دقیقه حداکثر ۵۰ درخواست به مسیر gateway/usersارسال کنند. برای این کار، تنظیمات زیر را در فایل ocelot.json اضافه می‌کنیم:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/users",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5001 }
      ],
      "UpstreamPathTemplate": "/gateway/users",
      "RateLimitOptions": {
        "ClientWhitelist": [],
```

```
      "EnableRateLimiting": true,
      "Period": "1m",
      "PeriodTimespan": 60,
      "Limit": 50
    }
  }
 ]
}
```

توضیح بخش‌های مختلف این تنظیمات

- **EnableRateLimiting**: این گزینه فعال‌سازی Rate Limiting را مشخص می‌کند. با تنظیم true، Ocelotمحدودسازی نرخ درخواست‌ها را انجام می‌دهد.
- **Period**: بازه زمانی که محدودیت برای آن اعمال می‌شود. در اینجا 1mبه معنی یک دقیقه است.
- **PeriodTimespan**: مدت زمان به ثانیه. برای مثال 60به معنی ۶۰ ثانیه است.
- **Limit**: حداکثر تعداد درخواست‌هایی که کاربر در طول بازه زمانی مشخص‌شده می‌تواند ارسال کند. در این مثال، هر کاربر تنها می‌تواند ۵۰ درخواست در هر دقیقه ارسال کند.
- **ClientWhitelist**: لیستی از کلاینت‌ها که از این محدودیت‌ها معاف هستند (مثلاً کاربران یا سرورهای خاصی که نیاز به دسترسی بدون محدودیت دارند).

وقتی نرخ درخواست‌ها از حد تعیین‌شده بیشتر شود چه اتفاقی می‌افتد؟

اگر کاربری بیش از تعداد مجاز درخواست ارسال کند، Ocelotبه او یک پاسخ با کد وضعیت 429 (Too Many Requests) می‌دهد که به کاربر اعلام می‌کند تعداد درخواست‌ها بیش از حد مجاز است و باید کمی صبر کند تا بتواند دوباره درخواست ارسال کند.

انواع Rate Limiting درOcelot

Ocelotبه شما این امکان را می‌دهد که Rate Limiting را بر اساس موارد مختلفی اعمال کنید:

- **IPآدرس‌ها** :درخواست‌ها را بر اساس IP آدرس کاربر محدود کنید.
- **تعداد درخواست به ازای هر مسیر** :مثلاً تعداد درخواست‌های مجاز برای مسیرهای خاص.
- **شناسه کاربر یا کلاینت** :برای هر کاربر یا کلاینت می‌توان محدودیت خاصی تعریف کرد.

مزایای استفاده ازRate Limiting

- **حفاظت از منابع** :از مصرف بی‌رویه منابع و سرور جلوگیری می‌شود.
- **کاهش ریسک حملات** :**DDoS** با محدودسازی تعداد درخواست‌ها، می‌توان تا حدی از حملات DDoS جلوگیری کرد.
- **بهبود تجربه کاربری** :کاربران عادی از کارایی بهتر سیستم بهره‌مند می‌شوند و کاربرانی که سعی در سوءاستفاده دارند محدود می‌شوند.

خلاصه

حفاظت از درخواست‌ها با Rate Limiting در Ocelot به شما کمک می‌کند تا از منابع سیستم محافظت کرده و دسترسی بهینه و منصفانه‌تری برای همه کاربران فراهم کنید. با استفاده از این ویژگی، می‌توانید تضمین کنید که سرویس شما همیشه پایدار و در دسترس باقی بماند.

Cache

خیره‌سازی موقت (Caching) در Ocelot به این معناست که پاسخ‌های برخی درخواست‌ها برای مدت محدودی در حافظه نگه داشته می‌شوند. این ویژگی باعث می‌شود که درخواست‌های تکراری نیازی به رسیدن دوباره به سرویس‌های اصلی نداشته باشند و از همان پاسخ‌های ذخیره‌شده استفاده کنند. این کار می‌تواند باعث کاهش بار روی سرورهای اصلی، کاهش زمان پاسخ‌دهی و بهبود عملکرد کلی سیستم شود.

چرا Caching مفید است؟

در بسیاری از موارد، داده‌های برخی درخواست‌ها تا مدتی ثابت باقی می‌مانند (مثل اطلاعات محصولات یا پروفایل کاربران) و نیاز نیست که هر بار این داده‌ها از سرور دریافت شوند. اگر چنین داده‌هایی را در حافظه ذخیره کنیم، درخواست‌های بعدی که به همان داده‌ها نیاز دارند، از کش استفاده کرده و سریع‌تر پاسخ می‌گیرند. به این ترتیب:

- **سرعت پاسخ‌دهی افزایش می‌یابد** چون نیازی به پردازش مجدد داده‌ها نیست.
- **بار روی سرورها کاهش می‌یابد** و منابع سرور برای درخواست‌های جدیدتر آزاد می‌مانند.

- **هزینه‌های منابع کم می‌شود** چون نیاز به پردازش مجدد داده‌ها کاهش می‌یابد.

چطور Caching در Ocelot کار می‌کند؟

در Ocelot می‌توانیم Caching را در فایل پیکربندی ocelot.jsonبرای مسیرهای خاصی فعال کنیم. این تنظیمات مشخص می‌کنند که پاسخ یک درخواست به مدت زمان مشخصی در حافظه باقی بماند. اگر در این مدت درخواست مشابهی بیاید، پاسخ از کش داده می‌شود و نیازی به پردازش مجدد نیست.

یک مثال ساده از تنظیماتCaching

فرض کنید می‌خواهیم برای درخواست‌های مسیر gateway/users/که لیست کاربران را برمی‌گرداند، Cachingرا فعال کنیم و پاسخ‌ها به مدت ۶۰ ثانیه در کش باقی بمانند. تنظیمات زیر را در ocelot.jsonاضافه می‌کنیم:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/users",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5001 }
      ],
      "UpstreamPathTemplate": "/gateway/users",
      "FileCacheOptions": {
```

```
      "TtlSeconds": 60,
      "Region": "UserCache"
    }
  }
 ]
}
```

توضیح بخش‌های مختلف تنظیماتCaching

- **FileCacheOptions**:این بخش برای تنظیمات کش استفاده می‌شود.
- **TtlSeconds**:این گزینه مدت زمان ذخیره‌سازی پاسخ در حافظه (به ثانیه) را مشخص می‌کند. در این مثال، پاسخ‌ها به مدت ۶۰ ثانیه در کش باقی می‌مانند.
- **Region**:این گزینه یک ناحیه برای کش تعیین می‌کند. این ناحیه به ما کمک می‌کند که برای گروه‌های مختلفی از درخواست‌ها از کش‌های مختلف استفاده کنیم و آن‌ها را جداگانه مدیریت کنیم.

چه زمانی از Caching استفاده کنیم؟

Cachingبرای درخواست‌هایی که به دفعات زیاد ارسال می‌شوند و پاسخ آن‌ها تغییرات زیادی ندارد، بسیار مفید است. به عنوان مثال:

- اطلاعات محصولاتی که به ندرت تغییر می‌کنند.
- اطلاعات مربوط به پروفایل‌های کاربری.
- نتایج جستجو یا داده‌های آماری که در طول روز ثابت می‌مانند.

مزایای استفاده ازCaching

- **افزایش سرعت پاسخ‌دهی** :درخواست‌های تکراری سریع‌تر پاسخ داده می‌شوند چون نیازی به دسترسی دوباره به سرور نیست.
- **کاهش بار روی سرورهای اصلی** :چون بسیاری از درخواست‌ها از حافظه پاسخ داده می‌شوند، سرور اصلی برای درخواست‌های جدید آزادتر خواهد بود.
- **صرفه‌جویی در منابع** :با کاهش نیاز به پردازش‌های سنگین و مکرر، مصرف منابع کاهش پیدا می‌کند.

معایب احتمالی

- **عدم بروزرسانی فوری** :اگر داده‌ها به سرعت تغییر کنند، کاربران ممکن است اطلاعات قدیمی را ببینند تا زمانی که کش به‌روزرسانی شود.
- **استفاده از حافظه اضافی** :نگهداری داده‌ها در حافظه ممکن است باعث افزایش مصرف حافظه شود.

خلاصه

Cachingدر Ocelot امکان ذخیره‌سازی موقت پاسخ‌ها را فراهم می‌کند، که به بهبود عملکرد و کاهش بار روی سرورها کمک می‌کند. این ویژگی به ویژه برای داده‌هایی که به طور مکرر درخواست می‌شوند ولی تغییر زیادی ندارند، بسیار مفید است.

(Authentication & Authorization)

تایید هویت و مجوزدهی (Authentication & Authorization) در Ocelot به شما کمک می‌کند که از امنیت API Gateway اطمینان حاصل کنید. این ویژگی به شما امکان می‌دهد تا تنها کاربران معتبر به سرویس‌ها دسترسی داشته باشند و هر کاربر تنها به بخش‌هایی دسترسی پیدا کند که مجاز به استفاده از آن‌ها است.

تفاوت Authentication وAuthorization

1. **Authentication (تایید هویت**: (بررسی می‌کند که کاربر واقعاً همان فردی است که ادعا می‌کند. این مرحله معمولاً با استفاده از اطلاعاتی مانند نام کاربری و رمز عبور یا توکن‌های امنیتی انجام می‌شود.
2. **Authorization (مجوزدهی**: (بعد از تأیید هویت، مشخص می‌کند که کاربر به کدام بخش‌ها و سرویس‌ها دسترسی دارد. این مرحله تعیین می‌کند که کاربر مجاز است چه کارهایی را انجام دهد، مثلاً مشاهده داده‌های خاص، ویرایش، یا حذف آن‌ها.

چطور Authentication و Authorization در Ocelot کار می‌کنند؟

درOcelot ، می‌توانیم از **JWT (JSON Web Tokens)** یا **OAuth2** برای تایید هویت استفاده کنیم. به این صورت که کاربر ابتدا توکن احراز هویت خود را دریافت می‌کند (معمولاً از یک سرور جداگانه برای احراز هویت)، سپس این توکن در هر درخواست به API Gateway ارسال می‌شود Ocelot .نیز با بررسی اعتبار توکن، درخواست کاربر را قبول یا رد می‌کند.

تنظیمات Authentication و Authorization درOcelot

برای پیکربندی این ویژگی‌ها درOcelot ، باید اطلاعات احراز هویت را در فایل ocelot.jsonتنظیم کنیم. در ادامه مثالی از تنظیمات برای یک سرویس با استفاده از JWTآورده شده است:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/users",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5001 }
      ],
      "UpstreamPathTemplate": "/gateway/users",
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "Bearer",
        "AllowedScopes": [ "user.read", "user.write" ]
      }
    }
  ],
```

```
  "GlobalConfiguration": {
    "AuthenticationProviders": [
      {
        "Key": "Bearer",
        "IdentityServerUrl": "https://authserver.com",
        "ApiName": "my_api",
        "RequireHttps": true
      }
    ]
  }
}
```

توضیح بخش‌های مختلف تنظیمات Authentication وAuthorization

- **AuthenticationOptions**:این بخش در سطح هر مسیر تنظیم می‌شود و نشان می‌دهد که برای دسترسی به این مسیر، تایید هویت لازم است.
  - **AuthenticationProviderKey**:نوع ارائه‌دهنده احراز هویت را مشخص می‌کند. در اینجا از) Bearerکه برای توکن‌های JWT استفاده می‌شود (استفاده کرده‌ایم.
  - **AllowedScopes**:مشخص می‌کند که کاربر برای دسترسی به این مسیر چه مجوزهایی (Scopes) نیاز دارد. این Scopes تعیین می‌کنند که کاربر به چه نوع دسترسی‌ها یا عملیاتی مجاز است. به عنوان مثال، user.readبرای خواندن و user.writeبرای نوشتن است.

- **GlobalConfiguration**: این بخش برای تنظیمات عمومی احراز هویت در تمام سرویس‌ها استفاده می‌شود.
  - **AuthenticationProviders**: ارائه‌دهنده‌های تایید هویت را مشخص می‌کند. در اینجا از یک سرور احراز هویت خارجی با URL مشخص استفاده شده است.
  - **IdentityServerUrl**: آدرس سرور احراز هویت را مشخص می‌کند.
  - **ApiName**: نام API برای احراز هویت.

عملکرد این تنظیمات

1. **ارسال درخواست**: کاربر درخواستی را به API Gateway می‌فرستد و توکن احراز هویت JWT خود را در بخش Authorization هدر درخواست قرار می‌دهد.
2. **تایید هویت** Ocelot: توکن را بررسی کرده و در صورت معتبر بودن آن، اجازه دسترسی به درخواست را می‌دهد.
3. **مجوزدهی** Ocelot: سپس بررسی می‌کند که آیا کاربر دارای Scopes لازم برای انجام درخواست است یا خیر. اگر Scopes کاربر با Scopes مورد نیاز مطابقت داشته باشند، درخواست ادامه می‌یابد، وگرنه خطای عدم دسترسی ارسال می‌شود.

مزایای استفاده از Authentication و Authorization

- **امنیت بیشتر**: فقط کاربران معتبر می‌توانند به سیستم دسترسی پیدا کنند، و امکان دسترسی افراد غیرمجاز کاهش می‌یابد.
- **دسترسی محدود و مدیریت شده**: به هر کاربر فقط مجوزهای مورد نیازش داده می‌شود و کاربران نمی‌توانند از منابعی که برایشان تعریف نشده‌اند استفاده کنند.

- **انعطاف‌پذیری در مدیریت دسترسی‌ها** :می‌توان برای هر سرویس و هر مسیر به صورت جداگانه Scopes تعریف کرد تا کاربران مجوزهای متفاوتی داشته باشند.

خلاصه

Authenticationو Authorization در Ocelot به شما این امکان را می‌دهد که دسترسی به سرویس‌های خود را مدیریت کنید و مطمئن شوید که فقط کاربران معتبر و مجاز می‌توانند از امکانات سیستم استفاده کنند. این قابلیت‌ها به خصوص در پروژه‌هایی که نیاز به امنیت بالا دارند و داده‌های حساس دارند، بسیار مفید و حیاتی است.

(Request & Response Transformation)

تبدیل درخواست و پاسخ (Request & Response Transformation) در Ocelot به شما این امکان را می‌دهد که محتوا و ساختار درخواست‌ها و پاسخ‌ها را قبل از ارسال به سرویس‌های پشت‌زمینه (Downstream) یا پس از دریافت پاسخ از آن‌ها، تغییر دهید. این ویژگی برای هماهنگ‌سازی API Gateway با فرمت‌های مختلف و تطبیق درخواست‌ها و پاسخ‌ها با نیازهای سرویس‌های مختلف مفید است.

چرا Request & Response Transformation اهمیت دارد؟

در پروژه‌های بزرگ، ممکن است با سرویس‌های مختلفی روبرو شوید که هرکدام فرمت داده‌ها و ساختار درخواست‌ها و پاسخ‌های مخصوص به خود را دارند. مثلاً یک سرویس ممکن است داده‌ها را به صورت JSON دریافت کند و دیگری به فرمت XML نیاز داشته باشد، یا شاید برخی سرویس‌ها به پارامترهای خاصی در URL نیاز دارند که در

درخواست اولیه وجود ندارند. با استفاده از ویژگی Transformation در Ocelot می‌توانید درخواست‌ها و پاسخ‌ها را مطابق نیاز هر سرویس تبدیل کنید.

چطور Request & Response Transformation در Ocelot کار می‌کند؟

Ocelotامکان تعریف تبدیل‌های مختلف را در فایل ocelot.jsonفراهم می‌کند. می‌توانید تعیین کنید که هدرها، پارامترهای مسیر، یا بدنه درخواست‌ها و پاسخ‌ها چگونه تغییر کنند.

یک مثال ازRequest Transformation

فرض کنید یک سرویس داریم که انتظار دارد درخواستی شامل پارامتر userIdدر مسیر (/api/users/{userId}) باشد. در سمت کلاینت، درخواست‌ها بدون این پارامتر ارسال می‌شوند. می‌توانیم با استفاده ازRequest Transformation ، پارامتر را در مسیر اضافه کنیم.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/users/{userId}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5001 }
      ],
      "UpstreamPathTemplate": "/gateway/users",
```

```
    "RequestParameters": {
      "Add": {
        "userId": "{Claims[user_id]}"
      }
    }
  }
 ]
}
```

توضیح بخش‌های تنظیمات بالا

- **DownstreamPathTemplate**: مسیر درخواست در سرویس پشت‌زمینه را مشخص می‌کند {userId} .به عنوان پارامتر مسیر قرار داده شده است.
- **UpstreamPathTemplate**: مسیر اصلی که کاربران استفاده می‌کنند.
- **RequestParameters**: این بخش به شما اجازه می‌دهد پارامترهای مورد نیاز را در درخواست اضافه کنید.
    - **Add**: در اینجا پارامتر userId از اطلاعات مربوط به ادعای کاربر (Claims) گرفته و به مسیر اضافه می‌شود.

یک مثال از Response Transformation

فرض کنید پاسخ سرویس شامل داده‌هایی است که کلاینت به همه آن‌ها نیاز ندارد، و فقط یک بخش خاص از آن مورد نظر است. می‌توانیم با استفاده از Response Transformation، تنها بخشی از پاسخ را برگردانیم.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/products",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5002 }
      ],
      "UpstreamPathTemplate": "/gateway/products",
      "DownstreamResponseHeadersTransform": {
        "Remove": ["Server"]
      },
      "ResponseCondition": "productId > 100",
      "ResponseTemplate": "{ 'ProductName': '{DownstreamResponseBody.ProductName}', 'Price': '{DownstreamResponseBody.Price}' }"
```

```
    }
  ]
}
```

توضیح بخش‌های مختلف درResponse Transformation

- **DownstreamResponseHeadersTransform**:تنظیماتی برای تغییر هدرهای پاسخ است. در این مثال، هدر Serverحذف می‌شود.
- **ResponseTemplate**:این قالب مشخص می‌کند که چه بخش‌هایی از پاسخ اصلی در پاسخ نهایی قرار بگیرند. در اینجا، فقط ProductNameو Priceاز پاسخ اصلی برگردانده می‌شوند.
- **ResponseCondition**:شرطی است که باید برای اجرای Transformation برقرار باشد. در این مثال، این تبدیل تنها برای محصولاتی با productIdبیشتر از 100 اعمال می‌شود.

مزایای استفاده ازRequest & Response Transformation

- **هماهنگی با سرویس‌های مختلف** :به راحتی می‌توان درخواست‌ها را برای تطبیق با فرمت‌های مختلف تنظیم کرد.
- **بهبود امنیت** :می‌توان هدرها یا بخش‌هایی از پاسخ را حذف کرد تا اطلاعات غیرضروری به کلاینت باز نگردد.
- **ساده‌سازی داده‌ها** :تنها داده‌های ضروری به کلاینت باز می‌گردند و از ارسال داده‌های اضافی جلوگیری می‌شود.

موارد کاربردRequest & Response Transformation

1. **تطبیق فرمت داده‌ها** :مثلاً تبدیل JSON به XML و بالعکس.

2. **افزودن یا حذف هدرها** :برای مطابقت با نیازهای امنیتی یا تنظیمات خاص.
3. **تغییر ساختار بدنه درخواست یا پاسخ** :مثل اضافه کردن پارامترها، فیلتر کردن داده‌ها و یا اصلاح آن‌ها.
4. **مدیریت داده‌های حساس** :حذف اطلاعات غیرضروری یا خصوصی از پاسخ‌ها.

خلاصه

ویژگی Request & Response Transformation در Ocelot به شما امکان می‌دهد که درخواست‌ها و پاسخ‌ها را طبق نیاز سرویس‌ها و کلاینت‌ها تنظیم کنید. این ویژگی به ویژه در سناریوهای یکپارچه‌سازی سیستم‌ها، بهبود امنیت و ساده‌سازی داده‌ها بسیار کاربردی است.

(Support for different protocols)

Ocelotاز پروتکل‌های مختلف پشتیبانی می‌کند تا بتواند با انواع سرویس‌های مختلف و تکنولوژی‌های موجود ارتباط برقرار کند. این ویژگی به شما امکان می‌دهد که API Gatewayخود را به راحتی با سرویس‌هایی که از پروتکل‌های متفاوتی استفاده می‌کنند، یکپارچه کنید.

چرا پشتیبانی از پروتکل‌های مختلف اهمیت دارد؟

در پروژه‌های بزرگ و پیچیده، احتمال زیادی وجود دارد که سرویس‌های مختلف از پروتکل‌های متفاوتی استفاده کنند. مثلاً برخی سرویس‌ها با HTTP کار می‌کنند، در حالی که سرویس‌های دیگر ممکن است به پروتکل‌های گوناگون مانندHTTPS ، WebSocketیا gRPC نیاز داشته باشند API Gateway .اگر از پروتکل‌های مختلف

پشتیبانی کند، می‌تواند درخواست‌ها را برای هر سرویس به پروتکل مناسب هدایت کند و همه این سرویس‌ها را در یک نقطه (Gateway) مدیریت کند.

پروتکل‌های پشتیبانی‌شده درOcelot

Ocelotبه طور پیش‌فرض از پروتکل‌های مختلفی پشتیبانی می‌کند که از جمله آن‌ها می‌توان به موارد زیر اشاره کرد:

1. **HTTPو HTTPS**: این دو پروتکل رایج‌ترین پروتکل‌های وب هستند Ocelot .به خوبی از هر دو پشتیبانی می‌کند و می‌توان به سادگی درخواست‌ها را به هر دو پروتکل هدایت کرد. در فایل تنظیمات ocelot.json، می‌توان با مشخص کردن DownstreamSchemeبین HTTP و HTTPS جابجا شد.
2. **WebSocket**:برای پشتیبانی از ارتباط‌های بلادرنگ (Real-time) می‌توان از WebSocketاستفاده کرد. این پروتکل برای ارتباطات دوطرفه کاربرد دارد و Ocelotمی‌تواند درخواست‌های WebSocket را به درستی به سرویس‌های مربوطه ارسال کند.
3. gRPC: **gRPC**یک فریم‌ورک Remote Procedure Call (RPC) است که توسط گوگل توسعه یافته است و معمولاً برای سیستم‌های توزیع‌شده استفاده می‌شود Ocelot .قابلیت مسیردهی gRPC را فراهم می‌کند و می‌توانید درخواست‌های gRPC را از طریق Gateway به سرویس‌های پشتیبان ارسال کنید.

تنظیمات Ocelot برای پروتکل‌های مختلف

درOcelot ، می‌توانیم نوع پروتکل هر سرویس را به سادگی در فایل پیکربندی ocelot.jsonمشخص کنیم.

نمونه تنظیمات برای HTTP وHTTPS

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/users",
      "DownstreamScheme": "https", // این قسمت مشخص می‌کند که درخواست به سرویس پشت‌زمینه با HTTPS ارسال شود
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5001 }
      ],
      "UpstreamPathTemplate": "/gateway/users"
    }
  ]
}
```

در اینجا، DownstreamSchemeبه صورت httpsتنظیم شده است، به این معنی که درخواست‌ها به سرویس پشت‌زمینه با استفاده از پروتکل HTTPS ارسال می‌شوند.

نمونه تنظیمات برای WebSocket

برای فعال‌سازی پشتیبانی از WebSocket درOcelot ، می‌توانید مسیر را به گونه‌ای تنظیم کنید که درخواست‌ها به WebSocket هدایت شوند:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/ws/notifications",
      "DownstreamScheme": "ws",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 6001 }
      ],
      "UpstreamPathTemplate": "/gateway/notifications"
    }
  ]
}
```

در اینجا، DownstreamSchemeبه صورت wsتنظیم شده است که نشان می‌دهد درخواست به سرویس WebSocket ارسال می‌شود.

نمونه تنظیمات برایgRPC

پیکربندی Ocelot برای gRPC نیز مشابه است، با این تفاوت که gRPC به ارتباطات HTTP/2نیاز دارد. در مثال زیر، درخواست‌های gateway/orders/به یک سرویس gRPCدر پورت 7001 ارسال می‌شود.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/grpc/orders",
      "DownstreamScheme": "http", // برای gRPC پروتکل HTTP/2 استفاده می‌شود
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 7001 }
      ],
      "UpstreamPathTemplate": "/gateway/orders",
      "Grpc": true
    }
  ]
}
```

در اینجا با تنظیم Grpc: true مشخص می‌کنیم که این مسیر باید از gRPC پشتیبانی کند.

مزایای پشتیبانی از پروتکل‌های مختلف

- **انعطاف‌پذیری بالا** :می‌توانید با سرویس‌های مختلف و با پروتکل‌های متنوع به راحتی ارتباط برقرار کنید.

- **هماهنگی سرویس‌ها در یک نقطه** :به جای اینکه کلاینت‌ها با سرویس‌های مختلف و پروتکل‌های متنوع مستقیم ارتباط برقرار کنند، می‌توانند به Gatewayمتصل شده و Gateway وظیفه تطبیق پروتکل را بر عهده گیرد.
- **ساده‌سازی مدیریت امنیت** :می‌توان به سادگی از HTTPS برای تمام ارتباطات استفاده کرد یا امنیت WebSocket را از طریق Gateway تنظیم نمود.

خلاصه

پشتیبانی از پروتکل‌های مختلف در Ocelot به شما اجازه می‌دهد که درخواست‌های کلاینت‌ها را با پروتکل‌های مختلف، از جملهHTTP ، HTTPS، WebSocketوgRPC ، به سرویس‌های پشت‌زمینه ارسال کنید. این ویژگی به ویژه در سیستم‌های توزیع‌شده که نیاز به ارتباطات بلادرنگ و پروتکل‌های خاص دارند، بسیار کاربردی است.

مسیرهای دینامیک (Dynamic Routing) در Ocelot این امکان را فراهم می‌کند که مسیریابی به صورت پویا و خودکار انجام شود. این ویژگی زمانی مفید است که ساختار یا مسیر سرویس‌ها در حال تغییر باشد یا به تعداد زیادی سرویس با پیکربندی متفاوت نیاز داشته باشیم. در چنین شرایطی، به جای تغییر دستی و مداوم تنظیمات، Ocelotبه کمک مسیرهای دینامیک می‌تواند درخواست‌ها را بر اساس الگوها یا قوانین خاصی مسیردهی کند.

کاربردهای مسیرهای دینامیک

مسیرهای دینامیک برای پروژه‌هایی مفید است که در آن‌ها:

- **سرویس‌های جدید به صورت خودکار ایجاد می‌شوند**، مثلاً در معماری‌های میکروسرویسی که سرویس‌ها ممکن است پویا به وجود آیند یا حذف شوند.

- **سرویس‌ها به صورت مداوم آپدیت یا جا به جا می‌شوند** و نمی‌خواهیم به صورت دستی فایل‌های پیکربندی را تغییر دهیم.
- **نیاز به مسیردهی بر اساس شرایط خاص داریم**، مثلاً مسیردهی بر اساس محتوای هدر، شناسه‌ی کاربر یا نسخه.API

چطور Dynamic Routing در Ocelot کار می‌کند؟

Ocelotبه شما امکان می‌دهد که مسیریابی را بر اساس پارامترهای مسیر، هدرها، QueryString، یا داده‌های دیگری به صورت پویا تنظیم کنید. به این ترتیب، می‌توانید به جای تعریف مسیرهای ثابت، از الگوهای دینامیک استفاده کنید.

یک مثال ساده از مسیر دینامیک

فرض کنید می‌خواهیم تمام درخواست‌هایی که به {id}/{service}/gateway/می‌آیند، به سرویس‌های مربوطه هدایت کنیم. برای مثال، اگر {service}برابر با usersو {id} برابر با 123باشد، درخواست به api/users/123/هدایت شود.

در فایل ocelot.jsonمی‌توانیم از الگوی زیر استفاده کنیم:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/{service}/{id}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5001 }
      ],
```

```
      "UpstreamPathTemplate": "/gateway/{service}/{id}"
    }
  ]
}
```

توضیح تنظیمات بالا

- **DownstreamPathTemplate**: مسیر مقصد در سرویس پشت‌زمینه است که {service}و {id}به صورت دینامیک از مسیر درخواستی گرفته می‌شوند.
- **UpstreamPathTemplate**: مسیر ورودی به Gateway است که الگوی دینامیک {service}و {id}را شامل می‌شود.

این الگو باعث می‌شود که درخواست‌ها به gateway/users/123/به طور خودکار به api/users/123/هدایت شوند، بدون نیاز به تعریف مسیرهای متعدد برای هر سرویس و شناسه.

مثال پیشرفته‌تر: مسیر دینامیک با استفاده ازQueryString

فرض کنید می‌خواهیم بر اساس نسخهAPI ، مسیرها را به سرویس‌های متفاوتی هدایت کنیم. مثلاً درخواست‌هایی با نسخه v1به یک سرویس و درخواست‌های با نسخه v2به سرویس دیگری هدایت شوند. این کار را می‌توان با بررسی QueryString انجام داد.

```
{
  "Routes": [
```

```
{
  "DownstreamPathTemplate": "/api/v1/users",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 5001 }
  ],
  "UpstreamPathTemplate": "/gateway/users",
  "UpstreamHttpMethod": [ "Get" ],
  "RouteIsCaseSensitive": false,
  "AddQueriesToRequest": {
    "version": "v1"
  }
},
{
  "DownstreamPathTemplate": "/api/v2/users",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 5002 }
  ],
```

```
    "UpstreamPathTemplate": "/gateway/users",
    "UpstreamHttpMethod": [ "Get" ],
    "RouteIsCaseSensitive": false,
    "AddQueriesToRequest": {
      "version": "v2"
    }
  }
 ]
}
```

در اینجا، درخواست‌هایی که دارای نسخه v1 هستند به پورت 5001 هدایت می‌شوند و درخواست‌های v2 به پورت 5002، که به این ترتیب می‌توانیم نسخه‌های مختلف API را مدیریت کنیم.

مزایای استفاده ازDynamic Routing

- **انعطاف‌پذیری بالا** :مسیرهای دینامیک این امکان را می‌دهند که بدون نیاز به تغییرات مکرر، به سرویس‌های جدید یا سرویس‌های تغییر یافته دسترسی داشته باشید.
- **پشتیبانی از توسعه‌پذیری** :می‌توانید به راحتی سرویس‌های جدید را اضافه کرده یا مسیرهای موجود را به‌روزرسانی کنید.
- **کاهش پیچیدگی مدیریت** :دیگر نیازی به ایجاد مسیرهای متعدد و ایستا برای هر سرویس نیست؛ الگوهای دینامیک همه چیز را مدیریت می‌کنند.

خلاصه

Dynamic Routingدر Ocelot قابلیت مسیردهی پویا را فراهم می‌کند که در پروژه‌های بزرگ و سیستم‌های میکروسرویسی بسیار کاربردی است. با این ویژگی، Ocelotبه شما امکان می‌دهد که مسیرها را بر اساس الگوهای مختلف تنظیم کنید و به راحتی با تغییرات و افزودنی‌های جدید در سیستم خود سازگار شوید.

مانیتورینگ و گزارش‌گیری(Logging & Monitoring)

مانیتورینگ و گزارش‌گیری (Logging & Monitoring) در Ocelot یکی از قابلیت‌های مهم برای پایش و بررسی عملکرد API Gateway است. این ویژگی به توسعه‌دهندگان و تیم‌های عملیات کمک می‌کند که مشکلات احتمالی را سریع شناسایی کنند، سلامت سیستم را بررسی کنند و عملکرد درخواست‌ها را تحلیل کنند.

اهمیت Logging وMonitoring

در یک سیستم مبتنی بر میکروسرویس، درخواست‌ها از میان چندین سرویس عبور می‌کنند و API Gateway نقشی کلیدی در این ارتباط‌ها دارد. اگر Gateway به درستی پایش نشود، مشکلات ممکن است به موقع شناسایی نشوند و بر عملکرد کلی سیستم تاثیر بگذارند Logging .و Monitoring به تیم‌ها کمک می‌کنند:

1. **مشکلات و خطاها را سریع‌تر شناسایی کنند.**
2. **عملکرد سیستم را بررسی کنند** و گلوگاه‌های احتمالی را شناسایی کنند.
3. **امنیت سیستم را بالا ببرند**، چون با بررسی گزارش‌ها می‌توان درخواست‌های مشکوک یا حملات را شناسایی کرد.

امکانات Logging و Monitoring درOcelot

Ocelotبه طور پیش‌فرض با برخی از ابزارهای لاگ‌گیری و مانیتورینگ سازگار است و می‌تواند با کتابخانه‌های متداول دات‌نت مانند **Serilog**و **Microsoft.Extensions.Logging**استفاده شود. علاوه بر این، به راحتی می‌توان آن را با ابزارهای مانیتورینگ خارجی مثل **Prometheus**و **Grafana**یکپارچه کرد.

چطور Logging را در Ocelot تنظیم کنیم؟

برای ثبت لاگ درOcelot ، می‌توان از سیستم لاگ‌گیری دات‌نت استفاده کرد و تنظیمات مربوطه را در فایل پیکربندی اعمال کرد. در اینجا مثالی از تنظیم Serilog برای لاگ‌گیری آورده شده است:

1. **نصب :Serilog** ابتدا کتابخانه Serilog را به پروژه اضافه کنید:

```
dotnet add package Serilog.AspNetCore
```

تنظیم Serilog در Program.cs:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
```

```
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        })
        .UseSerilog((context, configuration) =>
        {
            configuration
                .MinimumLevel.Debug()
                .Enrich.FromLogContext()
                .WriteTo.Console()
                .WriteTo.File("logs/log.txt", rollingInterval: RollingInterval.Day);
        });
}
```

1. **فعال‌سازی لاگ‌ها در Ocelot**: Ocelot از لاگ‌های ثبت‌شده برای پیگیری درخواست‌ها و خطاها استفاده می‌کند. با این کار، لاگ‌های مربوط به هر درخواست ثبت می‌شوند که شامل جزئیاتی مانند آدرس درخواست، هدرها، زمان پاسخ‌دهی و خطاها است.

یکپارچه‌سازی با ابزارهای مانیتورینگ

برای مانیتورینگ کامل، معمولاً از ابزارهایی مانند **Prometheus** و **Grafana** استفاده می‌شود. Prometheus داده‌های مربوط به عملکرد سیستم را جمع‌آوری می‌کند و

Grafanaبه شما امکان می‌دهد این داده‌ها را به صورت نمودارهای گرافیکی مشاهده کنید.

1. **اضافه کردن Prometheus Middleware**: با نصب کتابخانه‌های لازم، می‌توانید داده‌های Prometheus را در پروژه Ocelot دریافت کنید.
2. **تنظیمات Exporter برای Prometheus**: Prometheus ازExporter ها برای جمع‌آوری داده‌های مانیتورینگ استفاده می‌کند. شما می‌توانید ازExporter های مربوط به دات‌نت برای این کار استفاده کنید.
3. **ایجاد داشبورد در Grafana**: پس از جمع‌آوری داده‌ها، از طریق Grafana داشبوردهای مانیتورینگ ایجاد می‌کنید که شامل معیارهایی مانند تعداد درخواست‌ها، میزان تاخیر(Latency) ، درصد خطاها، و میانگین زمان پاسخ‌دهی (Response Time) است.

مزایای Logging و Monitoring درOcelot

- **شناسایی خطاها**: خطاها و هشدارها به صورت لاگ ثبت می‌شوند که به رفع سریع مشکلات کمک می‌کند.
- **ارزیابی عملکرد**: بررسی زمان‌های پاسخ‌دهی و تعداد درخواست‌ها به شما کمک می‌کند تا عملکرد Gateway و سرویس‌ها را ارزیابی کنید.
- **بهبود امنیت**: با مشاهده لاگ‌ها و الگوهای درخواست‌ها، می‌توان فعالیت‌های مشکوک را شناسایی کرد و اقدامات امنیتی را بهبود داد.
- **افزایش بهره‌وری**: با مانیتورینگ دقیق، تیم‌ها می‌توانند گلوگاه‌های موجود را شناسایی کرده و کارایی سیستم را بهبود بخشند.

موارد کاربرد Logging وMonitoring

1. **پایش سلامت سیستم**: اطمینان از این که Gateway به درستی کار می‌کند.

2. **ردیابی درخواست‌ها** :پیگیری درخواست‌ها و مسیر طی‌شده توسط آن‌ها برای عیب‌یابی سریع.
3. **مدیریت منابع** :اطمینان از اینکه منابع Gateway بهینه استفاده می‌شوند.
4. **مدیریت ترافیک** :بررسی تعداد درخواست‌ها در زمان‌های مختلف برای ارزیابی ترافیک و تنظیمات مربوط به مقیاس‌پذیری.

خلاصه

Logging و Monitoring در Ocelot به شما امکان می‌دهد که بر عملکرد API Gateway نظارت کنید، مشکلات را شناسایی و بهینه‌سازی‌های لازم را اعمال کنید. با استفاده از ابزارهایی مانند Serilog برای لاگ‌گیری و Prometheus و Grafana برای مانیتورینگ، می‌توانید سیستم خود را با دقت و کارایی بیشتری مدیریت کنید.

تجمیع درخواست‌ها(Request Aggregation)

تجمیع درخواست‌ها (Request Aggregation) در Ocelot قابلیتی است که به کمک آن می‌توان چندین درخواست به سرویس‌های مختلف را در یک درخواست واحد تجمیع کرد و به کلاینت ارسال کرد. این ویژگی زمانی مفید است که نیاز داشته باشیم داده‌های مختلف از چند سرویس را در یک پاسخ واحد ارائه دهیم، به جای آن که کلاینت چندین بار به API Gateway درخواست بدهد.

کاربردهای Request Aggregation

Request Aggregation برای ساده‌سازی ارتباط بین کلاینت و سرویس‌های پشت‌زمینه کاربرد دارد. برخی از سناریوهای رایج شامل موارد زیر است:

1. **دریافت اطلاعات ترکیبی** :مثلاً در یک سیستم فروشگاهی، ممکن است بخواهید اطلاعات محصول، قیمت و نظرات کاربران را در یک درخواست دریافت کنید.

2. **کاهش تعداد درخواست‌ها** :به جای اینکه کلاینت مجبور باشد چندین درخواست جداگانه ارسال کند و منتظر دریافت پاسخ‌های جداگانه باشد، Gateway درخواست‌ها را به سرویس‌های مختلف می‌فرستد و پاسخ ترکیبی را برمی‌گرداند.
3. **بهبود کارایی و کاهش تأخیر (Latency)**: چون تجمیع درخواست‌ها به جای سمت کلاینت در سمت Gateway انجام می‌شود، تاخیر شبکه و زمان انتظار برای کلاینت کاهش می‌یابد.

چطور Request Aggregation را در Ocelot تنظیم کنیم؟

در Ocelot می‌توانیم تجمیع درخواست‌ها را در فایل ocelot.jsonپیکربندی کنیم. به طور کلی، در این فایل مشخص می‌کنیم که کدام درخواست‌ها به چه سرویس‌هایی ارسال شوند و چگونه نتیجه‌ها تجمیع شوند.

مثال تنظیمات Request Aggregation در ocelot.json

فرض کنید می‌خواهیم اطلاعات یک محصول و نظرات آن محصول را در یک درخواست تجمیع کنیم. این اطلاعات از دو سرویس متفاوت به دست می‌آیند: یکی برای اطلاعات محصول و دیگری برای نظرات.

```
{
  "Routes": [
    {
      "UpstreamPathTemplate": "/gateway/productdetails/{productId}",
      "DownstreamPathTemplates": [
        "/api/products/{productId}",
        "/api/reviews/{productId}"
```

```
      ],
      "UpstreamHttpMethod": [ "Get" ],
      "Aggregator": "ProductReviewsAggregator"
    }
  ],
  "Aggregates": [
    {
      "RouteKeys": [
        "/api/products/{productId}",
        "/api/reviews/{productId}"
      ],
      "Aggregator": "ProductReviewsAggregator"
    }
  ]
}
```

در اینجا تنظیمات فوق به موارد زیر اشاره دارند:

- **UpstreamPathTemplate**: مسیر درخواستی که کلاینت به Gateway می‌فرستد، مثلاً /gateway/productdetails/123.

- **DownstreamPathTemplates**: مسیرهایی که API Gateway به صورت همزمان درخواست‌ها را به آن‌ها می‌فرستد. در این مثال، درخواست‌ها به دو سرویس /api/products/123 و /api/reviews/123 ارسال می‌شوند.
- **Aggregator**: نام کلاس تجمیع‌کننده‌ای که مشخص می‌کند پاسخ‌های دریافتی چگونه ترکیب شوند.

نوشتن کلاس Aggregator

در مثال بالا، ProductReviewsAggregator نام کلاسی است که مسئول تجمیع پاسخ‌های دو سرویس می‌باشد. برای ایجاد این کلاس، باید یک کلاس سفارشی تجمیع‌کننده بنویسیم که از IDefinedAggregator در Ocelot پیروی کند.

نمونه‌ای از کد کلاسAggregator

```
using System.Threading.Tasks;
using Ocelot.Middleware;
using Ocelot.Multiplexer;

public class ProductReviewsAggregator : IDefinedAggregator
{
    public async Task<DownstreamResponse> Aggregate(List<HttpContext> responses)
    {
        var productResponse = await responses[0].Items.DownstreamResponse().Content.ReadAsStringAsync();
        var reviewsResponse = await responses[1].Items.DownstreamResponse().Content.ReadAsStringAsync();

        // ترکیب دو پاسخ
        var combinedResponse = $"{{\"product\": {productResponse}, \"reviews\": {reviewsResponse}}}";

        return new DownstreamResponse(
```

```
            new StringContent(combinedResponse),
            HttpStatusCode.OK,
            responses.SelectMany(r =>
r.Items.DownstreamResponse().Headers).ToList(),
            "application/json");
    }
}
```

توضیح کد

در اینجا، کلاس ProductReviewsAggregator داده‌های پاسخ دو سرویس را می‌گیرد و آن‌ها را در یک شیء JSON ترکیب می‌کند. نتیجه نهایی شامل داده‌های محصول و نظرات است و به صورت یک پاسخ واحد به کلاینت بازگردانده می‌شود.

مزایای استفاده ازRequest Aggregation

- **کاهش تعداد تماس‌ها از سمت کلاینت** :کلاینت تنها یک درخواست می‌فرستد و پاسخ ترکیبی دریافت می‌کند.
- **بهبود عملکرد کلاینت** :کلاینت به جای مدیریت چندین درخواست و انتظار برای هر کدام، به یک پاسخ واحد دسترسی پیدا می‌کند.
- **بهینه‌سازی کارایی سیستم** :کاهش تعداد تماس‌های شبکه و پاسخ‌های مستقل به کاهش تاخیر و افزایش کارایی سیستم کمک می‌کند.

خلاصه

Request Aggregationدر Ocelot به شما امکان می‌دهد چندین درخواست را در یک پاسخ واحد تجمیع کرده و به کلاینت بازگردانید. این ویژگی در سیستم‌های

میکروسرویسی کاربرد فراوان دارد، چرا که ارتباطات کلاینت و سرویس‌ها را ساده می‌کند و کارایی را افزایش می‌دهد. با نوشتن کلاس‌های Aggregator سفارشی، می‌توانید درخواست‌ها را به روش‌های مختلف ترکیب و پاسخ‌های موردنیاز خود را از API Gatewayارائه کنید.

پشتیبانی از Canary Testing یکی از ویژگی‌های مهم درAPI Gateway ها مانند **Ocelot**است که به شما امکان می‌دهد تغییرات و به‌روزرسانی‌های جدید سرویس‌ها را به طور تدریجی و ایمن به کاربران عرضه کنید. درCanary Testing ، ابتدا یک نسخه جدید از سرویس فقط برای بخشی از کاربران یا ترافیک تست شده منتشر می‌شود و پس از تأیید موفقیت‌آمیز، به تدریج به سایر کاربران نیز عرضه می‌شود. این روش به شناسایی مشکلات احتمالی قبل از انتشار کامل کمک می‌کند.

مفهومCanary Testing

اصطلاح "Canary" از تاریخچه معدن‌کاری گرفته شده است. در گذشته، معدن‌کاران از پرندگان قناری در معادن استفاده می‌کردند تا در صورت وجود گازهای مضر، پرنده زودتر از انسان‌ها علائم مسمومیت را نشان دهد. در دنیای نرم‌افزار، Canary Testing به این صورت عمل می‌کند که تغییرات یا نسخه‌های جدید یک سرویس ابتدا در یک بخش کوچک از ترافیک یا کاربران منتشر می‌شود، و سپس بر اساس نتایج این آزمایش، به تدریج برای بقیه کاربران فعال می‌شود.

پشتیبانی Canary Testing درOcelot

Ocelotاز Canary Testing به کمک ویژگی‌های پیکربندی مخصوصی که در فایل ocelot.jsonتعریف می‌شود، پشتیبانی می‌کند. این ویژگی به شما این امکان را می‌دهد که درخواست‌ها را به نسخه‌های مختلف سرویس (مانند نسخه قدیمی و جدید) تقسیم کرده و به طور همزمان چندین نسخه از سرویس را آزمایش کنید.

چطور Canary Testing را در Ocelot پیکربندی کنیم؟

برای پیاده‌سازی Canary Testing درOcelot ، باید درخواست‌ها را بین چندین نسخه از سرویس تقسیم کنید و مشخص کنید که هر نسخه به چه میزان از ترافیک دسترسی خواهد داشت.

1.تنظیمات اولیه

فرض کنید می‌خواهید تغییرات جدید را به تدریج بر روی سرویس "ProductService" پیاده‌سازی کنید. ابتدا باید دو نسخه مختلف از سرویس ایجاد کرده باشید: یک نسخه قدیمی و یک نسخه جدید. سپس درخواست‌ها را به این دو نسخه تقسیم کنید.

در اینجا یک مثال از نحوه پیکربندی Canary Testing در Ocelot آورده شده است:

```
{
  "Routes": [
    {
      "UpstreamPathTemplate": "/gateway/products",
      "DownstreamPathTemplate": "/api/products",
      "UpstreamHttpMethod": [ "Get" ],
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5001 }
      ],
      "RouteIsCaseSensitive": false,
      "Priority": 1
```

```
    },
    {
      "UpstreamPathTemplate": "/gateway/products",
      "DownstreamPathTemplate": "/api/products",
      "UpstreamHttpMethod": [ "Get" ],
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 5002 }
      ],
      "RouteIsCaseSensitive": false,
      "Priority": 2
    }
  ]
}
```

2.تقسیم ترافیک

در مثال بالا، درخواست‌هایی که به gateway/products/می‌آیند به دو نسخه مختلف از سرویس "ProductService" ارسال می‌شوند: یکی به پورت 5001 و دیگری به پورت 5002. به کمک **Priority**، می‌توانیم کنترل کنیم که چه میزان ترافیک به هر نسخه ارسال شود.

- **Priority**:اولویت مسیریابی را مشخص می‌کند. به عنوان مثال، با قرار دادن یک سرویس در اولویت بیشتر، می‌توانید ترافیک بیشتری را به آن سرویس هدایت کنید.
- **DownstreamHostAndPorts**:این تنظیمات مشخص می‌کند که درخواست‌ها به کدام سرویس (نسخه قدیمی یا جدید) ارسال شوند.

.3پیکربندی تقسیم ترافیک به صورت تدریجی

برای انجامCanary Testing ، معمولاً می‌خواهیم ابتدا درصد کوچکی از ترافیک را به نسخه جدید ارسال کنیم و سپس به تدریج این درصد را افزایش دهیم Ocelot .به طور مستقیم ابزارهای پیچیده تقسیم ترافیک مانند درصد را ارائه نمی‌دهد، اما می‌توان این کار را با استفاده از فیلترهای سفارشی یا با تقسیم ترافیک بر اساس شرایط خاص انجام داد.

برای مثال، می‌توانید از **Rate Limiting**یا **Header-based routing**استفاده کنید تا کنترل بیشتری بر نحوه تقسیم ترافیک داشته باشید.

مزایای Canary Testing درOcelot

- **کاهش ریسک** :با تست نسخه‌های جدید تنها برای بخش کوچکی از کاربران، خطر بروز مشکلات بزرگ کاهش می‌یابد.
- **آزمایش تغییرات به صورت تدریجی** :می‌توانید تغییرات خود را در محیط واقعی با کمترین تأثیر بر کاربران امتحان کنید.
- **بازخورد سریع‌تر** :با استفاده ازCanary Testing ، می‌توانید سریع‌تر مشکلات احتمالی را شناسایی کنید و قبل از اینکه تغییرات به تمامی کاربران منتشر شوند، اصلاحات لازم را اعمال کنید.

یکپارچگی با ابزارهای خارجی

اگر بخواهید از Canary Testing به صورت حرفه‌ای‌تر و با ابزارهای بیشتری نظارت کنید، می‌توانید Ocelot را با ابزارهایی مانند **Kubernetes**یا **Istio**ترکیب کنید که امکانات پیشرفته‌تری برای تقسیم ترافیک و Canary Testing فراهم می‌کنند.

خلاصه

Canary Testingدر Ocelot به شما این امکان را می‌دهد که تغییرات و نسخه‌های جدید سرویس‌ها را به صورت تدریجی و با خطر کمتر برای کاربران عرضه کنید. با استفاده از ویژگی‌های مسیریابی و اولویت‌ها درOcelot ، می‌توانید ترافیک را بین نسخه‌های مختلف سرویس‌ها تقسیم کرده و نتایج را ارزیابی کنید. این ویژگی باعث کاهش ریسک‌ها و مشکلات ناشی از انتشار تغییرات بزرگ می‌شود و به تیم‌ها کمک می‌کند که تغییرات را با اطمینان بیشتری پیاده‌سازی کنند.