.NET Microservices – Azure DevOps and AKS

Section 10: API Gateway using Ocelot - Notes

Introduction to API Gateway

An API Gateway is a server that acts as an API front-end, receiving API requests, routing them to the appropriate microservice, and then returning the responses back to the client. It sits between the client and your backend services, providing a single entry point for client requests.

Functions of an API Gateway:

1. **Request Routing**: Directs incoming requests to the appropriate microservice.
2. **Load Balancing**: Distributes incoming traffic evenly across multiple instances of a service.
3. **Authentication and Authorization**: Manages security by authenticating and authorizing requests.
4. **Aggregation**: Combines responses from multiple microservices into a single response for the client.
5. **Rate Limiting**: Controls the rate of requests to prevent overloading services.
6. **Caching**: Improves performance by caching frequently requested data.
7. **Logging and Monitoring**: Tracks and logs requests for monitoring and debugging purposes.
8. **Transformation**: Modifies requests and responses, such as adding headers or transforming data formats.

Why Use an API Gateway?

- **Simplifies Client Communication**: Clients make requests to a single gateway instead of multiple services.
- **Centralized Management**: Provides a central point for managing cross-cutting concerns like security, logging, and monitoring.
- **Improves Security**: Handles authentication, authorization, and SSL termination.
- **Optimizes Performance**: Supports caching and load balancing to improve response times and availability.
- **Facilitates Service Composition**: Aggregates results from multiple services into a single response.

Common API Gateway Patterns:

1. **Proxy Pattern**: The gateway simply forwards requests to the backend services.
2. **Aggregator Pattern**: Combines responses from multiple services into one response.
3. **Routing Pattern**: Routes requests based on URL paths, HTTP methods, or headers.
4. **Security Pattern**: Handles authentication, authorization, and SSL termination.

Introduction to Ocelot

Ocelot is a lightweight API Gateway built for .NET applications. It simplifies the management of microservices by providing features such as routing, load balancing, caching, and policy handling. Ocelot is particularly well-suited for environments where multiple services need to be exposed through a single entry point.

Key Features of Ocelot

1. **Routing**: Ocelot maps incoming HTTP requests to appropriate downstream services, making it easier to manage and direct traffic.
2. **Load Balancing**: Distributes requests among multiple instances of a service to ensure even distribution and high availability.
3. **Caching**: Supports response caching to reduce latency and improve performance by avoiding repeated calls to downstream services.
4. **Security**: Provides mechanisms for securing routes, including support for authentication and authorization.
5. **QoS (Quality of Service)**: Uses Polly integration to handle retries, circuit breakers, and other fault tolerance mechanisms.
6. **Logging and Monitoring**: Facilitates logging and monitoring of API requests for better observability and debugging.

Setting Up Ocelot

1. **Add Ocelot to Your Project**

Install the Ocelot NuGet package in your ASP.NET Core project:

dotnet add package Ocelot

2. **Configure Ocelot in Program.cs**

Set up Ocelot in the Program.cs file of your ASP.NET Core application:

```
using Microsoft.Extensions.DependencyInjection;
using Ocelot.DependencyInjection;
using Ocelot.Middleware;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddOcelot();

var app = builder.Build();

await app.UseOcelot();

app.Run();
```

- **AddOcelot()**: Registers Ocelot services for dependency injection.
- **UseOcelot()**: Adds Ocelot middleware to the HTTP request pipeline.

Configuration File: ocelot.json

The ocelot.json file is used to configure Ocelot. It is typically placed in the root directory of your project and defines routing rules, policies, and other settings.

Basic Structure of ocelot.json:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/products",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
```

```
      "Port": 5001
    }
   ],
   "UpstreamPathTemplate": "/products",
   "UpstreamHttpMethod": [ "Get" ]
  }
 ],
 "GlobalConfiguration": {
  "BaseUrl": "http://localhost:5000"
 }
}
```

- **Routes**: Defines the routing rules for how incoming requests are handled.
  - **DownstreamPathTemplate**: The path to the downstream service.
  - **DownstreamScheme**: The protocol scheme of the downstream service (http or https).
  - **DownstreamHostAndPorts**: The host and port of the downstream service.
  - **UpstreamPathTemplate**: The path used by clients to access the API Gateway.
  - **UpstreamHttpMethod**: HTTP methods that are allowed for the route (GET, POST, etc.).

- **GlobalConfiguration**: Contains global settings for Ocelot.
  - **BaseUrl**: The base URL for the API Gateway.

Key Concepts

1. **Downstream Service**: The service that Ocelot forwards the request to.
2. **Upstream Request**: The client request that reaches the API Gateway.
3. **ReRoute**: The configuration that maps upstream requests to downstream services.

ocelot.json and Its Full Properties Including Downstream and Upstream

The ocelot.json file is the heart of Ocelot's configuration. It defines how Ocelot should handle routing requests, apply policies, and manage various settings. Understanding its structure and properties is crucial for effectively setting up your API Gateway.

Basic Structure of ocelot.json

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/products",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/products",
      "UpstreamHttpMethod": [ "Get" ]
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "http://localhost:5000"
  }
}
```

Routes

The Routes section is where you define the routing rules. Each Route maps an incoming request to a downstream service. Here's a detailed explanation of the properties:

- **DownstreamPathTemplate**: Specifies the path to which Ocelot will forward the request. It represents the path on the downstream service.
    - Example: /api/products – Ocelot will route requests to this path on the downstream service.

- **DownstreamScheme**: Indicates the protocol scheme used by the downstream service. It can be http or https.
    - Example: http – The downstream service is accessed over HTTP.
- **DownstreamHostAndPorts**: An array of host and port combinations for the downstream service.
    - **Host**: The hostname or IP address of the downstream service.
        - Example: localhost – The downstream service is running on the local machine.
    - **Port**: The port number on which the downstream service is listening.
        - Example: 5001 – The downstream service listens on port 5001.
- **UpstreamPathTemplate**: Defines the path that clients use to access the API Gateway. This path is used by Ocelot to match incoming requests.
    - Example: /products – Clients send requests to /products, which are then routed to the downstream service.
- **UpstreamHttpMethod**: Specifies the HTTP methods allowed for this route. It is an array of methods like GET, POST, PUT, etc.
    - Example: ["Get"] – Only GET requests are allowed for this route.

GlobalConfiguration

The GlobalConfiguration section contains settings that apply globally across all routes.

- **BaseUrl**: Defines the base URL for the API Gateway. This is useful for setting the base address of the gateway and can be used to build full URLs for downstream services.
    - Example: http://localhost:5000 – The API Gateway listens on port 5000.

Additional Properties

Ocelot supports several additional properties in the ocelot.json configuration file for advanced scenarios:

- **AuthenticationOptions**: Configures authentication mechanisms for routes. You can specify authentication schemes, credentials, and more.
    - Example: { "AuthenticationProviderKey": "Bearer", "AllowedScopes": ["api1"] }
- **AuthorizationOptions**: Specifies authorization policies and roles required to access routes.
    - Example: { "AllowedScopes": ["api1"] }

- **QoSOptions**: Configures Quality of Service policies, such as retries and circuit breakers, using the Polly library.
    - Example: { "RetryCount": 3, "CircuitBreakerOptions": { "FailureThreshold": 0.5, "DurationOfBreak": "00:01:00" } }
- **RateLimitOptions**: Defines rate limiting rules to control the number of requests allowed from a client within a time period.
    - Example: { "ClientIdHeader": "X-Client-ID", "QuotaExceededMessage": "Rate limit exceeded" }
- **ResponseCachingOptions**: Configures caching for responses, including cache duration and the ability to vary cache based on request headers.
    - Example: { "CacheDuration": "00:05:00" }
- **ServiceDiscoveryProvider**: Defines settings for service discovery, such as using a service registry like Consul.
    - Example: { "Scheme": "http", "Host": "localhost", "Port": 8500 }
- **ReRouteOptions**: Provides additional options for configuring individual routes, such as request transformations and response transformations.
    - Example: { "RequestIdKey": "X-Request-ID", "RequestIdProvider": "Guid.NewGuid" }

Examples

Example 1: Basic Configuration

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/orders",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "order-service",
          "Port": 443
        }
      ],
      "UpstreamPathTemplate": "/orders",
```

```
    "UpstreamHttpMethod": [ "Get" ]
   }
 ],
 "GlobalConfiguration": {
   "BaseUrl": "http://localhost:5000"
 }
}
```

Example 2: Adding Rate Limiting

```
{
 "Routes": [
   {
    "DownstreamPathTemplate": "/api/products",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
     {
       "Host": "localhost",
       "Port": 5001
     }
    ],
    "UpstreamPathTemplate": "/products",
    "UpstreamHttpMethod": [ "Get" ],
    "RateLimitOptions": {
     "ClientIdHeader": "X-Client-ID",
     "QuotaExceededMessage": "Rate limit exceeded",
     "RateLimitRules": [
      {
       "Period": "1m",
       "Limit": 100
      }
     ]
```

```
    }
   }
 ],
 "GlobalConfiguration": {
   "BaseUrl": "http://localhost:5000"
 }
}
```

Key Points to Remember (for interview preparation):

1. **Routes Section**: Defines routing rules in ocelot.json, mapping upstream requests to downstream services.
2. **DownstreamPathTemplate**: Path to the downstream service that Ocelot will forward requests to.
3. **DownstreamScheme**: Protocol scheme used by the downstream service (http or https).
4. **DownstreamHostAndPorts**: Host and port of the downstream service.
5. **UpstreamPathTemplate**: Path that clients use to access the API Gateway.
6. **UpstreamHttpMethod**: Allowed HTTP methods for the route.
7. **GlobalConfiguration**: Global settings for the API Gateway, including the base URL.
8. **Advanced Properties**: Includes options for authentication, authorization, QoS, rate limiting, response caching, and service discovery.

Polly Integration with Ocelot

Polly is a .NET library that provides resilience and transient-fault-handling capabilities. It allows you to define policies such as retries, circuit breakers, timeouts, and more. Integrating Polly with Ocelot enhances the API Gateway's reliability by managing faults and improving system resilience.

Basic Integration Overview

To use Polly with Ocelot, you need to configure the Ocelot gateway to use Polly policies. This involves setting up policies for various fault-tolerance scenarios and applying them to the Routes in the ocelot.json configuration file.

Steps to Integrate Polly with Ocelot

1. **Add Polly NuGet Package**

First, ensure that the Polly NuGet package is installed in your project. You can add it using the Package Manager Console or the NuGet Package Manager.

dotnet add package Polly

2. **Define Polly Policies**

Define your Polly policies (like retries, circuit breakers, etc.) in your application code. This typically involves creating classes or methods that set up these policies.

Example: Defining Polly Policies

```
using Polly;
using Polly.CircuitBreaker;
using Polly.Retry;
using Polly.Timeout;
using Polly.Wrap;
using System;
using System.Net.Http;

public class PollyPolicies
{
   public IAsyncPolicy<HttpResponseMessage> GetRetryPolicy(int retryCount)
   {
      return Policy.HandleResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
         .WaitAndRetryAsync(retryCount, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)));
   }

   public IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy(int handledEventsAllowedBeforeBreaking, TimeSpan durationOfBreak)
   {
      return Policy.HandleResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
         .CircuitBreakerAsync(handledEventsAllowedBeforeBreaking, durationOfBreak);
```

```
    }

    public IAsyncPolicy<HttpResponseMessage> GetTimeoutPolicy(TimeSpan timeout)
    {
        return Policy.TimeoutAsync<HttpResponseMessage>(timeout);
    }

    public IAsyncPolicy<HttpResponseMessage> GetCombinedPolicy()
    {
        var retryPolicy = GetRetryPolicy(3);
        var circuitBreakerPolicy = GetCircuitBreakerPolicy(5, TimeSpan.FromMinutes(1));
        var timeoutPolicy = GetTimeoutPolicy(TimeSpan.FromSeconds(2));

        return Policy.WrapAsync(retryPolicy, circuitBreakerPolicy, timeoutPolicy);
    }
}
```

3. **Configure Ocelot to Use Polly Policies**

In your Ocelot configuration (ocelot.json), you need to integrate the Polly policies with your Routes. This typically involves using the QoSOptions to specify the policies.

Example: Integrating Polly Policies

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

```
    ],
    "UpstreamPathTemplate": "/products",
    "UpstreamHttpMethod": [ "Get" ],
    "QoSOptions": {
      "ExceptionsAllowedBeforeBreaking": 3,
      "DurationOfBreak": "00:00:30",
      "TimeoutValue": 2000
    }
   }
 ],
 "GlobalConfiguration": {
   "BaseUrl": "http://localhost:5000"
 }
}
```

In this configuration:

- **ExceptionsAllowedBeforeBreaking**: Specifies how many exceptions are allowed before the circuit breaker opens.
- **DurationOfBreak**: Defines how long the circuit breaker remains open before transitioning to a half-open state.
- **TimeoutValue**: Sets the timeout duration for requests.

4. **Configure Ocelot Startup**

In the Startup.cs or Program.cs file, ensure you register and configure the policies you have defined. This involves adding the required services and configuring Ocelot to use them.

Example: Registering and Configuring Ocelot

```
builder.Services.AddOcelot()
    .AddPolly();
await app.UseOcelot();
```

Example: Configuring QoSOptions in Code

You can also configure QoS options programmatically, which gives you more control over the policy setup.

Example: Programmatic Configuration

```
using Ocelot.DependencyInjection;

using Ocelot.Middleware;

using Microsoft.Extensions.DependencyInjection;

using Polly;



builder.Services.AddOcelot()

    .AddPolly();

...


await app.UseOcelot();
```

Polly Integration Summary

- **Polly Policies**: Define resilience policies like retries, circuit breakers, and timeouts.
- **Ocelot Configuration**: Integrate Polly policies into the Ocelot configuration using QoSOptions.
- **Startup Configuration**: Register Ocelot and Polly in your application startup.

Key Points to Remember (for interview preparation):

1. **Polly Integration**: Use Polly to add resilience to your API Gateway by handling faults and transient issues.
2. **Policies**: Define and configure policies such as retries, circuit breakers, and timeouts.
3. **QoSOptions**: Configure Quality of Service options in ocelot.json to apply Polly policies.
4. **Startup Configuration**: Ensure Polly and Ocelot are correctly registered and configured in your application's startup code.

QoSOptions

Quality of Service (QoS) options in Ocelot allow you to manage the reliability of your API Gateway by applying various fault-tolerance policies such as circuit breakers, timeouts, and retries. These options help ensure that your gateway handles failures gracefully and maintains a good user experience even when downstream services experience issues.

QoSOptions Overview

QoSOptions provides a way to configure various policies that enhance the resilience of your API Gateway. This section covers the configuration options available for QoS and how they interact with Polly policies.

QoSOptions Properties

1. **ExceptionsAllowedBeforeBreaking**
    - **Definition**: Specifies the number of exceptions (failed requests) that are allowed before the circuit breaker opens.
    - **Purpose**: Helps in managing how many failures are tolerated before the system considers the downstream service to be unreliable.
    - **Type**: Integer
    - **Example**: ExceptionsAllowedBeforeBreaking: 3

2. **DurationOfBreak**
    - **Definition**: Defines how long the circuit breaker remains open before transitioning to a half-open state. During this time, all requests to the downstream service are blocked.
    - **Purpose**: Controls the duration for which the system should wait before checking if the downstream service has recovered.
    - **Type**: TimeSpan (in hh:mm:ss format)
    - **Example**: DurationOfBreak: "00:00:30"

3. **TimeoutValue**
    - **Definition**: Sets the timeout duration for requests made through the API Gateway. If a request takes longer than this duration, it is considered to have failed.
    - **Purpose**: Ensures that requests that take too long are not indefinitely waited on, thus preventing system bottlenecks and improving responsiveness.
    - **Type**: TimeSpan (in milliseconds)
    - **Example**: TimeoutValue: 2000 (2 seconds)

4. **RetryCount** (Optional in some configurations)
    - **Definition**: Specifies the number of times a failed request should be retried before considering it a failure.
    - **Purpose**: Provides the ability to automatically retry requests that fail due to transient issues, improving resilience and reliability.
    - **Type**: Integer
    - **Example**: RetryCount: 5

Example Configuration in ocelot.json

Here's how you can configure QoS options in the ocelot.json file for a ReRoute:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/products",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/products",
      "UpstreamHttpMethod": [ "Get" ],
      "QoSOptions": {
        "ExceptionsAllowedBeforeBreaking": 3,
        "DurationOfBreak": "00:00:30",
        "TimeoutValue": 2000,
        "RetryCount": 5
      }
    }
  ],
  "GlobalConfiguration": {
```

```
  "BaseUrl": "http://localhost:5000"
 }
}
```

Applying QoSOptions Programmatically

To configure QoS options programmatically in your application code, you would typically use the Ocelot configuration API to apply these settings.

Example Code: Programmatic QoS Configuration

```
using Microsoft.Extensions.DependencyInjection;
using Ocelot.DependencyInjection;
using Ocelot.Middleware;

builder.Services.AddOcelot()
    .AddPolly(); // Ensure Polly is added if using Polly policies

...

app.UseOcelot().Wait();
```

In this example, Ocelot is configured to use the QoSOptions specified in the ocelot.json file. You can further customize the QoS settings by adding more detailed configurations in code if needed.

Key Points to Remember (for interview preparation):

1. **QoSOptions**: Configures how the API Gateway handles fault tolerance and resilience through properties like ExceptionsAllowedBeforeBreaking, DurationOfBreak, TimeoutValue, and RetryCount.
2. **Configuration in ocelot.json**: Define QoS policies in the Ocelot configuration file to apply fault-tolerance settings to your Routes.
3. **Programmatic Configuration**: Use code-based configurations if needed to further customize QoS settings beyond what's defined in ocelot.json.

Rate Limits

Rate limiting is a crucial feature in an API Gateway to manage the number of requests that a client can make to an API within a specific time period. It helps protect your backend services from being overwhelmed by too many requests and ensures fair usage across different clients.

Rate Limits Overview

Rate limiting involves setting thresholds on the number of requests allowed from a client in a given time window. Ocelot provides support for rate limiting through its configuration, allowing you to specify limits on requests based on various criteria.

Rate Limits Properties

1. **ClientId** (Optional)
    - **Definition**: A unique identifier for the client making the request.
    - **Purpose**: Allows rate limits to be applied based on individual client IDs, ensuring that different clients are subject to their own limits.
    - **Type**: String
    - **Example**: "ClientId": "client-123"

2. **QuotaExceededMessage**
    - **Definition**: The message that is returned to the client when the rate limit is exceeded.
    - **Purpose**: Provides a custom response to inform the client that they have exceeded their allowed request limit.
    - **Type**: String
    - **Example**: "QuotaExceededMessage": "Rate limit exceeded. Please try again later."

3. **RateLimitOptions**
    - **Definition**: Defines the rate limiting policy for the API route.
    - **Purpose**: Specifies how many requests are allowed within a certain time period.
    - **Properties**:
        - **Period**: The time period for which the request count is tracked.

        - **Type**: TimeSpan (in hh:mm:ss format)
        - **Example**: "Period": "00:01:00" (1 minute)
    - **Limit**: The maximum number of requests allowed in the defined period.
        - **Type**: Integer
        - **Example**: "Limit": 100 (100 requests per minute)

Example Configuration in ocelot.json

Here's how you can configure rate limiting in the ocelot.json file for a ReRoute:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/products",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/products",
      "UpstreamHttpMethod": [ "Get" ],
      "RateLimitOptions": {
        "Period": "00:01:00",
        "Limit": 100,
        "QuotaExceededMessage": "Rate limit exceeded. Please try again later."
      }
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "http://localhost:5000"
```

```
  }
}
```

In this example, the rate limit is set to allow up to 100 requests per minute. If the client exceeds this limit, they will receive a custom quota exceeded message.

Applying Rate Limits Programmatically

To configure rate limits programmatically, you would use the Ocelot configuration API to set up rate limiting policies.

Example Code: Programmatic Rate Limit Configuration

```
using Microsoft.Extensions.DependencyInjection;

using Ocelot.DependencyInjection;

using Ocelot.Middleware;



builder.Services.AddOcelot()

    .AddRateLimit(); // Ensure rate limiting services are added


...


await app.UseOcelot();
```

In this example, rate limiting is enabled for Ocelot. You may need to define additional settings in the ocelot.json file or use custom middleware if more complex rate-limiting scenarios are required.

Key Points to Remember (for interview preparation):

1. **Rate Limiting**: Manages the number of requests a client can make to the API within a specific time period.
2. **Configuration in ocelot.json**: Define rate limits using properties like Period, Limit, and QuotaExceededMessage.
3. **Programmatic Configuration**: Use Ocelot's configuration API to apply rate limiting if needed.

Response Caching

Response caching is a technique used to improve the performance and scalability of your API by storing and reusing previously generated responses. In an API Gateway context, response caching can significantly reduce the load on your backend services and improve response times for frequently requested data.

Introduction to Response Caching

When a client makes a request, the API Gateway can store the response in a cache. Subsequent requests for the same resource can be served from the cache rather than forwarding the request to the backend service. This reduces the number of calls to the backend and speeds up response times.

Response Caching Properties in Ocelot

Ocelot supports response caching through its configuration. You can specify how long responses should be cached and under what conditions. This involves defining caching policies in the ocelot.json file.

1. **CacheKey**
    - **Definition**: A unique key for identifying the cached response.
    - **Purpose**: Ensures that each cached response can be uniquely identified and retrieved.
    - **Type**: String
    - **Example**: "CacheKey": "products-cache-key"

2. **CacheDuration**
    - **Definition**: The duration for which the response should be cached.
    - **Purpose**: Controls how long the cached response remains valid before it is considered stale and needs to be refreshed.
    - **Type**: TimeSpan (in hh:mm:ss format)
    - **Example**: "CacheDuration": "00:10:00" (10 minutes)

3. **AllowCachedResponses**
    - **Definition**: A boolean value indicating whether cached responses are allowed.
    - **Purpose**: Enables or disables caching for a specific route.
    - **Type**: Boolean
    - **Example**: "AllowCachedResponses": true

Example Configuration in ocelot.json

Here's how you can configure response caching in the ocelot.json file for a ReRoute:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/products",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "UpstreamPathTemplate": "/products",
      "UpstreamHttpMethod": [ "Get" ],
      "ResponseCachingOptions": {
        "CacheKey": "products-cache-key",
        "CacheDuration": "00:10:00",
        "AllowCachedResponses": true
      }
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "http://localhost:5000"
  }
}
```

In this example, responses for the /products endpoint will be cached for 10 minutes. The cache key is set to "products-cache-key", ensuring that responses are uniquely identified.

Applying Response Caching Programmatically

To enable and configure response caching programmatically, you need to add response caching services and middleware in your ASP.NET Core application.

Example Code: Enabling Response Caching in ASP.NET Core

```
using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.DependencyInjection;



builder.services.AddResponseCaching();

builder.services.AddOcelot();



...


app.UseResponseCaching();

await app.UseOcelot();
```

In this example:

- AddResponseCaching() registers the response caching services.
- UseResponseCaching() adds the response caching middleware to the request pipeline.

Key Points to Remember (for interview preparation):

1. **Response Caching**: Stores and reuses responses to reduce load on backend services and improve performance.
2. **Configuration in ocelot.json**: Define caching policies using properties like CacheKey, CacheDuration, and AllowCachedResponses.
3. **Programmatic Configuration**: Use ASP.NET Core middleware and services to enable and configure response caching.