**Low-Level Design Document**

1. **Introduction**
2. **Architecture Overview**
3. **Component Details**
4. **Data Flow**
5. **Component Design**
6. **Azure Service Integration**
7. **Error Handling and Logging**
8. **Security and Authentication**
9. **Performance Considerations**
10. **Testing Strategy**
11. **Deployment**
12. **Maintenance**
13. **Monitoring**

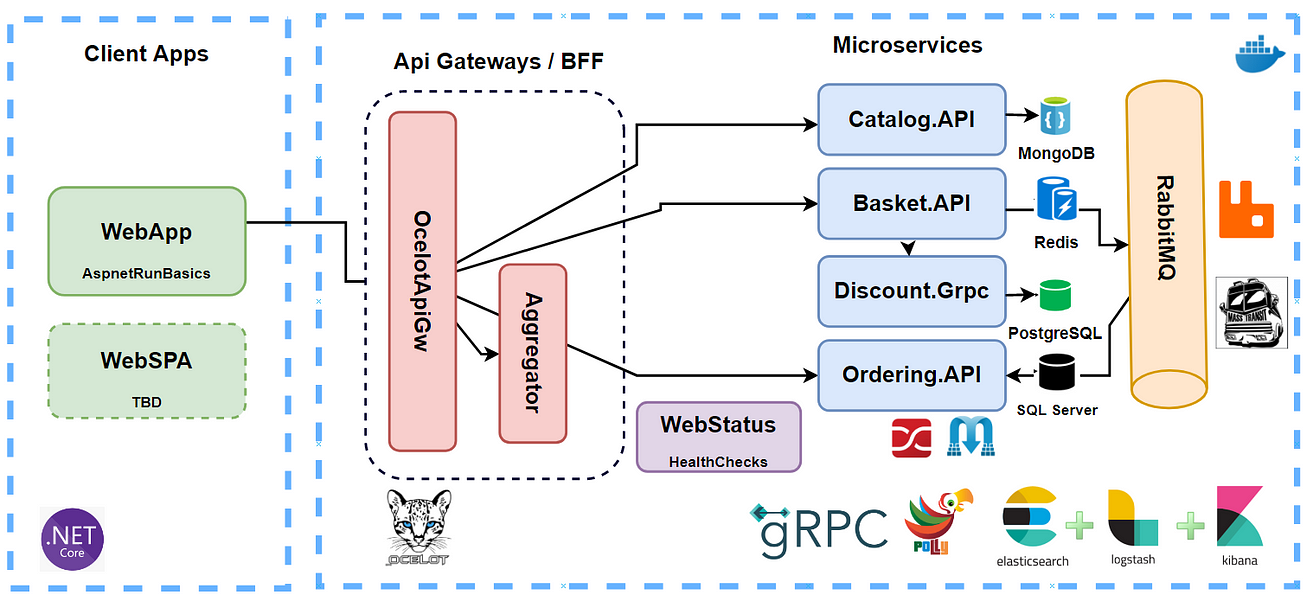
A Low-Level Design (LLD) document is a detailed description of how a system or a component will be implemented. It's a technical document that provides an in-depth look at the architecture, components, data flow, and interactions of a software system. Below is a template for creating a .NET Azure Low-Level Design document. This example focuses on designing a component that interacts with Azure services.

**Title: Low-Level Design Document - .NET Azure Component**

**Introduction**

*Building Microservices on .Net platforms which used Asp.Net Web API, Docker, RabbitMQ, gRPC, Ocelot API Gateway, MongoDB, Redis, PostgreSQL, SqlServer, Entity Framework Core, Dapper, CQRS and Clean Architecture implementation.*

1. ***Docker****for containerize and orchestrator*
2. *Microservices****communications****with****gRPC****and****RabbitMQ***
3. *API Gateways with****Ocelot API Gateway***
4. *databases platforms****NoSQL(MongoDB, Redis)***
5. ***Relational******databases(PostgreSQL, SqlServer)***
6. ***Dapper, Entity Framework****Core for ORM Tools*
7. *best practices****CQRS****with****Clean Architecture***



**1. What Are Microservices?**

* Microservices are a way of structuring software applications as a collection of small, independent services that work together to provide the complete functionality of an application.
* Each microservice is responsible for a specific business capability or function, such as user authentication, payment processing, or product catalog.

**2. Benefits of Microservices:**

* **Scalability:** Microservices can be scaled independently, allowing you to allocate more resources to parts of your application that need it the most.
* **Flexibility:** Teams can develop and update microservices independently, making it easier to adapt to changing business requirements.
* **Fault Isolation:** If one microservice fails, it doesn't necessarily bring down the entire application, enhancing overall system resilience.
* **Technology Diversity:** Different microservices can use different technologies or programming languages, enabling you to choose the best tool for each job.
* **Rapid Deployment:** Microservices can be deployed and updated quickly, reducing time-to-market for new features.

**4. Challenges and Considerations:**

* **Complexity:** While microservices offer flexibility, managing many services can become complex, so it's essential to have a well-defined governance strategy.
* **Data Management:** Handling data consistency and sharing among microservices can be challenging and requires careful design.
* **Monitoring and Security:** Implementing robust monitoring and security practices are critical to maintaining the health and integrity of your microservices architecture.

**5. Business Impact:**

* By adopting Azure microservices architecture, your organization can be more agile, responsive to market changes, and efficient in delivering customer experiences.
* It enables faster development cycles, reduced downtime, and improved scalability, all of which can contribute to increased customer satisfaction and revenue growth.

**2. Architecture Overview:**

* Choose appropriate Azure services that align with the application's needs (Azure App Service, Azure Kubernetes Service, Azure SQL Database, etc.).
* Utilize Azure Availability Zones to distribute resources across physically separate data centers for redundancy.
* Implement auto-scaling to handle varying loads and traffic spikes.
* Use Azure Traffic Manager for global load balancing across regions.
* Leverage Azure Virtual Network for secure communication between components.

**12-factor principles**

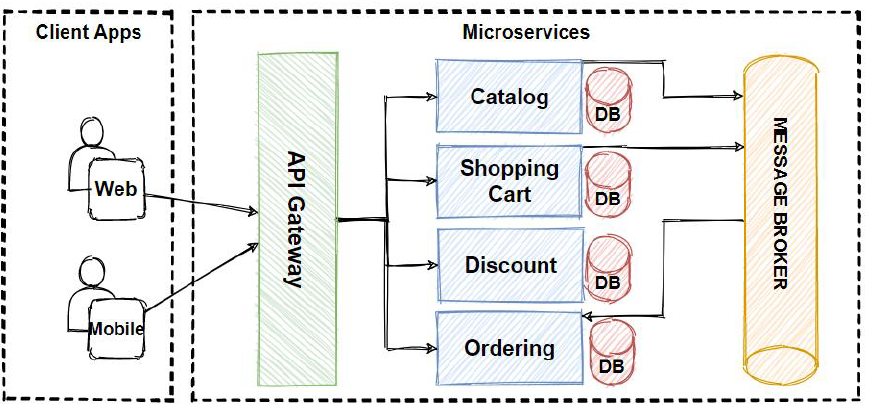
* **Codebase** (One codebase tracked in revision control, many deploys)
* **Dependencies** (Explicitly declare and isolate the dependencies)
* **Config** (Store configurations in an environment)
* **Backing Services** (treat backing resources as attached resources)

include any service that is communicated with over a network.

Treat backing services as an attached resource.

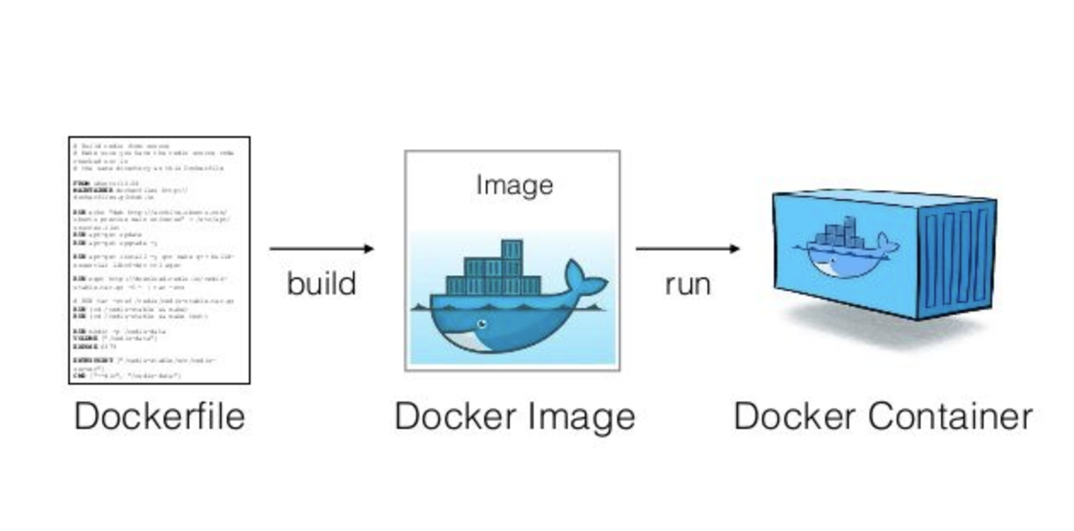
* **Build, release, and Run** (Strictly separate build and run stages)
* **Processes** (execute the app as one or more stateless processes)
* **Port Binding** (Export services via port binding)
* **Concurrency** - scaling the application (Scale out via the process model)
* **Disposability** - The microservice processes can be started or stopped immediately whenever necessary.
* **Dev/prod parity** (Keep development, staging, and production as similar as possible)
* **Logs** (Treat logs as event streams)
* **Admin processes** (Run admin/management tasks as one-off processes)





Description of how the component interacts with other components and Azure services.

|  |  |
| --- | --- |
| Catalog microservice | * ASP.NET Core Web API application * REST API principles, CRUD operations * **MongoDB NoSQL database**connection on docker containerization * N-Layer implementation with Repository Pattern * Swagger Open API implementation * Dockerfile and docker-compose implementation |
| Basket microservice | ASP.NET Core Web API application  REST API principles, CRUD operations  **Redis database** connection on docker containerization  **Consume Discount Grpc Service** for inter-service sync communication to calculate product final price  **Publish BasketCheckout Queue** with using **MassTransit**and **RabbitMQ**  Swagger Open API implementation  Dockerfile and docker-compose implementation |
| Discount microservice | ASP.NET **Grpc Server** application  Build a Highly Performant**inter-service gRPC Communication** with Basket Microservice  **Exposing Grpc Services** with creating **Protobuf messages**  Using **Dapper**for **micro-orm** implementation to simplify data access and ensure high performance  **PostgreSQL database** connection and containerization  Dockerfile and docker-compose implementation |
| Microservices Communication | Sync inter-service **gRPC Communication**  **Async**Microservices Communication with **RabbitMQ Message-Broker Service**  Using **RabbitMQ Publish/Subscribe Topic Exchange Model**  Using **MassTransit**for abstraction over**RabbitMQ Message-Broker system**  **Publishing BasketCheckout event queue** from **Basket**microservices and Subscribing this event from **Ordering**microservices  Create **RabbitMQ EventBus.Messages Common Class Library** and add references Microservices |
| Ordering microservice | ASP.NET Core Web API application  Implementing **DDD, CQRS and Clean Architecture** with using **Best Practices**  Developing **CQRS**with using**MediatR, FluentValidation**and **AutoMapper**nuget packages  **Consuming RabbitMQ BasketCheckout** event queue with using **MassTransit-RabbitMQ Configuration**  **SqlServer**database connection and containerization  Using **Entity Framework Core**ORM and auto migrate to **SqlServer**when application Startup  Swagger Open API implementation  Dockerfile and docker-compose implementation |
| API Gateway Ocelot microservice | Implement API Gateways with **Ocelot**  Sample microservices/containers to**reroute through the API Gateways**  Run multiple different **API Gateway/BFF**container types  **The Gateway aggregation pattern**in Shopping.Aggregator  Dockerfile and docker-compose implementation |
| Microservices Cross-Cutting Implementations |  |
|  |  |
|  |  |



**3. Component Details:**

• Detailed description of the .NET component's responsibilities and features.

• List of Azure services that the component interacts with (e.g., Azure Blob Storage, Azure SQL Database).

Azure Services:

|  |  |  |
| --- | --- | --- |
|  | Azure App Services | App service plan |
|  | Azure SQL Database | Replication, Disaster recovery, Disk snapshot, failover protection |
|  | Azure Storage | (Blob,table,disk, queue, Fle) |
|  | Azure Service Bus | (event grid, event hub) |
|  | Azure application gateway | (Front door, load balancer, Traffic manager) |
|  | Azure API Management |  |
|  | Azure Kubernetes Service |  |
|  | Azure Container Registries |  |
|  | Azure Private endpoint |  |
|  | Azure CosmosDB |  |
|  | Azure Durable Function |  |
|  | Azure Logic app |  |
|  | Azure KeyVault |  |
|  | Azure application Insight |  |
|  | Azure Log Analytics |  |
|  | Azure Monitor |  |
|  | Microsoft Defender for Cloud |  |
|  | Microsoft Sentinel |  |
|  | Azure Bastion |  |
|  | Azure DNS Service |  |
|  | Azure DevOps |  |
|  | Azure AD(User, Role) |  |
|  | Azure Policies |  |
|  | VNET, Subnet, NSG |  |
|  | Azure Redis Cache |  |
|  | Azure Virtual Machine |  |

**Service Code Flow Details:**

1. When PR raised then validation for Service changes Ready on feature branch. (Followed Code Review checklist, Lint, Unit Testing Code coverage via Sonar Cube tool, Vulnerability, Veracode code scanning etc.
2. After PR approval- Azure DevOps CICD Pipeline triggers for build, artifact (using Docker, ACR, AKS) and deploy on AKS Services via release pipeline with lead or SME approval.

**4. Data Flow:**

* Sequence diagrams illustrating the flow of data and interactions between the component and Azure services.
* Describe how data is fetched, processed, and stored.

Design a fault-tolerant data storage approach using Azure SQL Database or Azure Cosmos DB with replication across regions.

Implement geo-replication and automated backups to ensure data integrity and availability.

**5. Component Design:**

* Detailed class diagrams showing the internal structure of the .NET component.
* Explanation of each class's purpose, methods, and relationships.

**6. Azure Service Integration:**

* Details about how the component interacts with Azure services:
  + **Azure Blob Storage:** How the component uploads/downloads files, sets access permissions, etc.
  + **Azure SQL Database:** How the component establishes connections, performs CRUD operations, etc.
  + ... (Other Azure services as applicable)

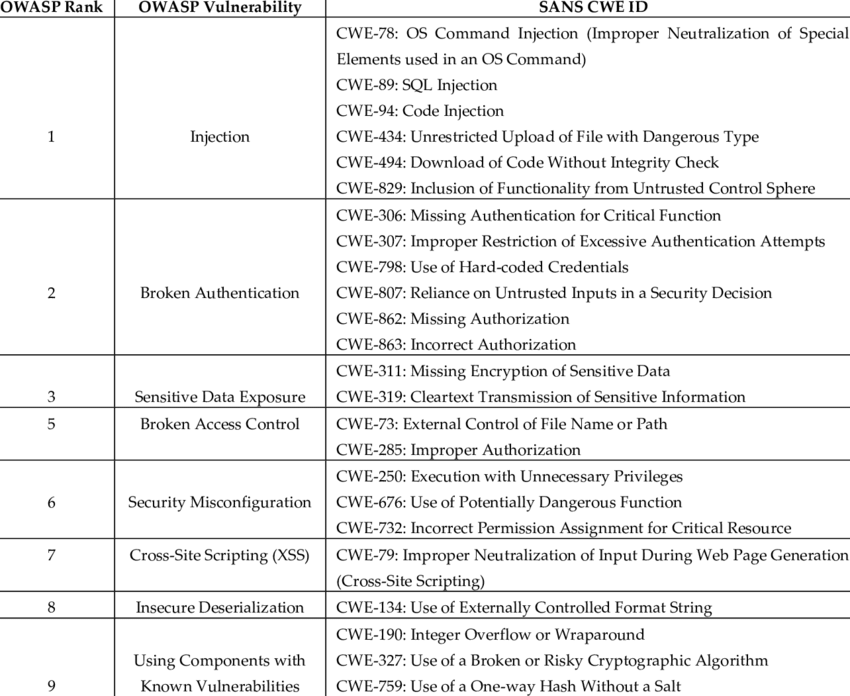
**7. Error Handling and Logging:**

* Explanation of how errors are handled within the component.
* Description of the logging mechanism used to record component activities and errors.

**8. Security and Authentication:**

* Explanation of how authentication and authorization are implemented for Azure services.
* Discuss any encryption, secure connections, or token-based authentication.

 Top 10 Security Vulnerabilities by OWASP



**Authentication Using IdentityFramework:**

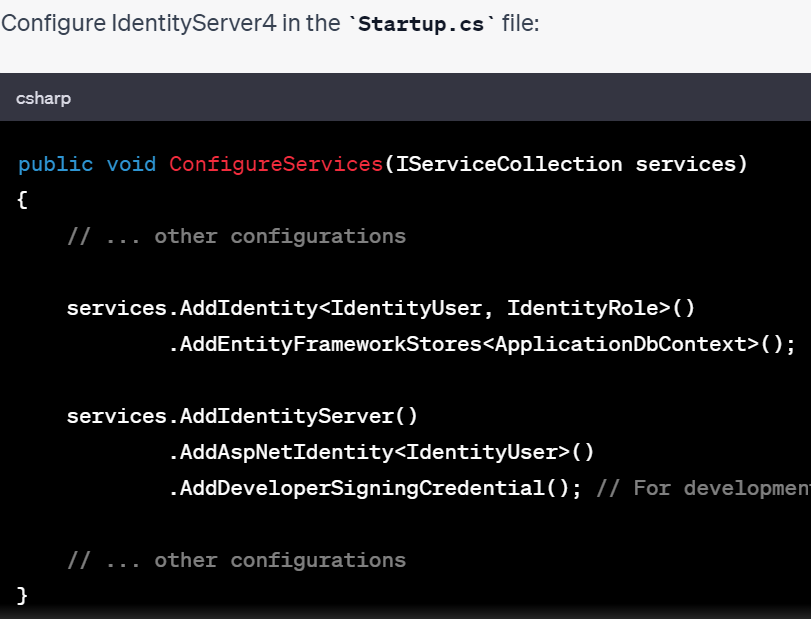
**Step 1: Create IdentityServer4 Application**

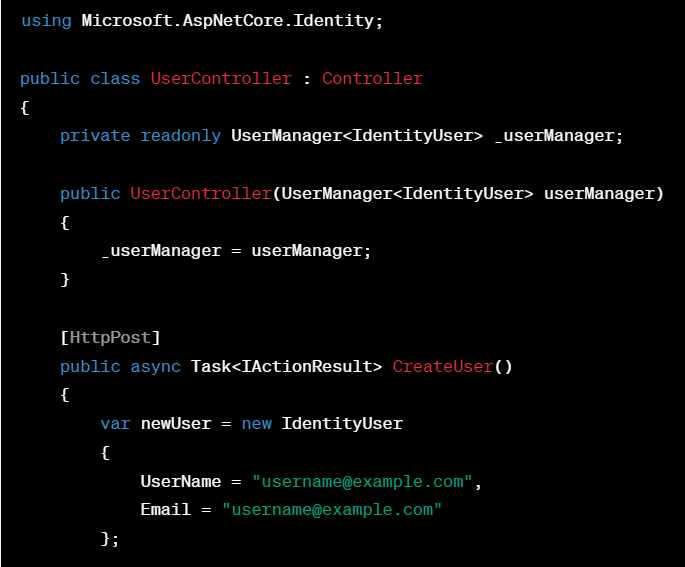
* Install-Package IdentityServer4
* Install-Package IdentityServer4.AspNetIdentity

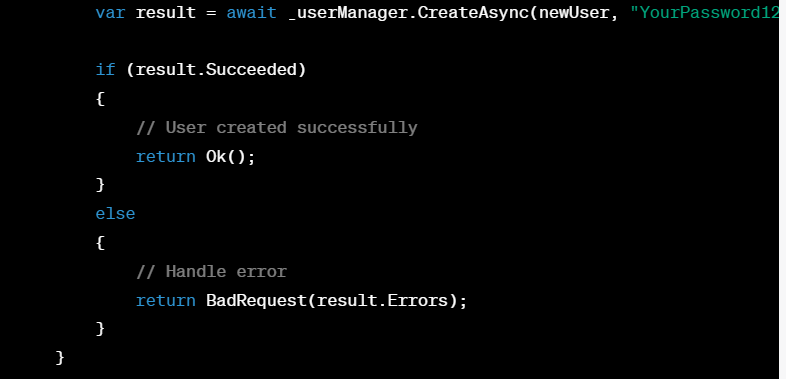
Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore

**1. Set Up Azure AD and App Service:**

* Set up an Azure Active Directory (AD) and register an application.
* Configure your Azure App Service to use Azure AD for authentication.







**Identity Server:**

var tokenResponse = await client.RequestClientCredentialsTokenAsync(new ClientCredentialsTokenRequest

{

Address = discoveryDocument.TokenEndpoint,

ClientId = "client",

ClientSecret = "secret",

Scope = "api1"

});

if (tokenResponse.IsError)

{

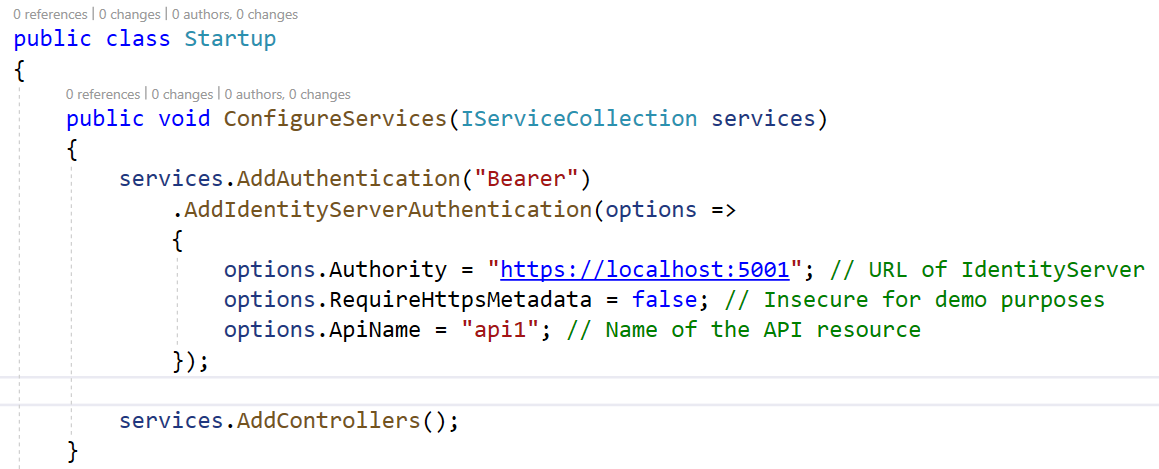
Console.WriteLine($"Error: {tokenResponse.Error}");

return;

}

Console.WriteLine($"Access Token: {tokenResponse.AccessToken}");

**API Call :**



using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

namespace Api.Controllers

{

[Authorize("ApiScope")]

[ApiController]

[Route("[controller]")]

public class WeatherForecastController : ControllerBase

{

using Microsoft.AspNetCore.Authentication.JwtBearer;

namespace Api

{

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

// ...

**services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)**

**.AddJwtBearer(options =>**

**{**

**options.Authority = "https://your-identity-server-url";**

**options.Audience = "api-resource"; // Should match the audience claim in the token**

**});**

**services.AddAuthorization(options =>**

**{**

**options.AddPolicy("ApiScope", policy =>**

**{**

**policy.RequireAuthenticatedUser();**

**policy.RequireClaim("scope", "api-scope"); // Should match the scope claim in the token**

**});**

**});**

// ...

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

}

}

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.AspNetCore.Authorization;

using Microsoft.Extensions.DependencyInjection;

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.Authority = "https://your-identity-server-url";

options.Audience = "api-resource"; // Should match the audience claim in the token

});

services.AddAuthorization(options =>

{

options.AddPolicy("ApiScope", policy =>

{

policy.RequireAuthenticatedUser();

policy.RequireClaim("scope", "api-scope"); // Should match the scope claim in the token

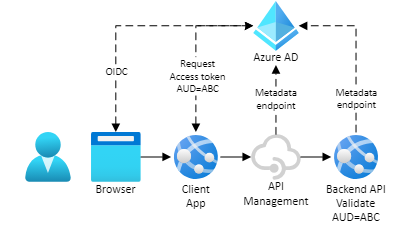
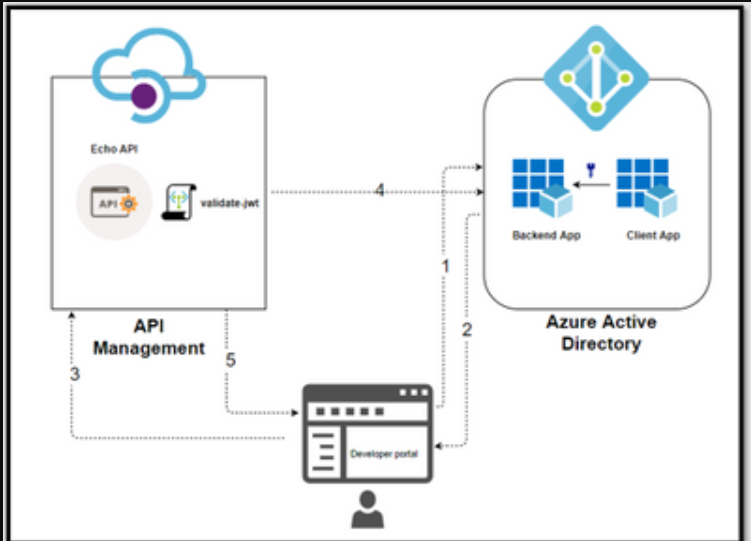
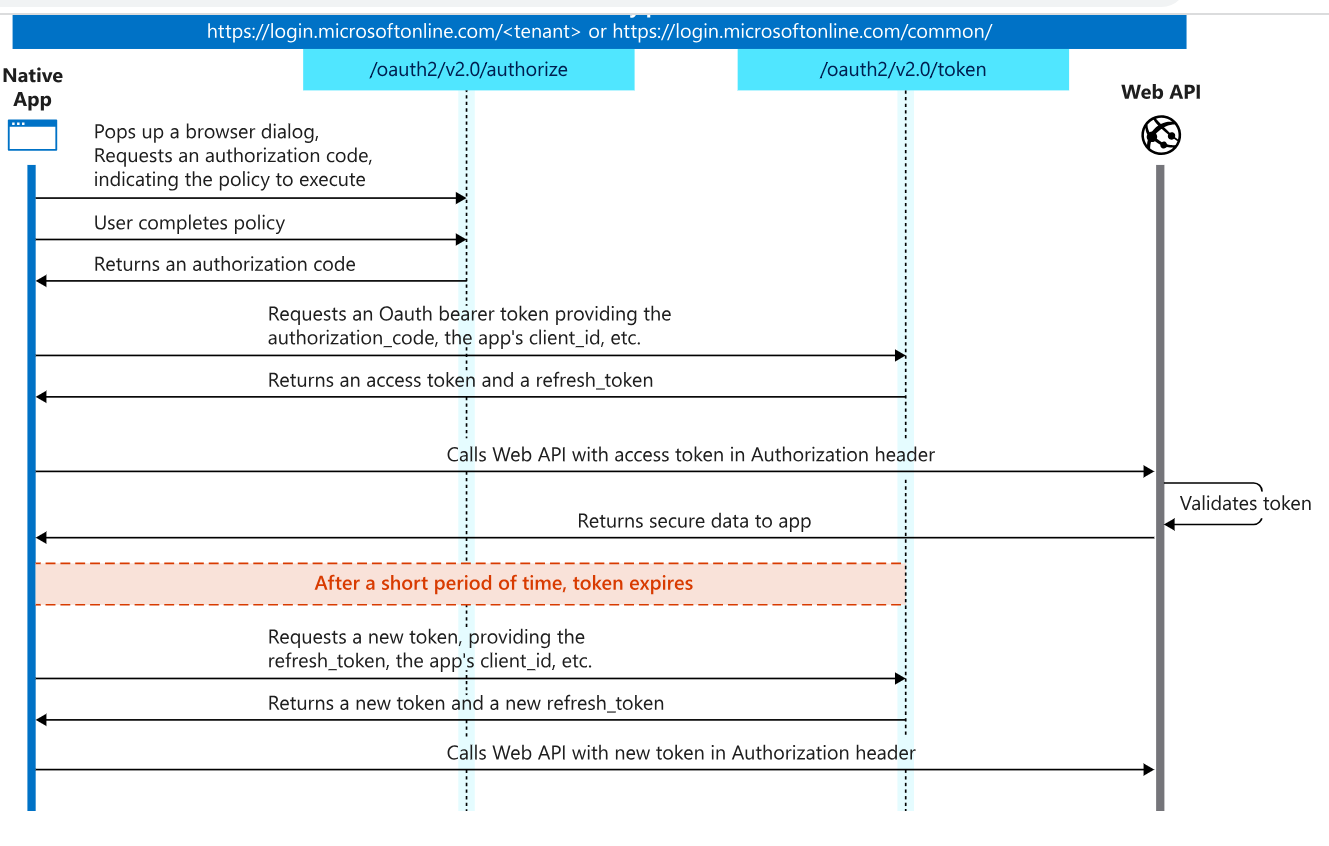
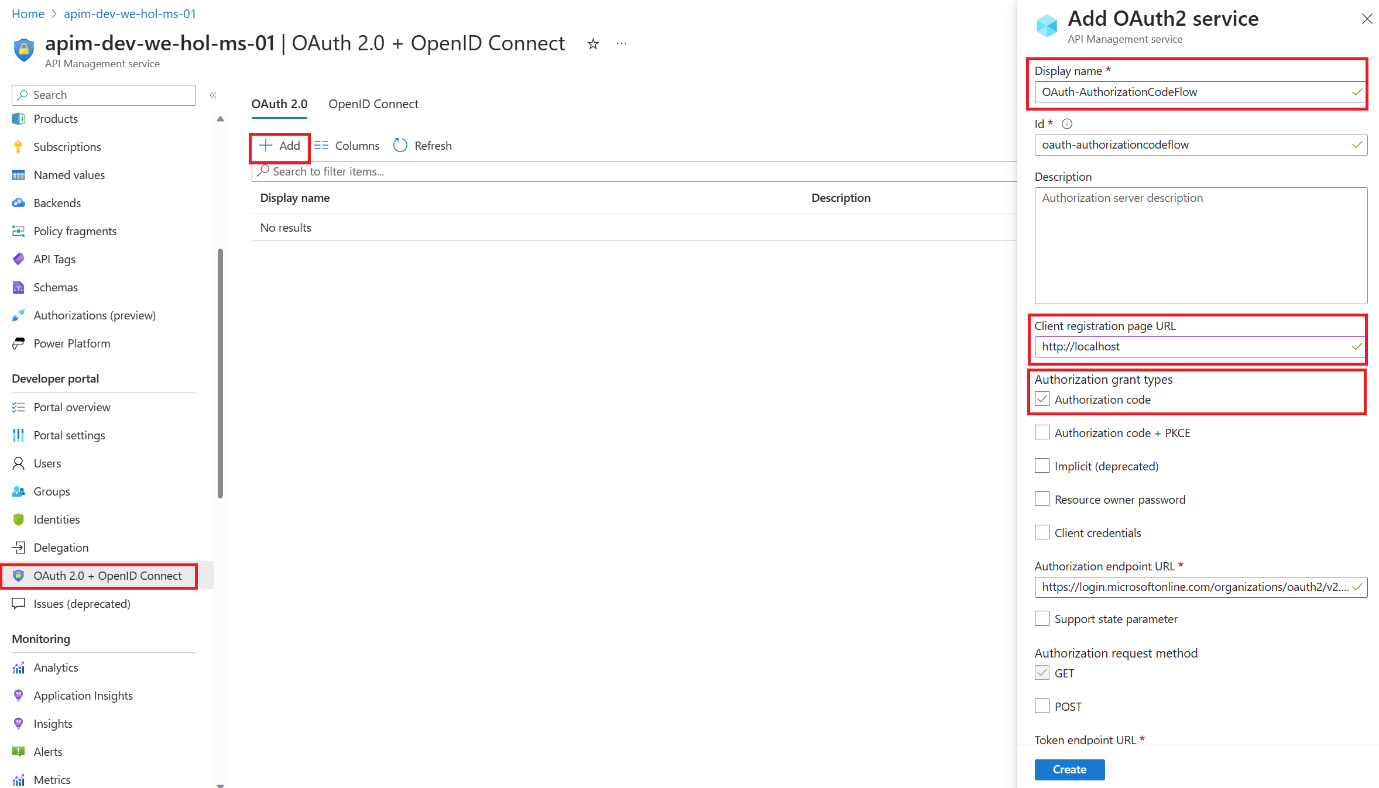
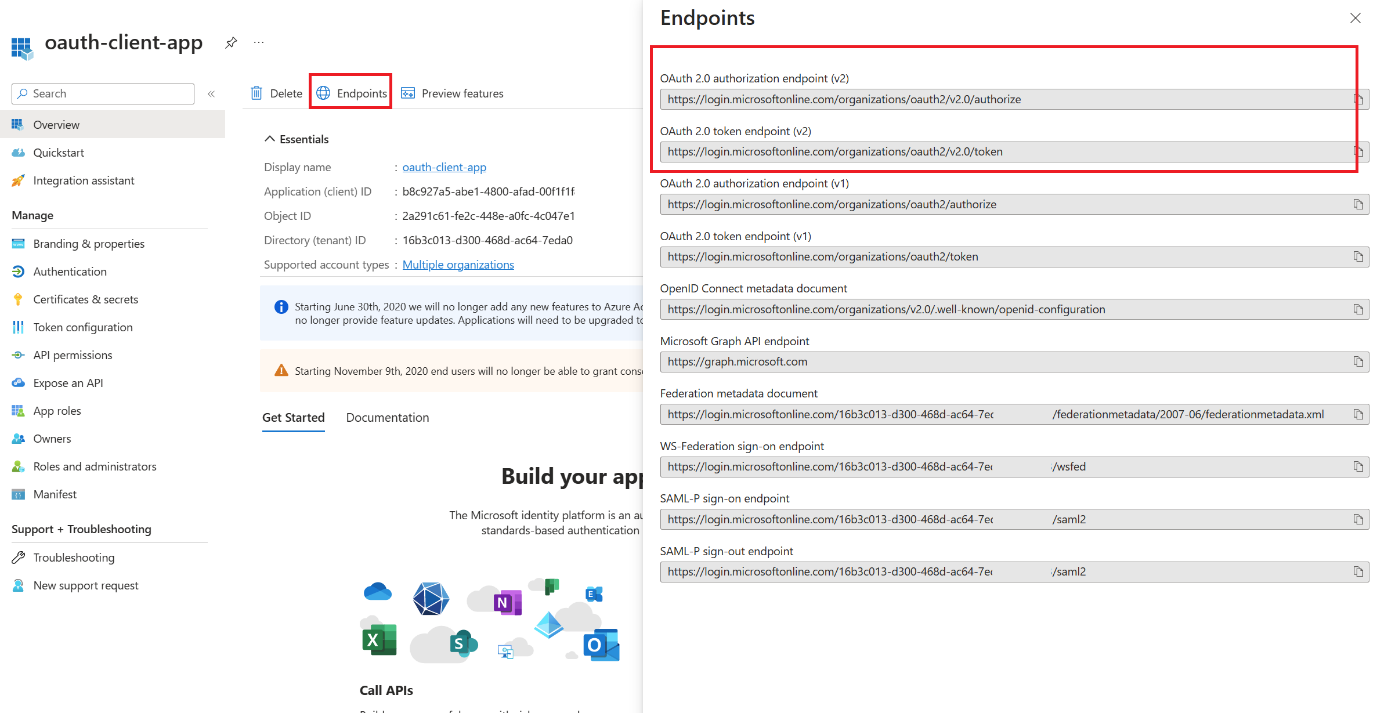
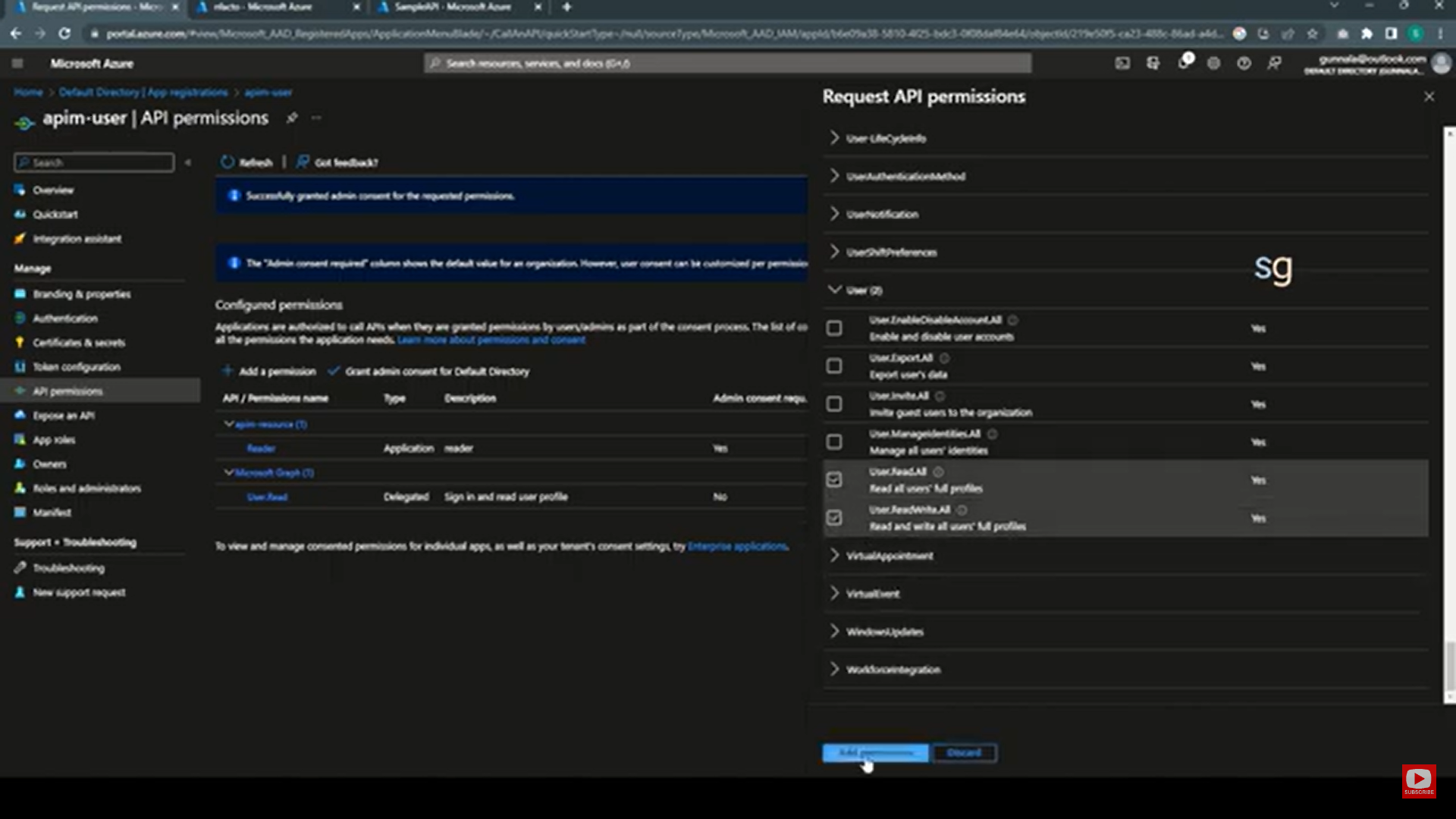
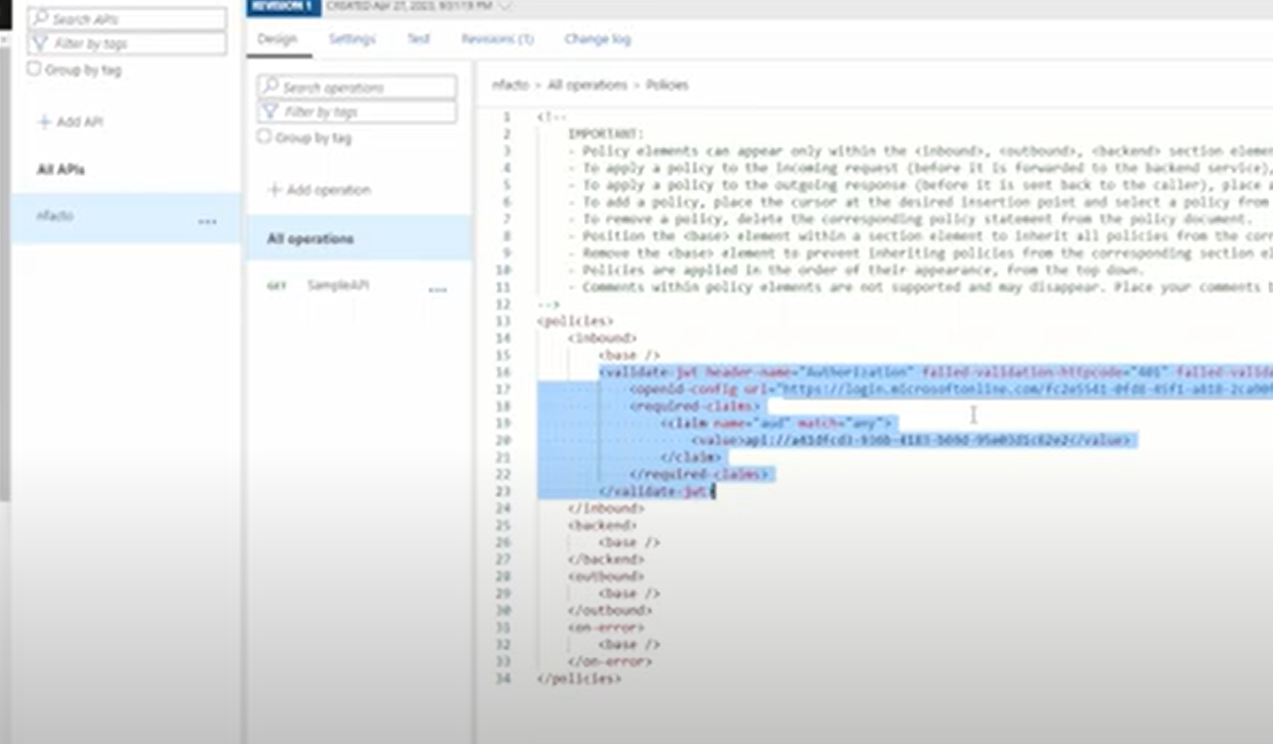
});

});

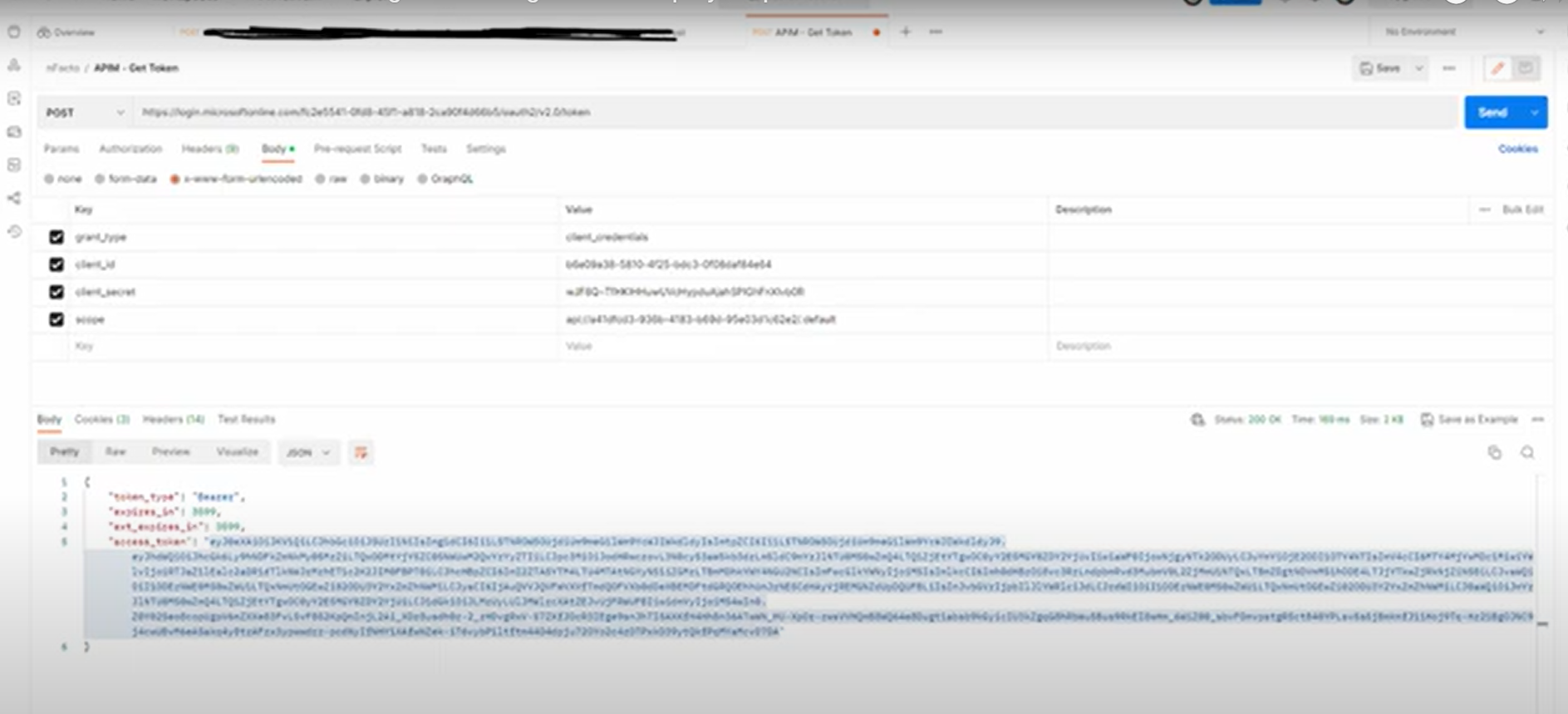
// ...

}

**2. Configure Azure API Management:**

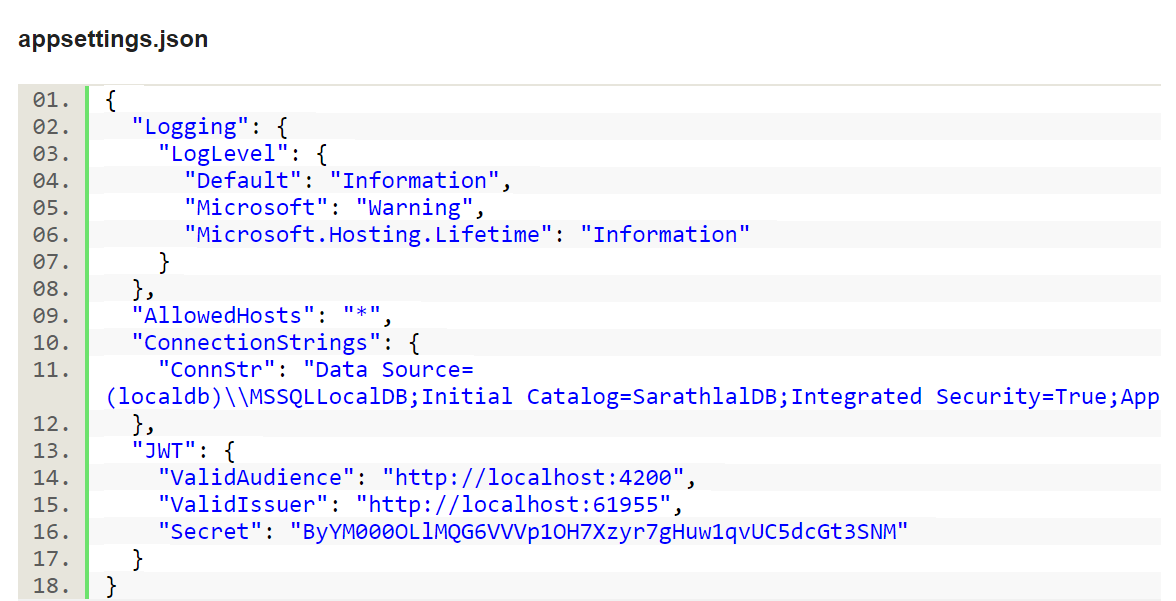
* Set up an Azure API Management service.
* Import the API from your Azure App Service into APIM.
* 
* 
* 
* 
* 
* 
* 
* <https://azure.github.io/apim-lab/apim-lab/7-security/security-7-2-3-oauth2-authorization-grant-flow.html>

1. **Obtain JWT Token:** In Postman, you can follow these steps to obtain a JWT token:

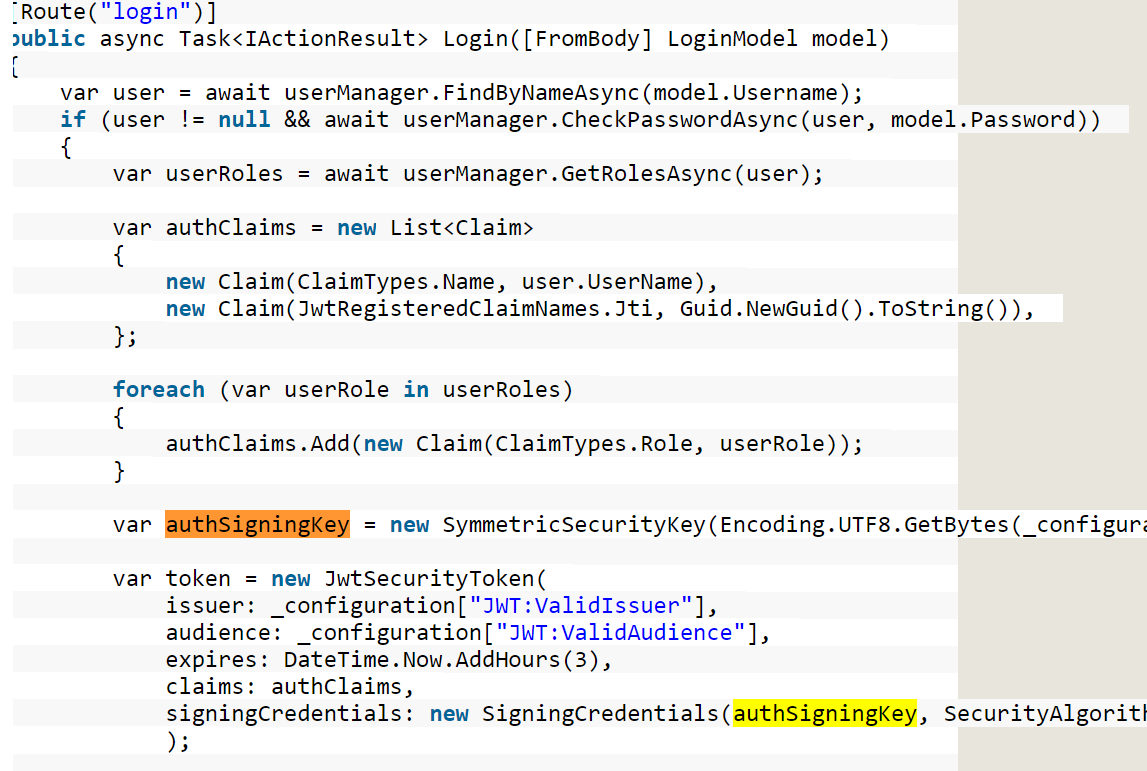


2nd way (MVC web app and web API)

https://www.c-sharpcorner.com/article/authentication-and-authorization-in-asp-net-core-web-api-with-json-web-tokens/



**var authSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_configuration[**"JWT:Secret"**]));**



**return** Ok(**new**

{

token = **new** JwtSecurityTokenHandler().WriteToken(**token**),

expiration = token.ValidTo

});



**9. Performance Considerations:**

* Discuss any performance bottlenecks anticipated and how they are addressed.
* Consider optimizations such as caching, asynchronous operations, etc.

1. **Testing Strategy:** - Describe how the component will be tested, including unit tests, integration tests, and Azure-specific testing (e.g., using Azure DevOps pipelines).

**11. Deployment:** - Describe the process of deploying the component to Azure, including any required configurations. - Discuss considerations for scalability and high availability.

**12. Maintenance and Monitoring:** - Explain how the component will be maintained, updated, and monitored in the Azure environment. - Describe any Azure monitoring tools or services that will be used.

**13. Conclusion:** - Summarize the key points of the Low-Level Design and reiterate the component's importance within the overall system.

Remember that the Low-Level Design document should be comprehensive, addressing technical details in depth. However, it's also essential to strike a balance between providing enough information and avoiding unnecessary complexity.

Design pattern:

 Azure DevOps/

**transformation from on-premises to Azure hosted solutions.**

Migrating from on-premises solutions to Azure-hosted solutions involves several steps and considerations to ensure a successful transition. Here's a high-level overview of the transformation process:

1. **Assessment and Planning:**
   * Evaluate your existing on-premises infrastructure, applications, and workloads to identify what can be migrated to Azure.
   * Determine the goals of your migration, whether it's cost savings, scalability, improved performance, or other factors.
   * Create a detailed migration plan that outlines the sequence of migration, dependencies, and potential challenges.
2. **Azure Resource Selection:**
   * Choose the appropriate Azure services and resources that align with your on-premises workloads. For example, virtual machines, containers, serverless functions, databases, etc.
   * Consider factors such as compute, storage, networking, and security requirements.
3. **Data Migration:**
   * Plan the migration of your data to Azure. This might involve using Azure Data Migration Services, Azure Database Migration Service, or other migration tools depending on your data sources.
   * Ensure data integrity and consistency during the migration process.
4. **Application Migration:**
   * Depending on your applications, you might need to rehost, refactor, rearchitect, or rebuild them to fit the Azure environment.
   * Lift-and-shift (rehosting) involves moving applications with minimal changes to Azure virtual machines.
   * Refactoring and rearchitecting involve adapting applications to leverage Azure-native services for better scalability and performance.
5. **Networking and Security:**
   * Configure Azure networking to replicate your on-premises network architecture. This might involve creating virtual networks, subnets, VPNs, and ExpressRoute connections.
   * Implement security measures, such as Azure Active Directory for identity and access management, and configure firewall rules to secure your resources.
6. **Testing and Validation:**
   * Conduct thorough testing of the migrated applications and workloads in the Azure environment.
   * Perform functional testing, performance testing, and user acceptance testing to ensure everything works as expected.
7. **Data Synchronization and Cutover:**
   * Plan the final data synchronization and cutover to minimize downtime.
   * Redirect users and traffic from on-premises to the Azure-hosted solutions.
8. **Monitoring and Optimization:**
   * Set up monitoring and alerting using tools like Azure Monitor and Azure Application Insights to ensure the performance and availability of your solutions.
   * Continuously monitor and optimize your Azure resources based on usage patterns and performance data.
9. **Training and Documentation:**
   * Provide training to your team on managing and operating Azure-hosted solutions.
   * Create documentation that outlines the new Azure environment, architecture, and operational procedures.
10. **Post-Migration Review:**
    * After migration, conduct a review to assess the success of the migration and identify areas for improvement.

Remember that the migration process can vary significantly based on your specific applications, workloads, and requirements. It's crucial to engage with Azure experts and utilize Azure's migration tools and resources to ensure a smooth and successful transition.