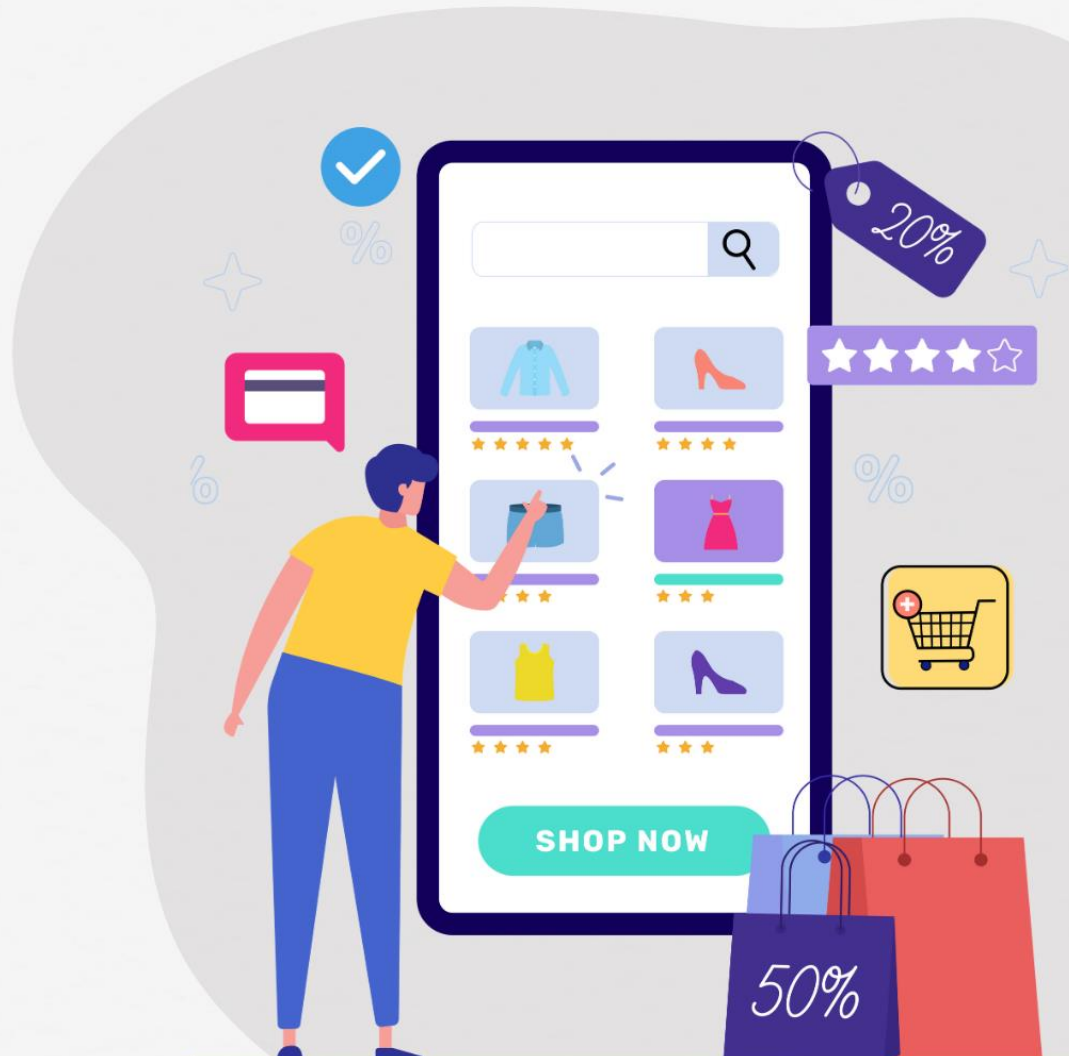


eShopFlix Website

GUIDED PROJECT



Building An eShopFlix Website

Description

The eCommerce industry is booming. In 2018, the global eCommerce market was valued at \$2.8 trillion and is expected to grow to \$4.9 trillion by 2025. With such rapid growth, there is ample opportunity for new entrants to build world-class eCommerce platforms.

Develop an eCommerce platform that adapts to changing consumer preferences. The retail industry constantly needs to introduce new solutions and services that will enhance customer experience and provide a great online shopping experience.

We are looking for a website that enables customers to enjoy a simple, seamless, and effective shopping experience across all stages of their purchasing journey!

Some of the stages of user shopping that you can innovate under include but are not limited to:

1. Public Pages

- Home Page
- About Us Page
- Contact Us Page

2. Account

- Login Page
- SignUp Page
- Forgot Password Page
- SignOut Page

3. Catalogue

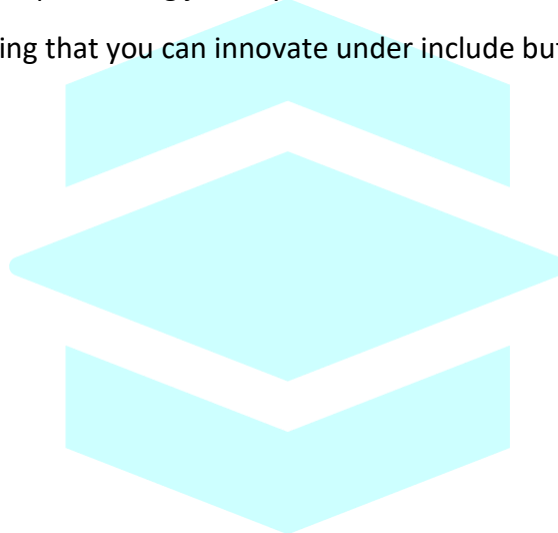
- Catalogue Listing Page
- Catalogue Details Page with AddToCart Option

4. Cart and Payment

- Cart Page
- Checkout Page
- Payment Page with payment gateway integration
- Payment Confirmation Page

5. User Module

- Dashboard Page
- Purchased Items Page
- My Order Page



- Profile Page
- Change Password Page

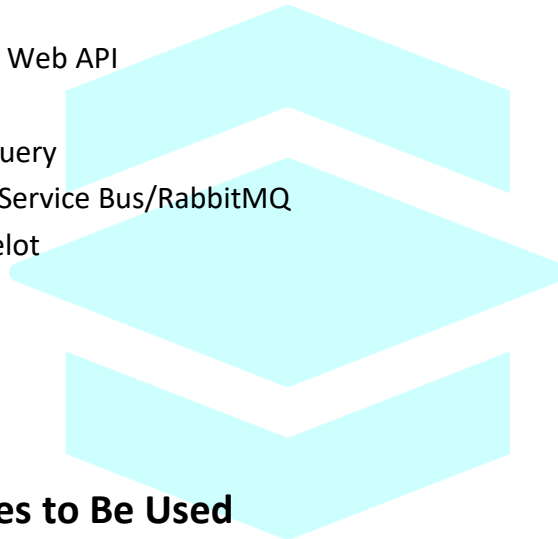
6. Admin Module

- Dashboard Page
- Catalogue Listing, Create and Edit Pages
- Category Listing, Create and Edit Pages
- Received Orders Page
- Profile Page
- Change Password Page

The stages mentioned above are for reference only. While these are some broad highlights, you are encouraged to think out of the box and develop innovative solutions.

Technologies to Be Used

- ASP.NET Core 8: MVC & Web API
- EF Core 8
- Frontend: Bootstrap, jQuery
- Message Broker: Azure Service Bus/RabbitMQ
- API Gateway: APIM/Ocelot
- Docker
- Kubernetes/AKS
- Azure DevOps
- Git



Architecture and Practices to Be Used

- Microservices Architecture
- Domain Driven Design
- Repository Pattern
- Service Pattern
- Dependency Injection
- Authentication and Authorization
- Token Based Security
- Communication Patterns: Sync and Async
- Database Patterns: Saga Pattern
- Cross Cutting Concerns Patterns: Key Vault
- Observability Patterns: Error Logging
- Deployment Patterns: Deployment to AKS using CI/CD

Prerequisites

To get the most from this project, you should be familiar with ASP.NET Core, EF Core, Azure and Microservices Architecture, Bootstrap and Microservices Patterns.

Intended Audience

- .Net Developers
- Senior .NET Developers
- .NET Tech Leads/Project Leads
- .NET Solution Architects

Version History

- Initial Release: 1.0 - 10th Oct 2024



Solution: Building an eShopFlix Project

Requirement Analysis and Planning

Understanding Problem

In today's digital era, the e-commerce industry is thriving as consumers increasingly prefer online shopping for convenience, variety, and accessibility. To compete in this dynamic environment, businesses need a scalable and reliable platform that provides users with seamless shopping experience. Building an e-commerce website involves automating workflows, managing product catalogs, tracking orders, handling payments, and ensuring secure transactions.

This project aims to design and develop a website that automates the workflow of online shopping. The platform will allow customers to browse products, make purchases, and track their orders while also providing administrators with tools to manage inventory, product listings, and orders.

Domain and Sub Domain

In Domain-Driven Design (DDD), the e-commerce platform's domain represents the problem space we are solving. Breaking this domain into subdomains helps in managing various business capabilities. Identifying these subdomains requires a deep understanding of the e-commerce business model, customer behavior, and operational processes.

Each domain consists of multiple subdomains, with each subdomain representing a distinct aspect of the e-commerce ecosystem. These subdomains can be classified into three categories:

- **Core Subdomains:** These are critical features that define the business's unique value proposition. In an e-commerce platform, the core subdomains include product management, order management, and payment processing.
- **Support Subdomains:** These subdomains assist the core operations and include inventory management, customer service, and marketing features like promotions and offers.
- **Generic Subdomains:** These are non-business-specific functionalities that enhance the platform's usability, such as authentication, notifications, and analytics.

Ecosystem

The key actors in our e-commerce ecosystem include:

- **Customers (Users):** Individuals or businesses that browse the platform, make purchases, and track orders.
- **Admin (Platform Owners):** Manage product listings, orders, inventory, and overall system health.
- **Vendors/Sellers:** Suppliers who list their products on the platform and manage their orders.
- **Delivery Partners:** Handle the shipping and delivery of products to customers.

This approach ensures that the project is well-planned and covers all necessary aspects of building a reliable, user-friendly e-commerce platform.

System Design Considerations

The following are system design considerations for building a microservices architecture for an e-commerce project using ASP.NET Core 8, Azure, and Azure DevOps:

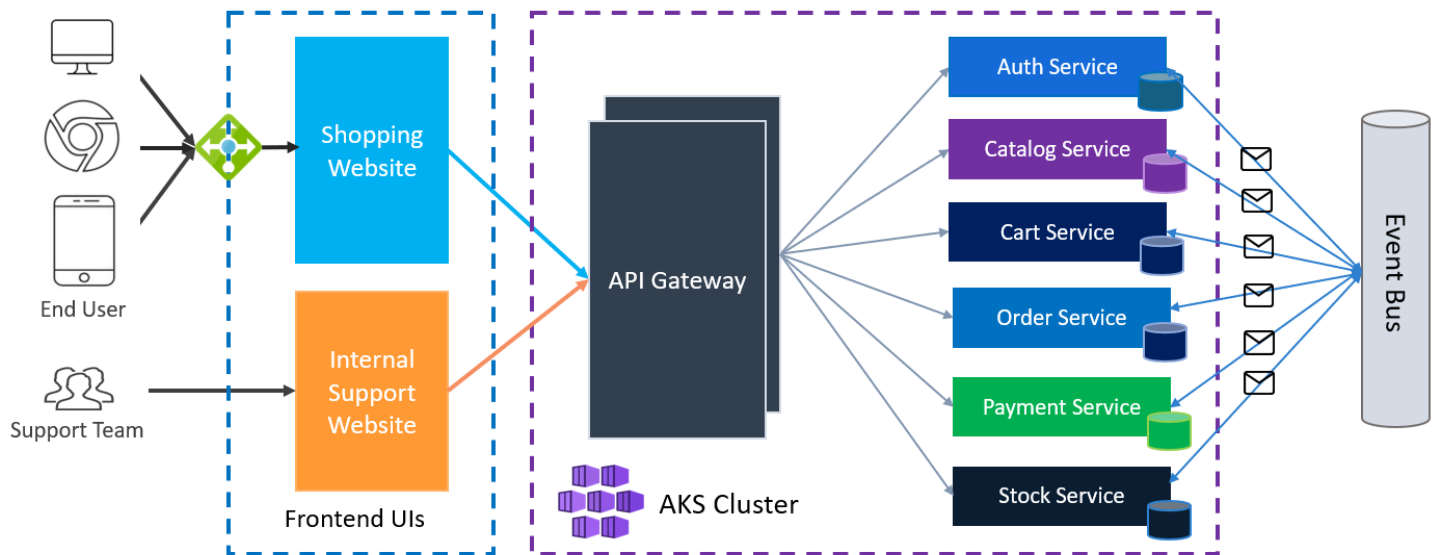
- **High Availability and Low Latency:** Utilize Azure Availability Zones to ensure uptime and minimize latency. Employ Azure Front Door for fast and reliable global access, balancing traffic across regions. Implement caching strategies (e.g., using Azure Cache for Redis) to reduce latency.
- **Scalability:** Leverage Azure Kubernetes Service (AKS) for automatic scaling of microservices. Use Azure Service Bus or Azure Event Grid for handling asynchronous communication between microservices, ensuring the system can scale without performance degradation.
- **Data Consistency and Integrity:** Implement Azure Cosmos DB or Azure SQL Database with strong consistency models for critical data. Use distributed transactions where necessary (e.g., using the Saga pattern) to maintain data integrity across microservices.
- **Redundancy and Failover:** Design microservices to be stateless and leverage Azure Load Balancer for redundancy. Enable Azure Traffic Manager to route traffic during failovers and distribute traffic among regions in case of service failure.
- **Security:** Implement Azure Active Directory (Azure AD) for secure authentication and authorization. Use Azure Key Vault for managing secrets, certificates, and encryption keys. Apply API gateway (like Azure API Management) to centralize security and enforce policies such as rate-limiting and token validation.
- **Content Delivery Network (CDN):** Integrate Azure CDN for faster content delivery, especially for static assets like product images, reducing load on backend services.
- **Efficient Storage:** Use Azure Blob Storage for managing large product catalogs and user data. Leverage Azure SQL Database or Cosmos DB for structured data storage based on scalability and performance needs.
- **System Monitoring:** Implement Azure Monitor and Azure Application Insights to track system performance, diagnose issues, and monitor security. Use Azure Log Analytics for centralized logging and to provide insights into infrastructure and service health.
- **CI/CD with Azure DevOps:** Use Azure DevOps Pipelines for automated build, test, and deployment of microservices, ensuring a smooth and efficient release cycle. Set up Infrastructure as Code (IaC) with Azure Resource Manager (ARM) templates or terraform for consistent and reliable infrastructure deployment.

Project Architecture

A well-designed project architecture is essential for any database-driven application. The architecture defines the overall structure of the project components and how they will communicate with each other. It is the heart of an application and decides the UI and database workflows in the system.

Microservices Architecture

Microservices architecture is a modern software development approach that structures an application as a collection of loosely coupled, independently deployable services. Each service is designed to perform a specific business function and communicates with other services through well-defined APIs. Here are the key components of our microservices architecture for the eShopFlix application:



Frontend Services

1. Shopping Website:

- Interface for end users to browse and purchase products.
- Accessible via web browsers and mobile devices.

2. Internal Support Website:

- Platform for the support team to assist users and manage inquiries.
- Facilitates internal processes and customer support tasks.

Backend Services

1. API Gateway:

- Acts as a single-entry point for all client requests.
- Routes requests to the appropriate backend services.

2. Microservices:

- **Auth Service:** Manages user authentication and authorization.
- **Catalog Service:** Handles product listings and details.
- **Cart Service:** Manages user shopping carts and items.
- **Order Service:** Processes orders and manages order history.
- **Payment Service:** Handles payment processing and transactions.
- **Stock Service:** Manages inventory levels and stock updates.

3. Common Layer:

- **State Machine:** The State Machine keeps track of the current state of a process. For example, "Order Placed," "Payment Processed," "Shipped," and "Delivered."
- **Messages:** Facilitate the exchange of information and coordination needed for processes to function effectively.

4. Event Bus:

- Facilitates communication between microservices through event-driven architecture.
- Ensures asynchronous processing and decoupling of services.

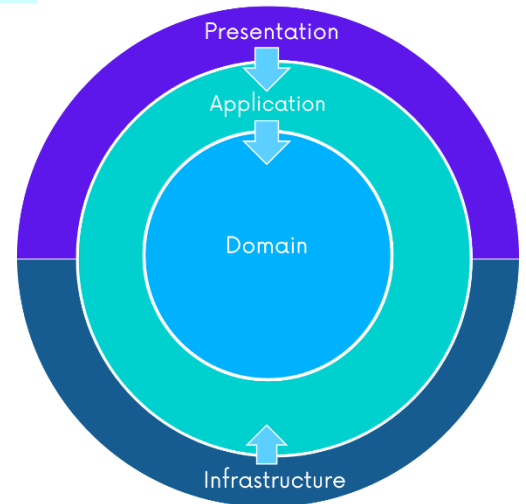
Infrastructure

- **AKS Cluster:** Hosts the backend services, ensuring scalability and management of containerized applications.
- **WebApp/Virtual Machine:** Provides a robust and flexible hosting solution that enables the deployment, scaling, and management of applications in a dynamic and efficient manner.

Service Level Architecture: Clean Architecture

Clean Architecture is often preferred for its principles of separation of concerns, testability, and maintainability, the choice of architecture should align with the specific needs of the microservice, the complexity of the domain, and team expertise.

Clean Architecture, proposed by Robert C. Martin (Uncle Bob), emphasizes separation of concerns by organizing code into layers. The core idea is to keep business logic independent from external concerns like frameworks, databases, and user interfaces.



Architecture Components

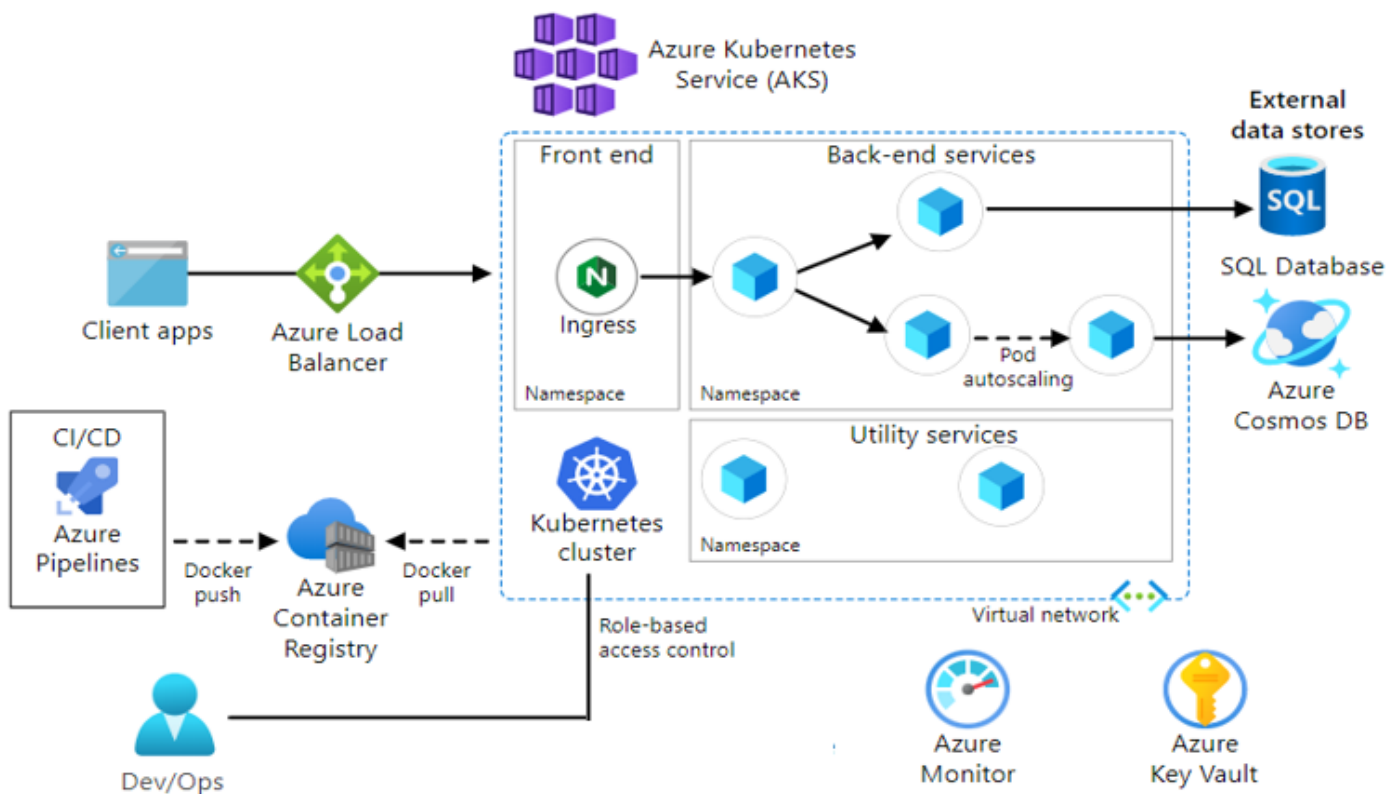
- **Domain Layer (Business Logic):** Business Entities/Objects, Enums, Constants, Repository Interfaces, Domain Events: Order Placed
- **Application Layer (Orchestrator):** DTOs, Application Services, Validations, Service Registration
- **Infrastructure Layer (Persistence):** Repository Implementation, DbContext, Logging, Mappers (EF Entities), Cache and External Services/Providers like APIs, email, messaging, payment
- **Presentation Layer:** Controllers, View/UI, ViewModels, UI Services

Data Flow

- Presentation Layer → Application Layer → Domain Layer
- Application Layer → Infrastructure Layer → DB/External Dependencies
- Infrastructure Layer Implements → Domain Layer Interfaces

AKS Deployment Using CI/CD

Azure Kubernetes Service (AKS) is a managed container orchestration service provided by Microsoft Azure, allowing users to deploy, manage, and scale containerized applications using Kubernetes. Implementing Continuous Integration and Continuous Deployment (CI/CD) pipelines for AKS automates the build, test, and deployment processes, enhancing efficiency, reducing errors, and accelerating the delivery of applications.

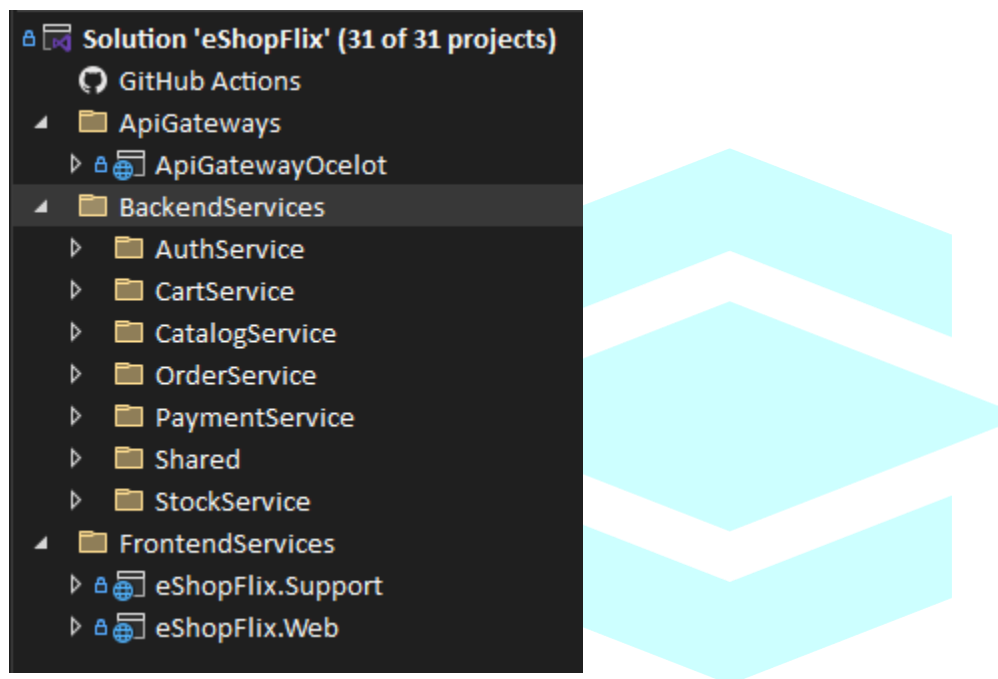


Application Services Structure

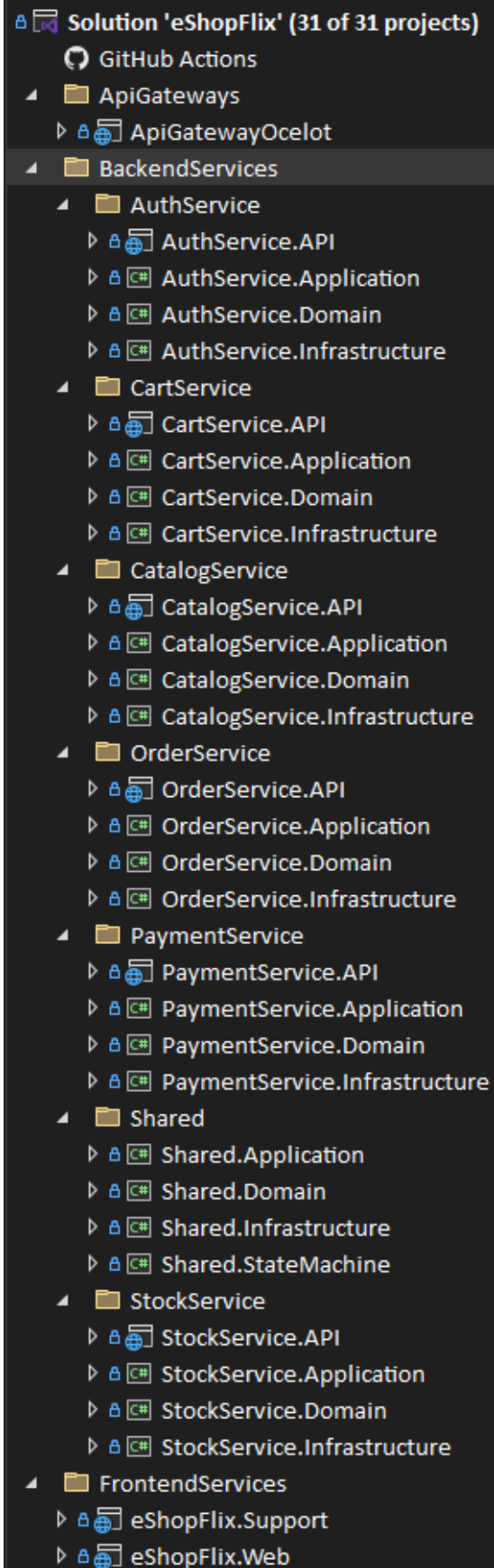
Creating Layers

Project layers help to create a well-organized project architecture. By specifying different types of functionalities in separate layers, it becomes easier to reuse code and keep the project maintainable. Furthermore, Project layers can also help to improve performance by reducing dependencies between different parts of the codebase. When creating project layers, it is important to consider the tradeoffs between flexibility and performance.

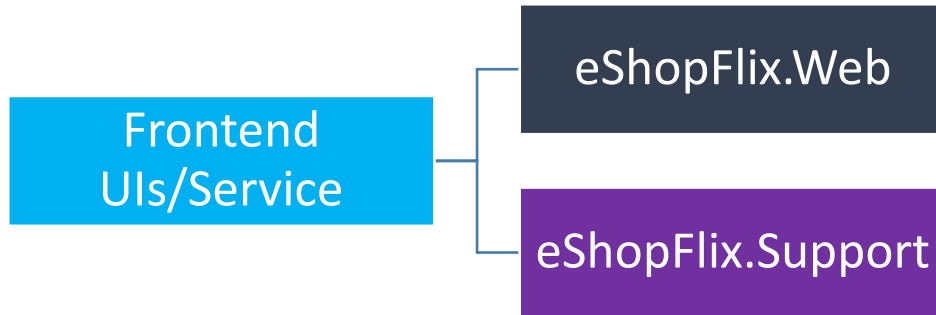
Create the blank solution in visual studio with the name eShopFlix then add the following sections and layers:



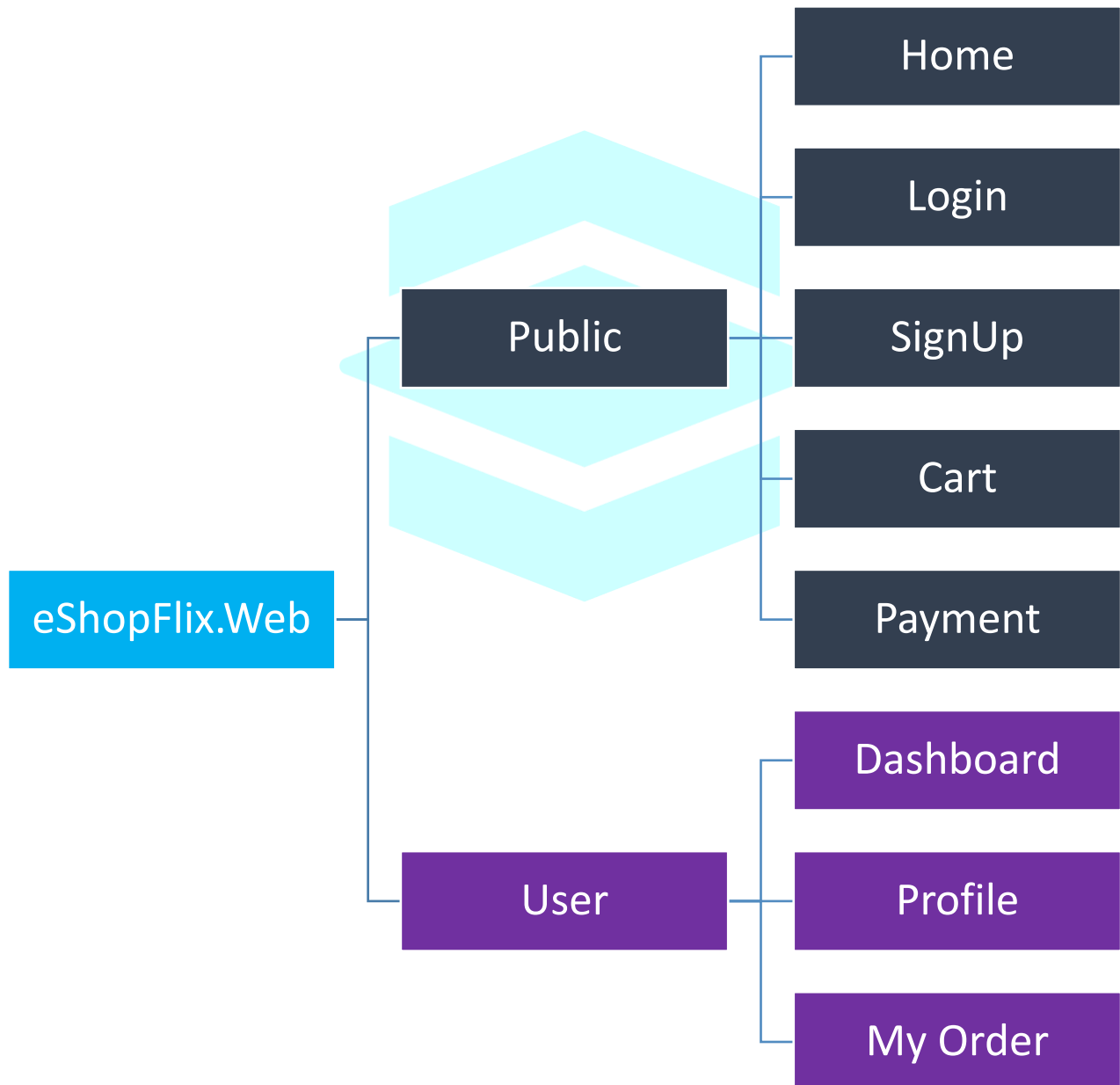
The detailed folder structure for the backend services is given below:

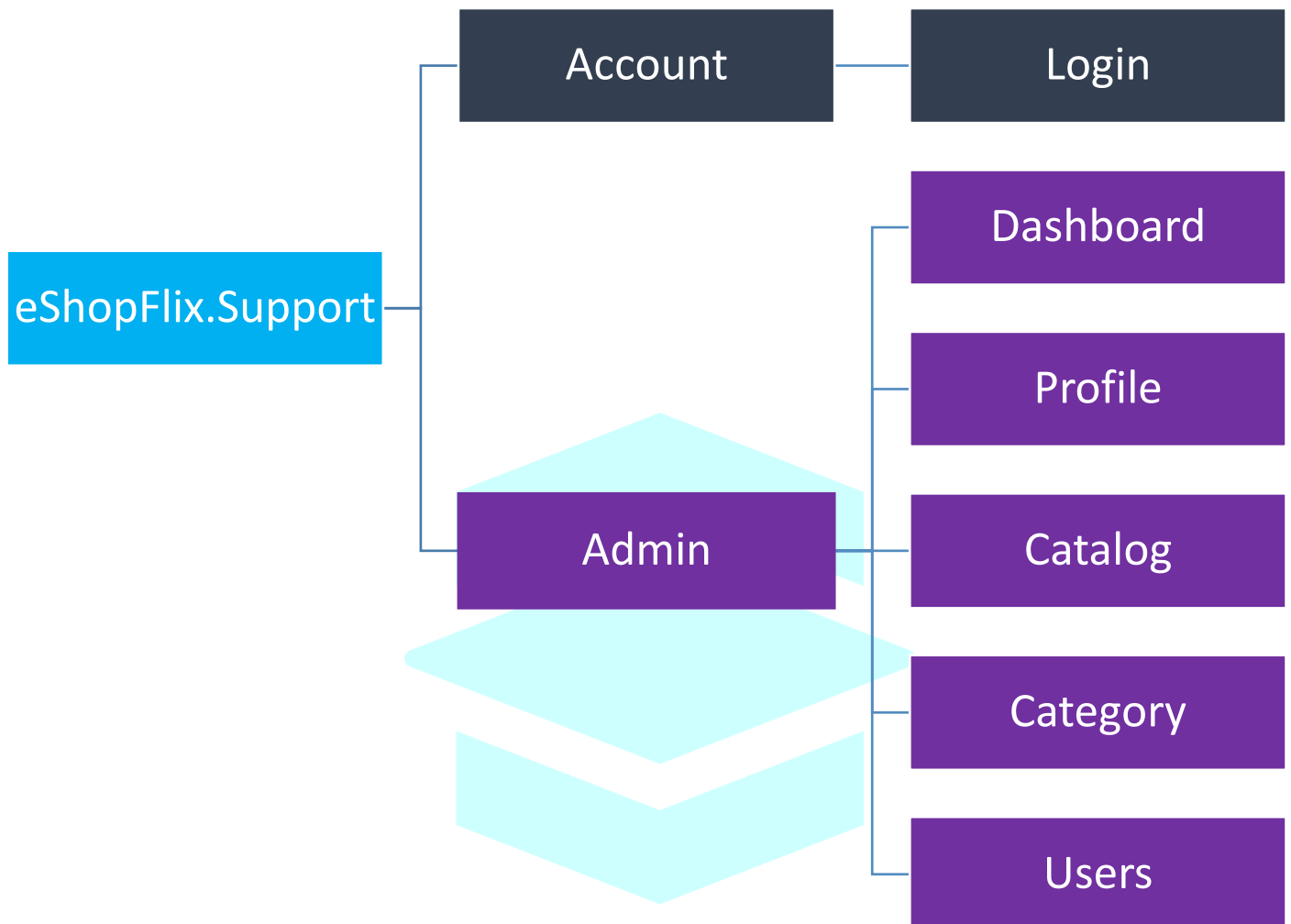


Frontend UIs/Services

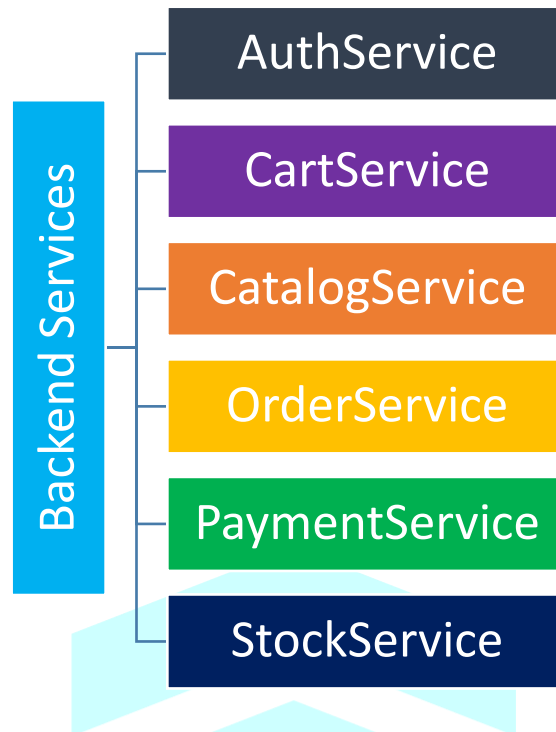


EndUser: eShopFlix.Web



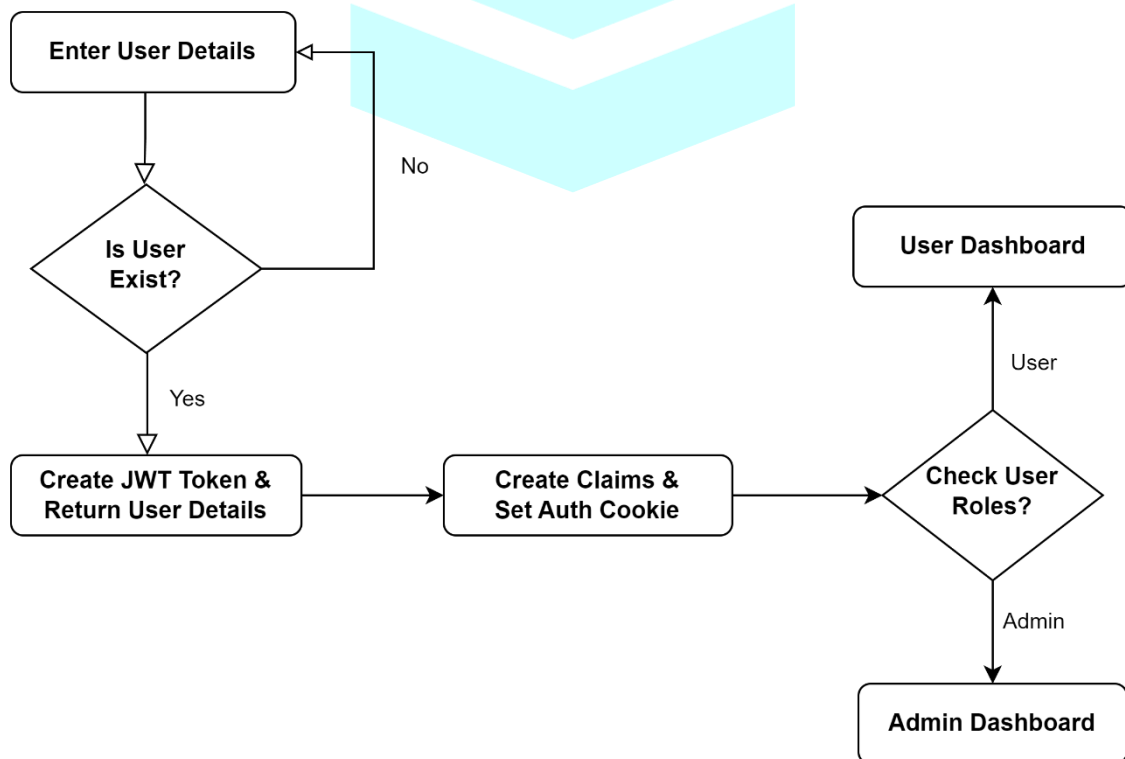


Backend Services

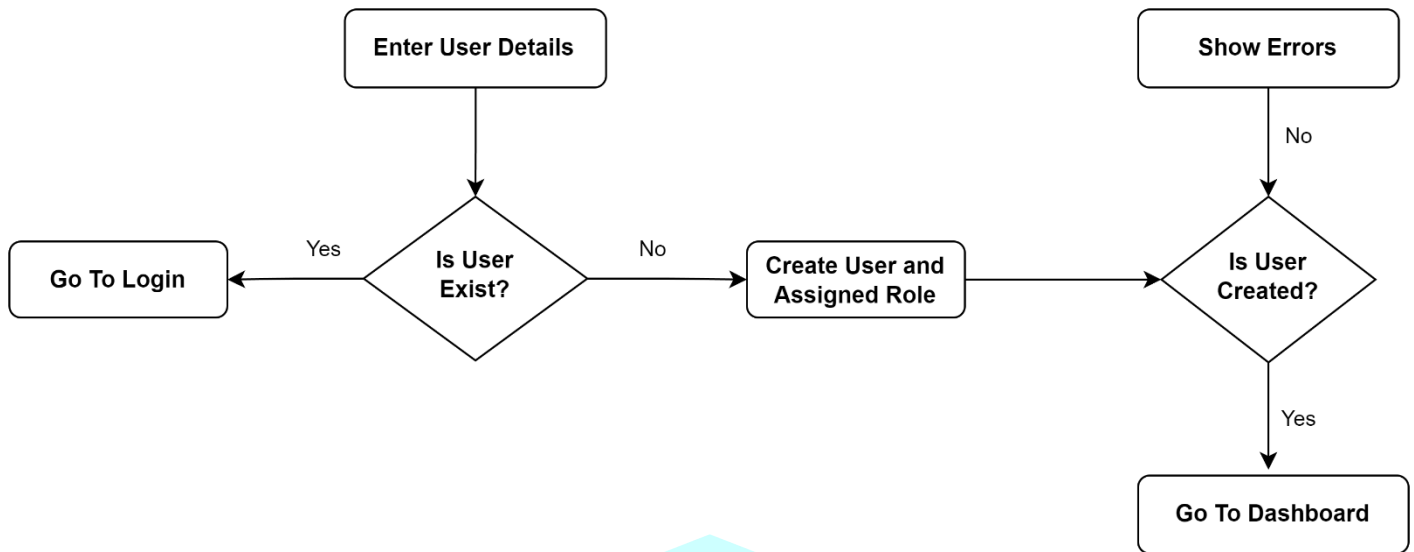


Coding Workflows

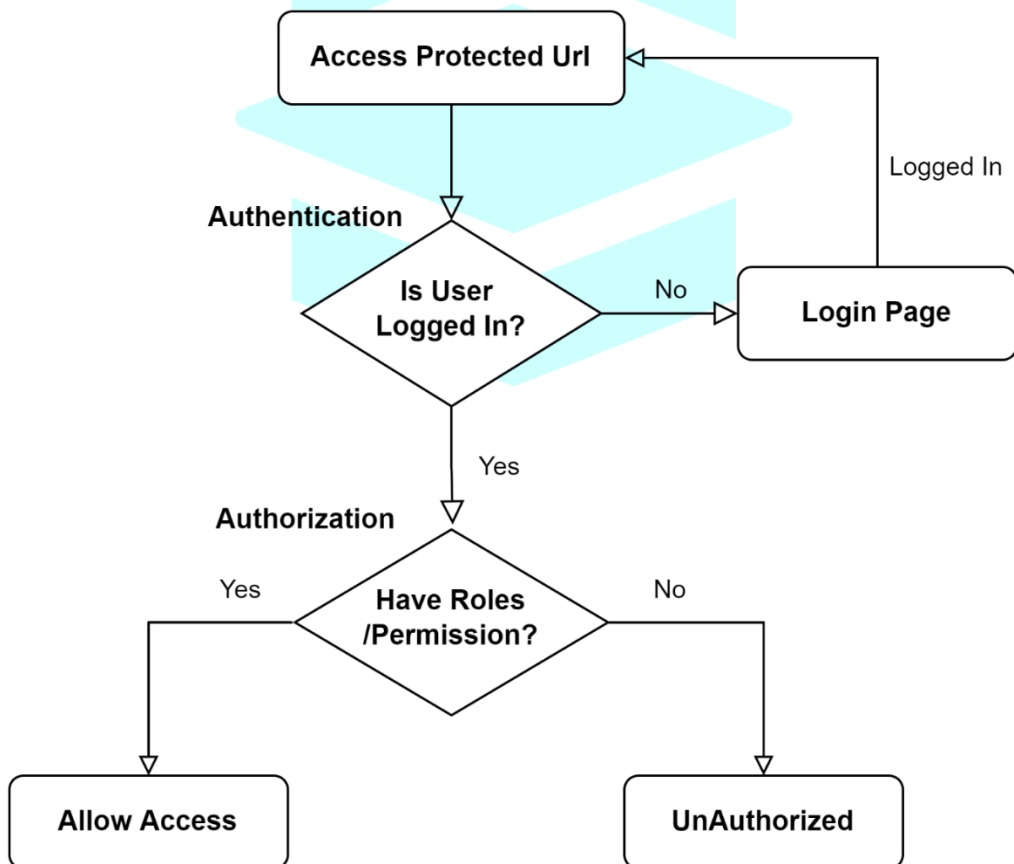
Login Workflow



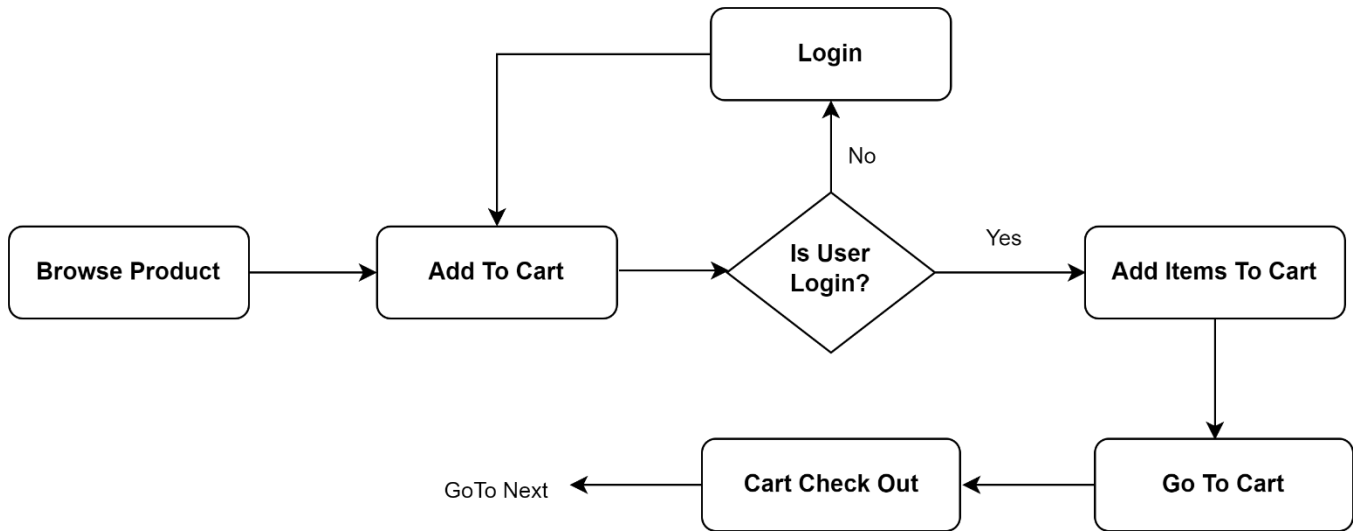
SignUp Workflow



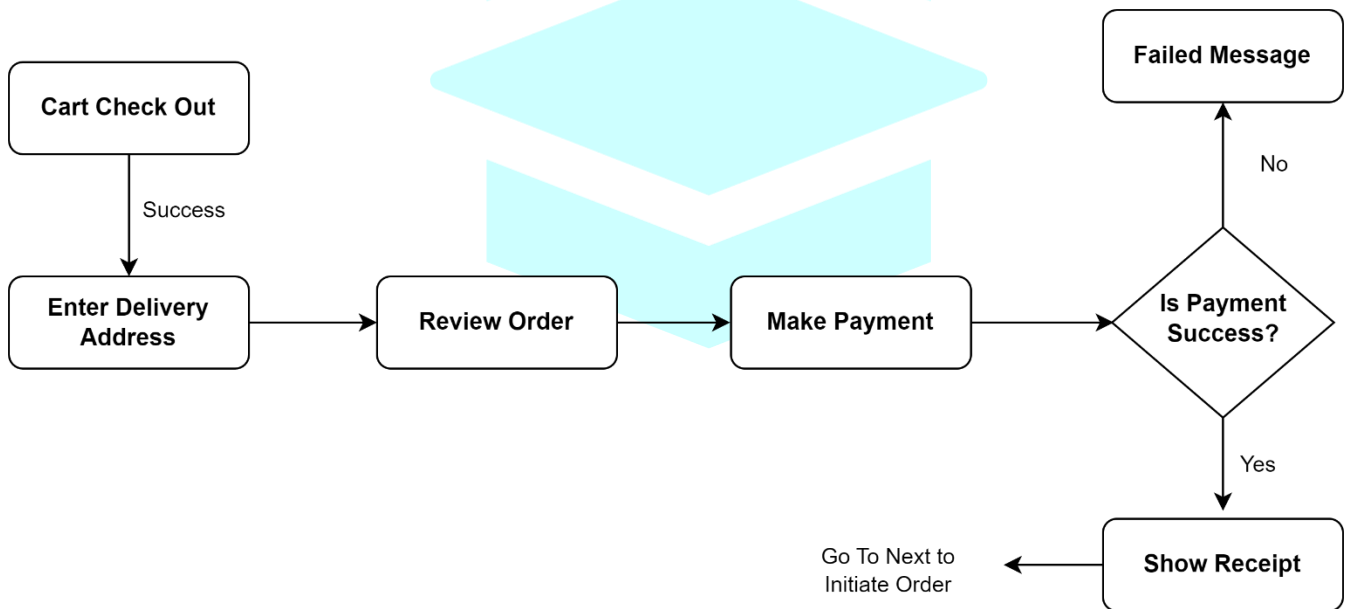
Authentication and Authorization Workflow



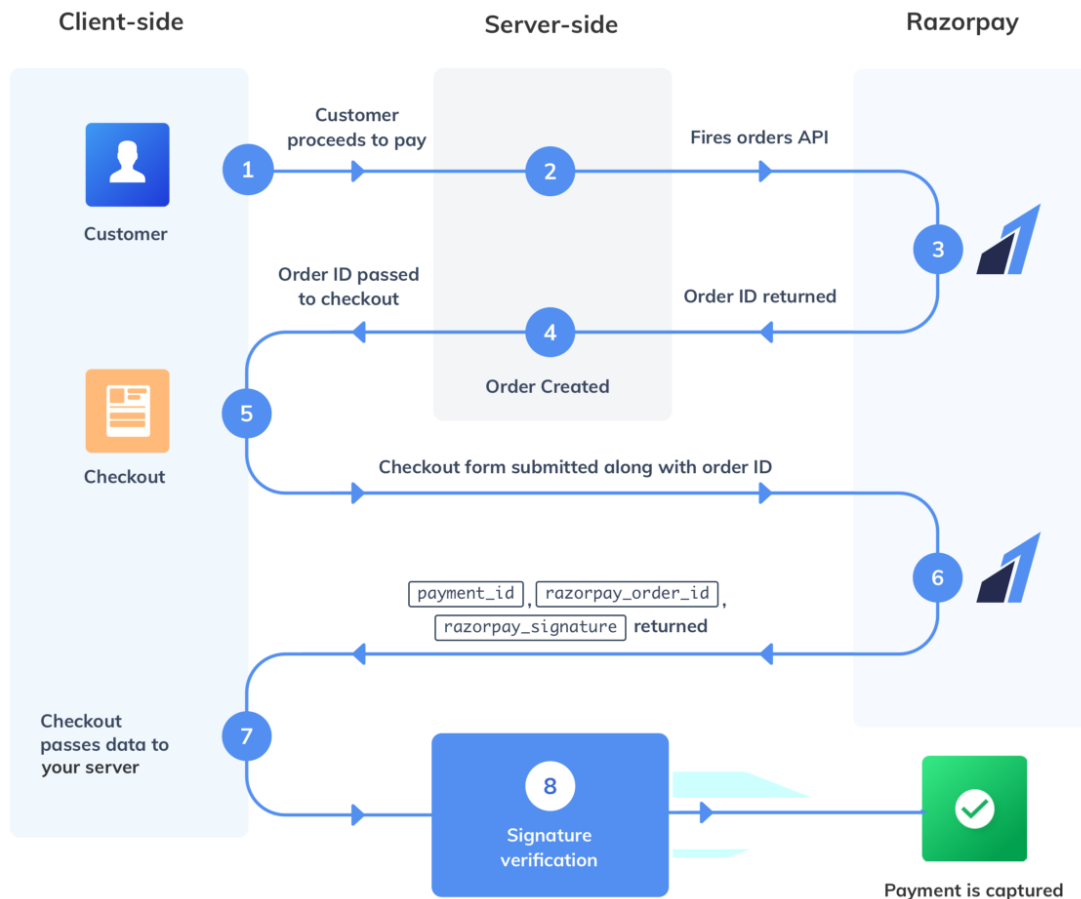
Shopping Cart Workflow



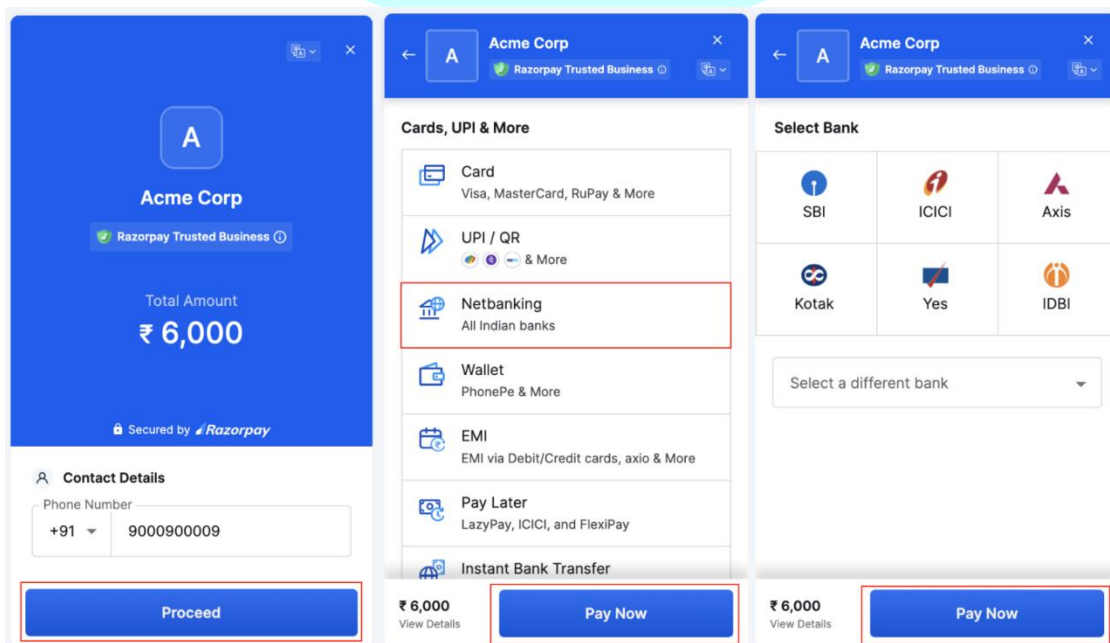
Payment Workflow



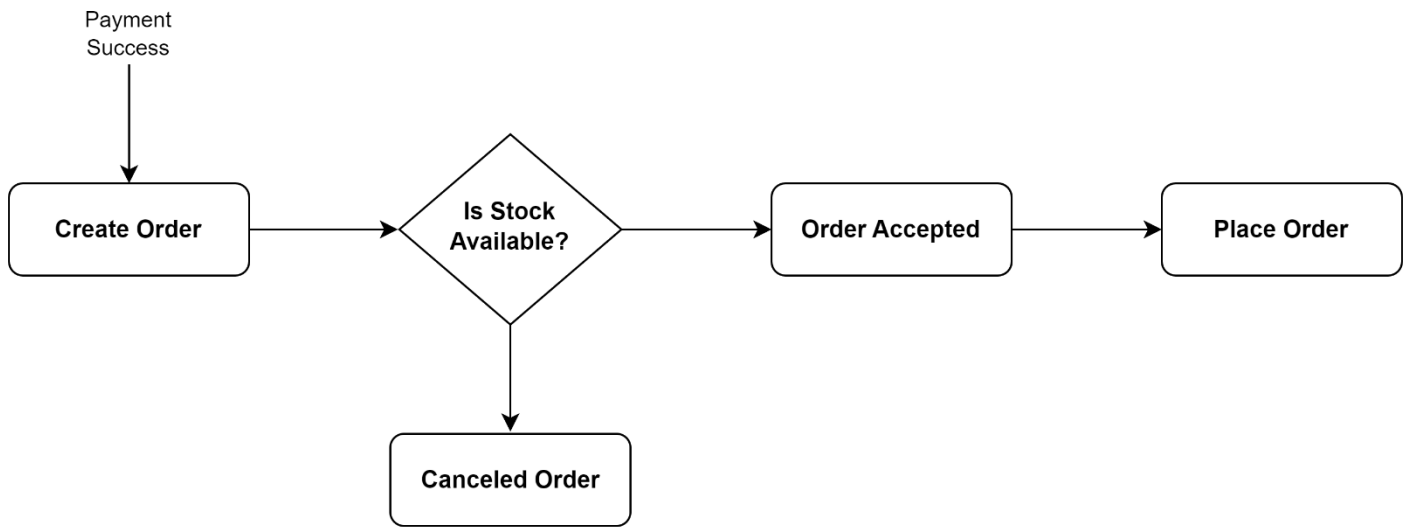
RazorPay Payment Gateway Workflow



RazorPay Payment Gateway UI

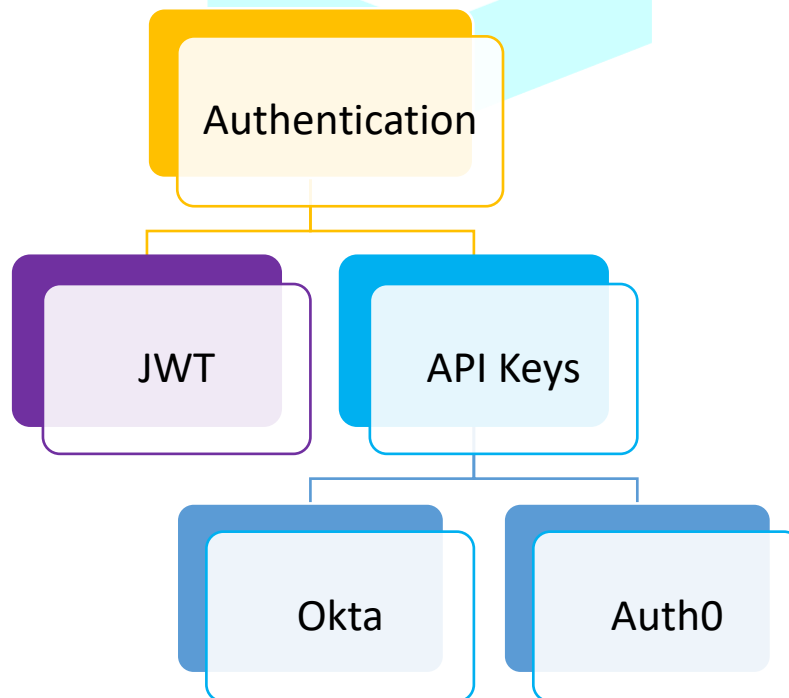


Order Workflow



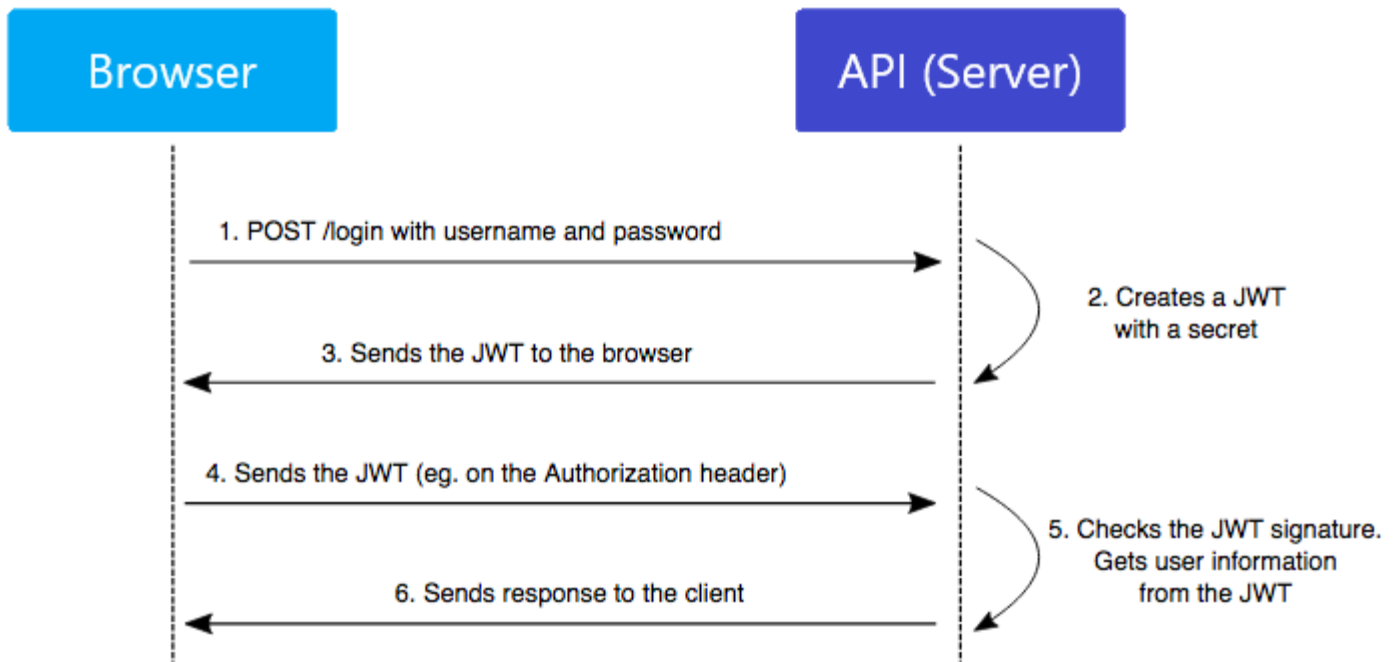
Security in Microservices

In microservices architecture, **ASP.NET Core** provides robust security through **JWT** authentication, enabling services to validate tokens independently without relying on a central server, ensuring scalability. It also integrates with third-party providers like **Okta** and **Auth0** for simplified identity management and single sign-on (SSO). Additionally, ASP.NET Core allows customization of authentication, supporting varied security needs across different services, making it ideal for securing microservices.



JWT Authentication

JSON Web Token (JWT) is an authentication method that uses a token to prove that a user is authenticated. The token is generated by the server and is typically sent to the client in the HTTP header. Once the user is authenticated, they can access resources that are protected by JWT.



JWT can be used with any type of application, but it is most commonly used with Single Page Applications (SPAs). SPAs are web applications that run in the browser and do not require a full-page refresh when the user navigates to different parts of the app. JWT is also used with APIs to authenticate users. When a user authenticates with a JWT-protected API, they will receive a token that they can use to access resources that are protected by JWT. JWT is a convenient and secure way to authenticate users, and it can be used with any type of application.

Creating JWT Token

```
private string GenerateJSONWebToken(UserModel userInfo)
{
    var securityKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey,
SecurityAlgorithms.HmacSha256);

    var claims = new[] {
        new Claim(JwtRegisteredClaimNames.Sub, userInfo.Name),
        new Claim(JwtRegisteredClaimNames.Email, userInfo.Email),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

    var token = new JwtSecurityToken(_issuer, _audience, claims, expires: DateTime.Now.AddHours(1),
    signingCredentials: credentials);

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

```

};

var token = new JwtSecurityToken(_config["Jwt:Issuer"],
                                _config["Jwt:Audience"],
                                claims,
                                //token expiry minutes
                                expires: DateTime.UtcNow.AddMinutes(60),
                                signingCredentials: credentials);

return new JwtSecurityTokenHandler().WriteToken(token);
}

```

Authentication and Authorization

Authentication and Authorization are powerful ways to control access to your web application's protected pages or URLs. By specifying which roles are allowed to access specific resources, you can ensure that only authorized users can access protected pages. In ASP.NET Core `IAuthorization` filter can help you to implement your custom authentication and authorization logic.

Validating JWT Token

```

var catalogAuthKey = builder.Configuration["Keys:CatalogAuthKey"];

builder.Services.AddAuthentication().AddJwtBearer(catalogAuthKey, options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = builder.Configuration["Jwt:Issuer"],
        ValidAudience = builder.Configuration["Jwt:Audience"],
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
    };
});

```

```
{
```

```

"DownstreamPathTemplate": "/api/catalog/{everything}",
"DownstreamScheme": "https",
"DownstreamHostAndPorts": [
  {
    "Host": "localhost",
    "Port": 7233
  }
],
"AuthenticationOptions": {
  "AuthenticationProviderKeys": "eShopFlixCatalogKey@12345678"
},
"RouteClaimsRequirement": {
  "Roles": "Admin"
},
"UpstreamPathTemplate": "/catalog/{everything}",
"UpstreamHttpMethod": ["Post", "Put", "Delete"]
}

```

Microservices Patterns

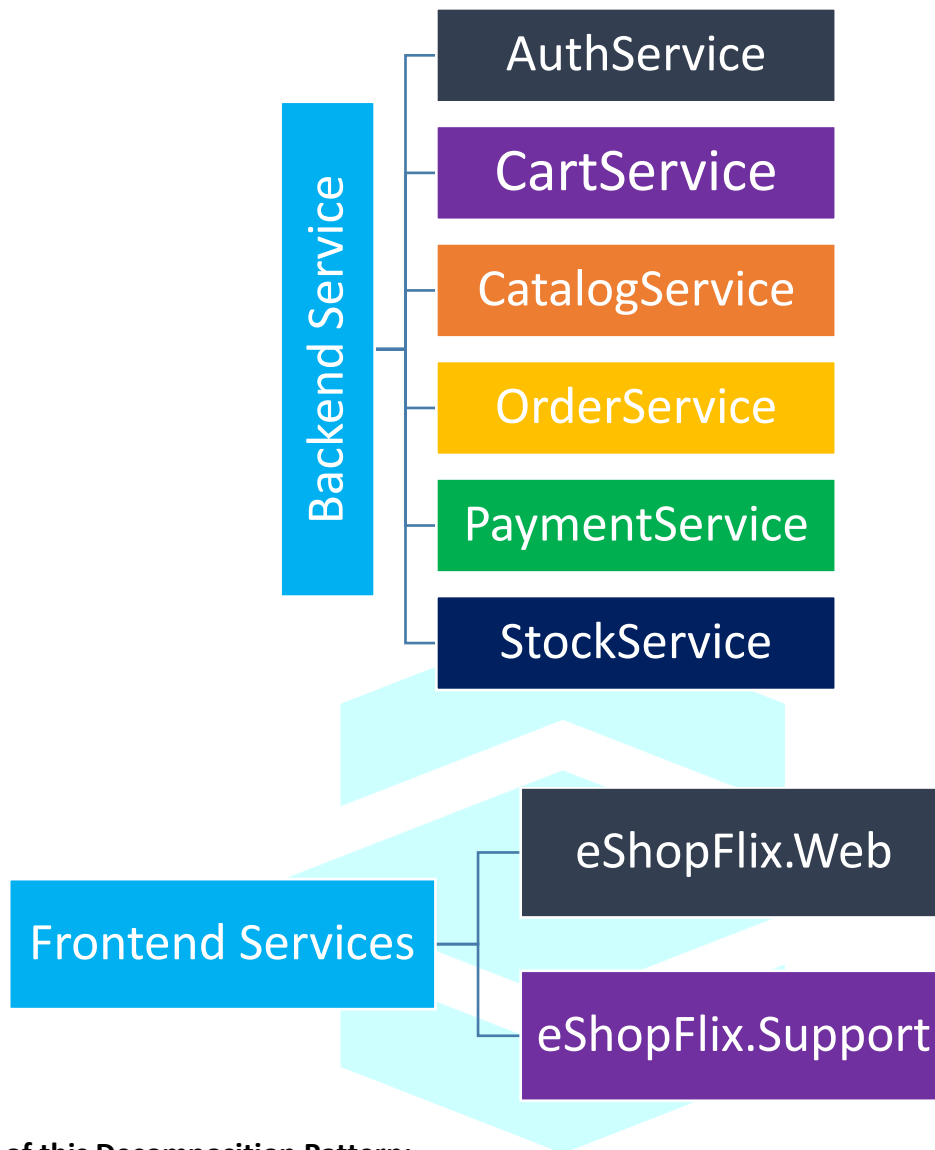
Decomposition Patterns:

The decomposition pattern followed by the eShopFlix folder structure is based on the **service-oriented decomposition** pattern, typical in a **microservices-based architecture**. The structure highlights the principle of **vertical slice decomposition**, where each business capability is isolated into its own service, maintaining a clear boundary between the various components of the application. Below is a detailed description:

Decomposition Pattern in the Folder Structure

1. **Service-Oriented Decomposition:** The folder structure shows a clear separation of services based on business domains such as AuthService, CartService, CatalogService, and PaymentService. Each service encapsulates a specific domain responsibility, making it independent of other services, following the microservices principles. Each service can be developed, deployed, and scaled independently, aligning with the **service-oriented decomposition** pattern.
2. **Layered Architecture Within Each Service:** Each service is organized into separate layers:
 - **API:** The API layer in each service (e.g., Auth.API, CartService.API) is responsible for exposing endpoints and handling HTTP requests, making each service interactable with external systems or other services.

- **Application:** The Application layer contains the business logic of each service (e.g., Auth.Application, CartService.Application), ensuring that the services are self-contained and handle domain-specific operations independently.
 - **Domain:** The Domain layer includes the core domain models, business entities, and interfaces, following **domain-driven design (DDD)** principles. This layer represents the business logic and data that are central to the service.
 - **Infrastructure:** The infrastructure layer deals with external resources such as databases, messaging systems, or third-party services (e.g., Auth.Infrastructure, CartService.Infrastructure). This separation ensures that the infrastructure concerns are decoupled from the business logic.
3. **API Gateway Pattern:** The inclusion of ApiGatewayOcelot highlights the usage of the **API Gateway pattern**, which aggregates requests from different clients and forwards them to the respective backend services. This acts as a single-entry point for frontend applications, handling concerns like routing, load balancing, and security.
 4. **Shared Module:** The Shared module contains common infrastructure or functionality that is reusable across different services. While having a shared module helps avoid code duplication, it is essential to ensure that it doesn't lead to tight coupling between services, which is often avoided in microservices to maintain service independence.
 5. **Frontend and Backend Separation:** The folder structure divides the project into BackendServices and FrontendServices. FrontendServices (e.g., WebApp) handles the UI and interaction layer, while BackendServices (e.g., AuthService, CartService) handle specific domain logic. This separation promotes better maintainability and scalability as the UI can evolve independently from the backend services.
 6. **Vertical Slice Decomposition:** Each service is an independent vertical slice of the system, encapsulating everything from the database and domain models to business logic and API, making each service self-sufficient. This pattern allows services to evolve independently of one another.



Advantages of this Decomposition Pattern:

- **Service Independence:** Each service is self-contained and follows the single responsibility principle. It ensures that changes in one service don't directly affect others.
- **Scalability:** Services can be scaled independently depending on the load or demand on specific business domains.
- **Ease of Deployment:** Each service can be deployed independently without impacting on other parts of the system.
- **Fault Isolation:** Failures in one service don't cascade to others, improving system resilience.
- **Domain-Driven Design:** The structure allows clear adherence to **domain-driven design (DDD)** by organizing code into domains and separating concerns within each domain.

Integration Patterns

An **integration pattern** in microservices refers to the approach or design used to facilitate communication, coordination, and data sharing between multiple microservices. Since microservices are typically small,

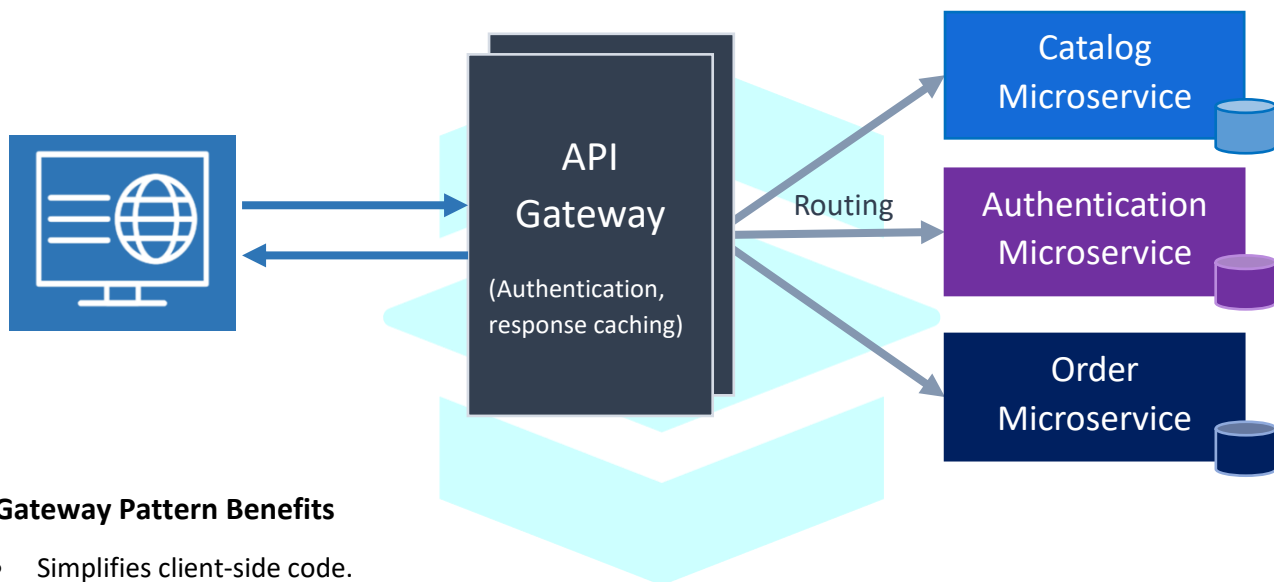
independently deployable services that communicate over the network, integration patterns ensure that these services interact efficiently without causing tight coupling or performance bottlenecks.

Here's an explanation of the implemented integration patterns used in the above microservices architecture:

API Gateway Pattern

The API Gateway serves as the single-entry point for all client requests, which it then routes to the appropriate backend services (e.g., AuthService, CartService, CatalogService, PaymentService).

- It handles cross-cutting concerns like authentication, logging, rate-limiting, and load balancing, abstracting the complexities of interacting with multiple services from the client.
- The API Gateway ensures that the frontend (WebApp) does not need to interact directly with individual services but instead communicates through a unified gateway, which forwards requests to the relevant services based on routing rules.



API Gateway Pattern Benefits

- Simplifies client-side code.
- Centralizes concerns like security and throttling.
- Reduces round trips by aggregating multiple backend services into a single request.

Communication Patterns

In microservices architecture, **communication patterns** define how microservices interact with each other and external systems. Since microservices are designed to be loosely coupled and independently deployable, choosing the right communication pattern is crucial for ensuring scalability, reliability, and performance.

The key **communication patterns** implemented in this structure are described below.

1. Synchronous Communication via REST/HTTP (Request-Response Pattern)

In synchronous communication, the client waits for the response after sending a request to the microservice. This pattern is useful for real-time, request-response-based interactions where one service requires immediate data from another service.

For example, when the **CartService** needs to get item details like name, image, it make a REST API call to the **CatalogService**. In this case:

- The **CartService** sends an HTTP request to the **CatalogService**.
- The **CatalogService** processes the request and responds with the result.
- The **CartService** then continues its execution based on the response.

Advantages:

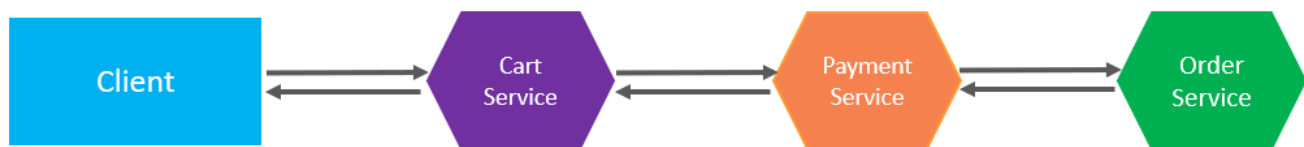
- Simple to implement using HTTP/REST, especially in ASP.NET Core.
- Widely understood and supported by modern frameworks.
- Effective for real-time, direct interactions.

2. Asynchronous Communication via Event-Driven Messaging

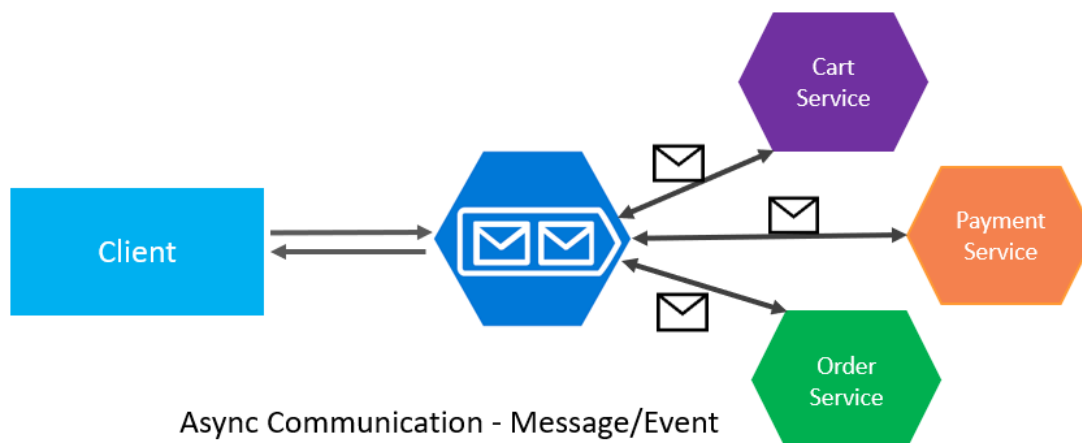
In asynchronous communication, the client does not wait for an immediate response. Instead, services interact via message-passing mechanisms like message brokers, events, or queues (e.g., RabbitMQ, Azure Service Bus).

For example:

- The **OrderService** can publish an event like **StockValidationRequested**, which is consumed by the **StockService** to initiate the stock validation process.
- After processing, the **StockService** publishes an event such as **OrderAccepted** if the stock is available, or **OrderCancelled** if the stock is not available.
- This communication is **asynchronous**: The **OrderService** does not need to wait for a response from the **StockService** and can continue executing other tasks after publishing the initial event.



Sync Communication - Request/Reply



Async Communication - Message/Event

Advantages:

- **Loose Coupling:** Services are not dependent on the availability of other services.
- **Scalability:** Easier to scale and handle a high volume of events across multiple services.
- **Fault Tolerance:** Since services are decoupled, failures in one service don't immediately affect others.

Database Patterns

In microservices architecture, each service typically has its own database to maintain loose coupling and ensure independent scalability. Database patterns help manage data across services while addressing concerns like data consistency, scalability, and fault tolerance.

1. Database per Service Pattern

Each microservice manages its own database, enforcing the principle of data ownership and independence. This ensures that services can be updated, deployed, and scaled without impacting others. For example:

- **UserService** manages its user profile database.
- **OrderService** manages its order history database.

Advantages:

- **Loose Coupling:** Services can operate independently, reducing the risk of cascading failures.
- **Scalability:** Each service can be scaled individually depending on its database load.
- **Technology Flexibility:** Different services can use different types of databases, such as SQL, NoSQL, or in-memory databases, depending on their specific needs.

2. Shared Database Pattern

In some cases, services may share a common database, especially when services are tightly related or when transitioning from a monolithic architecture. For example, both **StockService** and **InventoryService** might access the same database to manage product stock.

Advantages:

- **Easier Transaction Management:** Cross-service transactions are simpler since services access the same database.
- **Reduced Overhead:** Shared databases reduce the complexity of managing multiple databases.

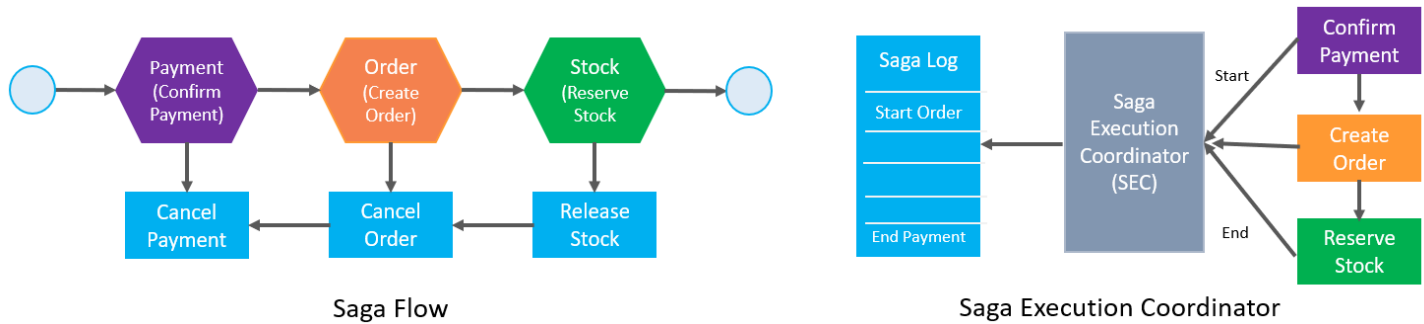
3. Saga Pattern (Distributed Transactions)

The Saga pattern manages distributed transactions across multiple services without relying on a single database transaction. A series of local transactions are performed by each service, and compensating actions are executed to roll back the process in case of a failure. For example:

- **OrderService** initiates an order.
- **PaymentService** processes payment.
- **StockService** updates inventory. If the stock is unavailable, the saga compensates by canceling the payment and rolling back the order.

Advantages:

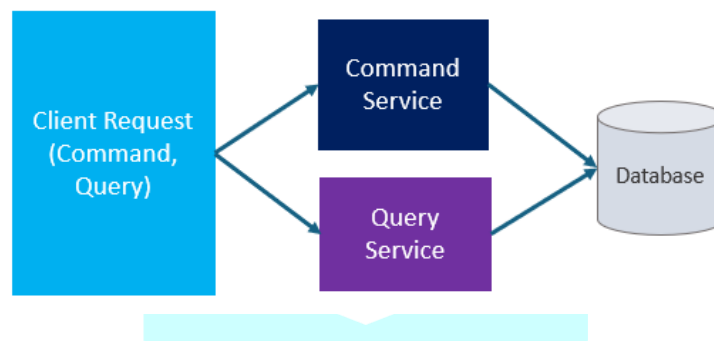
- **Maintains Data Consistency:** Ensures data integrity across multiple services without needing a centralized database.
- **Fault Tolerance:** Handles failures gracefully through compensating transactions.



4. CQRS (Command Query Responsibility Segregation) Pattern

CQRS separates read and write operations into different models, optimizing data management for performance and scalability. For example:

- **OrderService** uses a write model to handle order creation and updates.
- **ReportingService** uses a read model to retrieve order information for analytics purposes.



Advantages:

- **Performance Optimization:** Read and write operations can be optimized separately.
- **Scalability:** Each model can be scaled independently, depending on the demand for read or write operations.

Cross-Cutting Concerns Patterns

Cross-cutting concerns refer to features that affect multiple microservices across an application. These concerns, such as authentication, authorization, logging, and error handling, are essential for the smooth operation of the system but can complicate individual service logic. To manage these concerns, reusable patterns are employed to simplify implementation and ensure consistency across microservices.

1. Externalized Configuration Pattern

The externalized configuration pattern refers to storing configuration settings (e.g., database connections, API keys, feature flags) outside the service's codebase, usually in a centralized location. This makes it easier to manage configuration changes across environments (development, testing, production) without redeploying services.

Example: In **Spring Boot**, you can externalize configuration properties in application.properties or via a centralized config server like **Spring Cloud Config**. Azure or AWS provide key management services to store secrets and configuration.

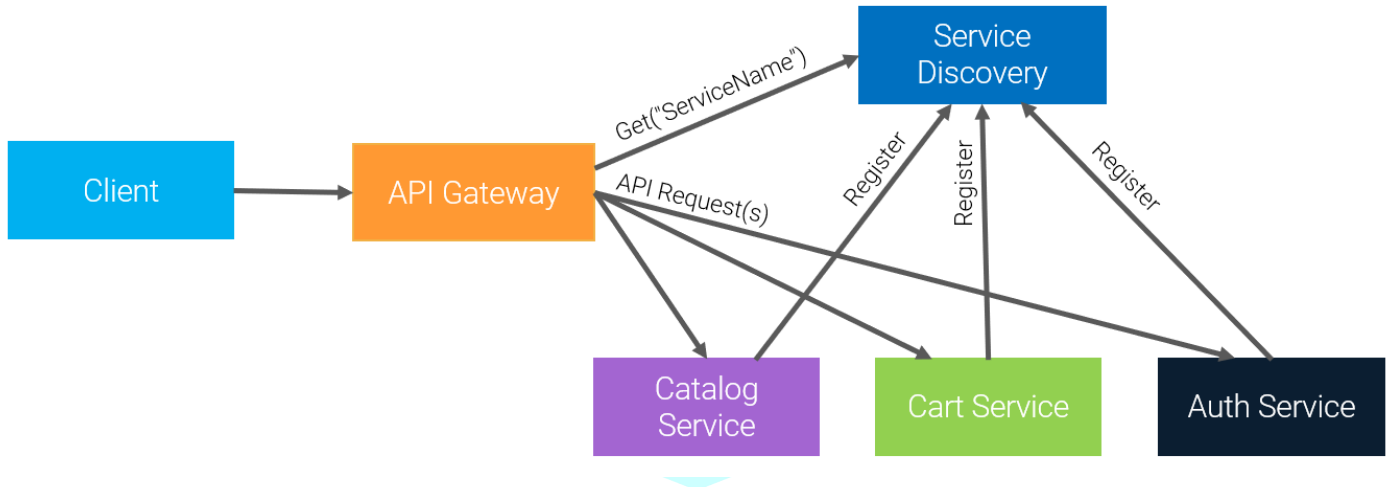
Advantages:

- Simplifies environment-specific configuration.
- Reduces the risk of exposing sensitive information in code.
- Allows for real-time updates to configuration without redeployment.

2. Service Discovery Pattern

In a dynamic microservices environment, services frequently scale up or down, and their instances may change network locations (IP addresses, ports). The service discovery pattern ensures that services can dynamically locate and communicate with each other by querying a service registry.

Example: Tools like **Consul**, **Eureka**, or **Kubernetes DNS** provide service discovery mechanisms where services register themselves and other services can discover them when needed.



Advantages:

- Facilitates dynamic scaling and load balancing.
- Reduces the need to hardcode service endpoints.
- Ensures more resilient inter-service communication.

3. Circuit Breaker Pattern

The circuit breaker pattern helps prevent service failures from cascading through the system by temporarily halting communication with failing services. When a service fails repeatedly, the circuit breaker "opens," and requests are redirected or handled by fallback mechanisms. For example:

- If **PaymentService** is down, the circuit breaker prevents further requests from reaching it and may trigger a fallback response in **OrderService**.

Advantages:

- **Failure Isolation:** Protects services from cascading failures, improving overall system resilience.
- **Graceful Degradation:** Allows services to degrade gracefully by providing fallback options when a dependency fails.
- **Improved Availability:** Helps prevent system-wide downtime by limiting the impact of individual service failures.

4. Rate Limiting Pattern

The rate limiting pattern controls the number of requests that can be processed by a microservice over a given period. It prevents overloads and protects services from being overwhelmed by high traffic or malicious requests. This pattern is typically implemented at the API Gateway or service level. For example:

- **OrderService** may have a rate limit that restricts each user to 100 order requests per hour to prevent abuse.

Advantages:

- **Prevents Overload:** Protects services from being overwhelmed by too many requests, ensuring stable operation.
- **Improved Security:** Helps defend against DDoS (Distributed Denial of Service) attacks by limiting traffic.
- **Cost Control:** Reduces the risk of overconsumption of resources, which can lead to higher operational costs.

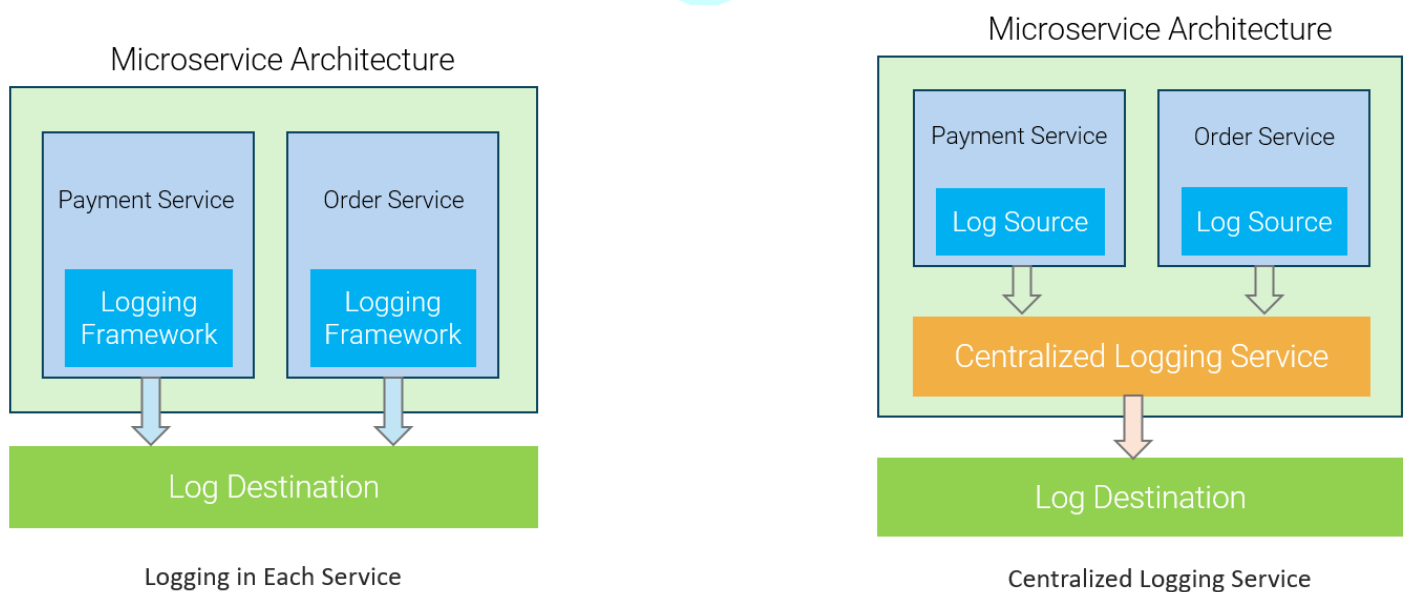
Observability Patterns

Observability in microservices architecture is essential for monitoring, troubleshooting, and improving the system's health and performance. Since microservices are distributed across multiple services, achieving full observability ensures that system failures and performance bottlenecks can be detected and resolved quickly.

1. Centralized Logging Pattern

In microservices, each service generates its own logs, which can become difficult to manage and trace across different services. The centralized logging pattern consolidates logs from all microservices into a central repository, allowing for easier analysis and debugging. For example:

- Services like **OrderService** and **PaymentService** send their logs to a centralized logging system like **ELK Stack** (Elasticsearch, Logstash, and Kibana).



Advantages:

- **Easier Debugging:** Logs from different services are available in one place, making it easier to trace errors across services.
- **Searchable Logs:** Centralized systems provide powerful querying and filtering to search logs quickly.
- **Improved Monitoring:** Helps track service-level issues and failures in real time.

2. Health Check Pattern

The health check pattern ensures that each microservice reports its health status regularly. Health checks typically involve checking the status of dependencies like databases, message brokers, or third-party APIs. Services expose health check endpoints (e.g., /health) that can be monitored by external tools. For example:

- **InventoryService** might expose a /health endpoint to check the status of its connection to the database and messaging system.

Advantages:

- **Automatic Monitoring:** Helps detect failing or unhealthy services.
- **Improved Availability:** Enables proactive detection of issues before they affect the system.
- **Seamless Recovery:** Health checks can trigger auto-recovery processes like restarting or rescaling the service.

3. Metrics Collection Pattern

Metrics collection involves gathering performance-related data from microservices, such as CPU usage, memory consumption, request rates, and error rates. These metrics are pushed to monitoring systems like **Prometheus** or **Grafana** for real-time visualization and analysis. For example:

- **UserService** might send metrics related to login failures, API response times, and throughput to a monitoring system.

Advantages:

- **Performance Monitoring:** Metrics help assess the overall performance of the system and individual services.
- **Anomaly Detection:** Alerts can be set based on thresholds for various metrics, such as CPU utilization exceeding 80%.
- **Capacity Planning:** Helps identify resource usage trends for scaling decisions.

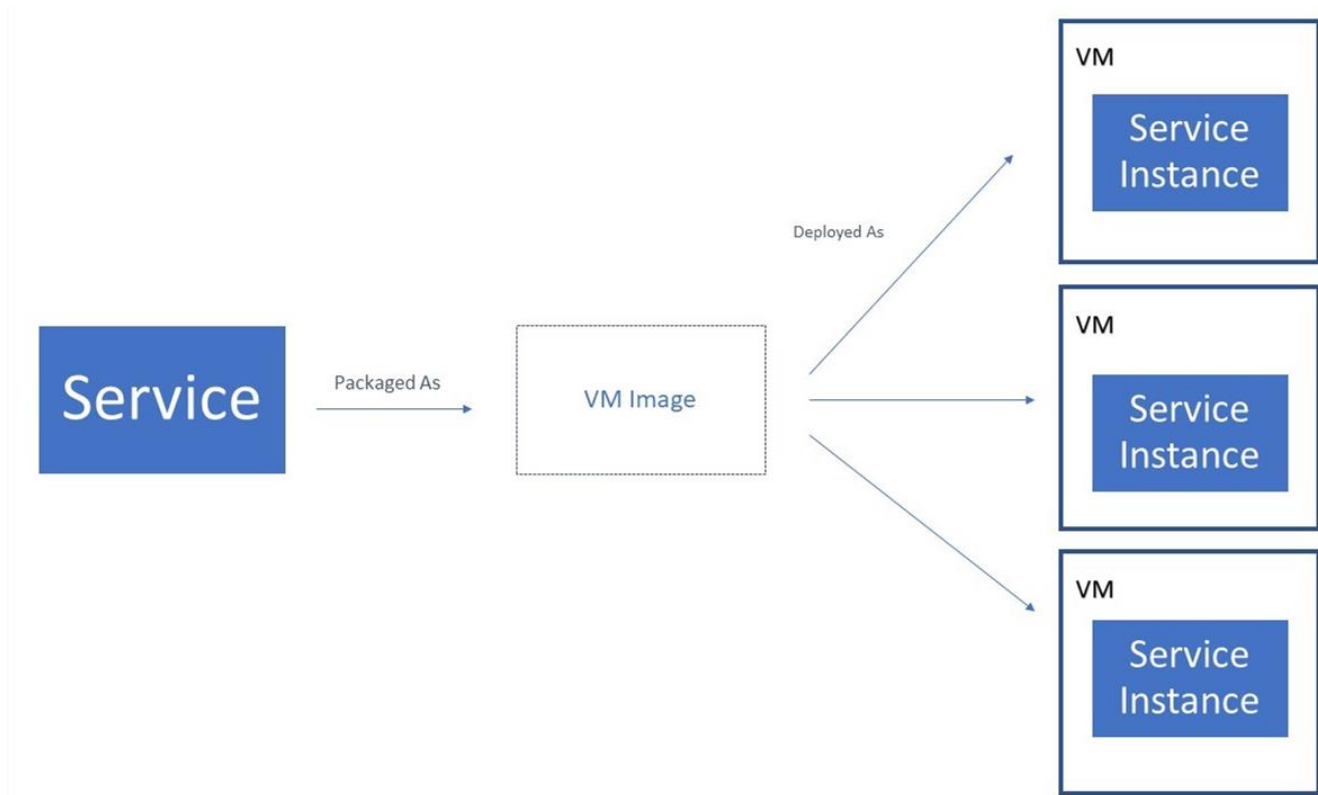
Deployment Patterns

Deployment patterns in microservices architecture determine how services are packaged, deployed, and managed across various environments. These patterns help achieve scalability, isolation, and flexibility in managing microservices.

1. Single Service per Host Pattern

Each microservice is deployed on its own server or virtual machine. This isolation ensures that services don't interfere with each other and can be scaled independently. For example:

- **OrderService** runs on a separate virtual machine from **InventoryService**.



Advantages:

- **Isolation:** Faults in one service don't affect others.
- **Independent Scaling:** Each service can be scaled individually based on demand.
- **Simplified Management:** Easier to monitor and maintain individual services.

2. Multiple Services per Host Pattern

Multiple microservices are deployed on the same server or virtual machine. This pattern is often used in environments with limited resources or for low-traffic services. For example:

- **PaymentService** and **NotificationService** may run on the same server due to their low resource requirements.

Advantages:

- **Resource Efficiency:** Reduces infrastructure costs by utilizing server resources more effectively.
- **Simplified Management:** Fewer servers or virtual machines to manage.

3. Service Instance per Container Pattern

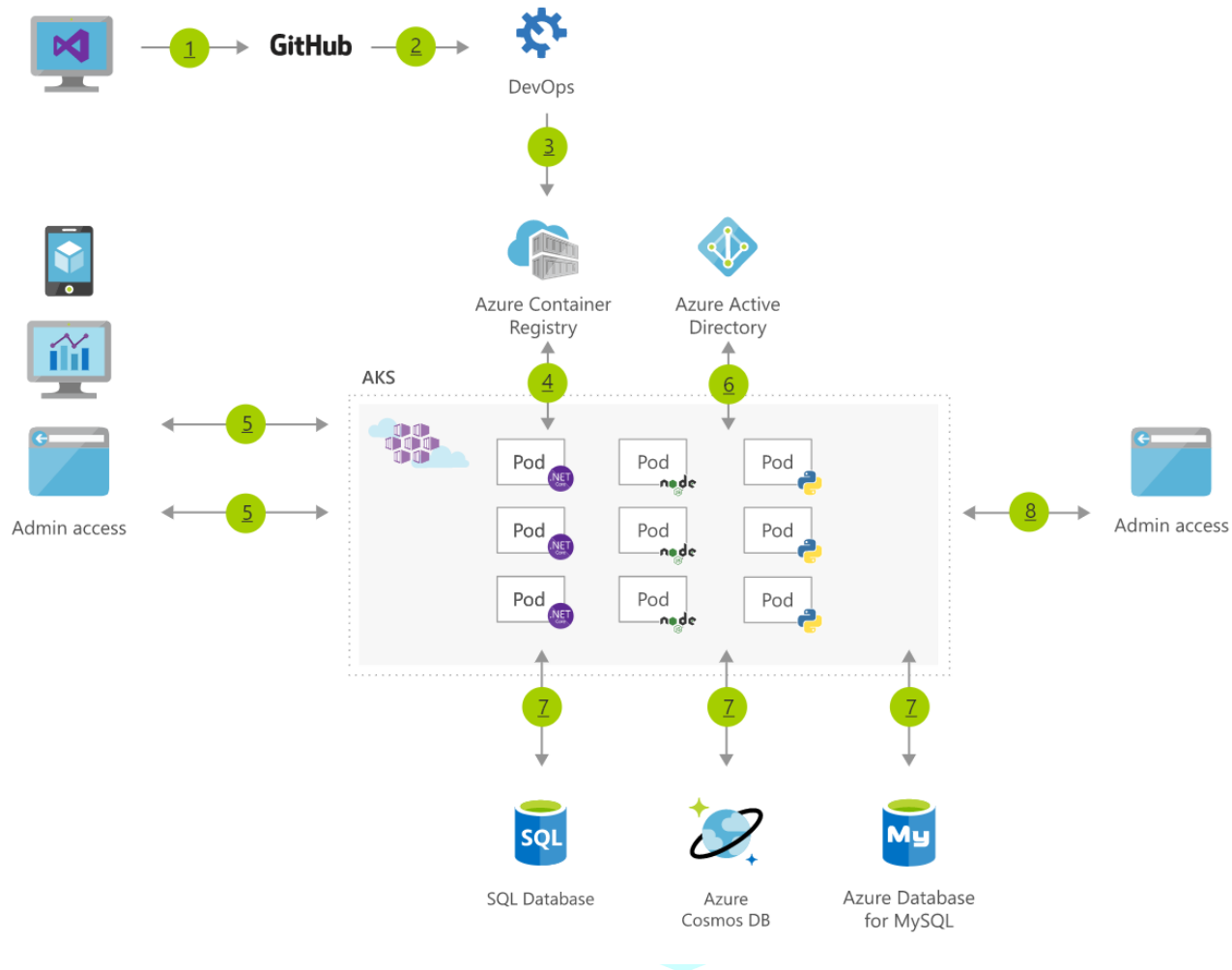
In this pattern, each microservice is packaged and deployed in its own container using technologies like Docker. Containers provide an isolated runtime environment for each service, ensuring consistency across different environments. For example:

- **UserService** is deployed in a Docker container, as is **CartService**.

Advantages:

- **Portability:** Containers can run consistently across development, testing, and production environments.

- **Isolation:** Services are isolated in their own environments, reducing the risk of conflicts.
- **Scalability:** Containers can be easily scaled up or down as needed.



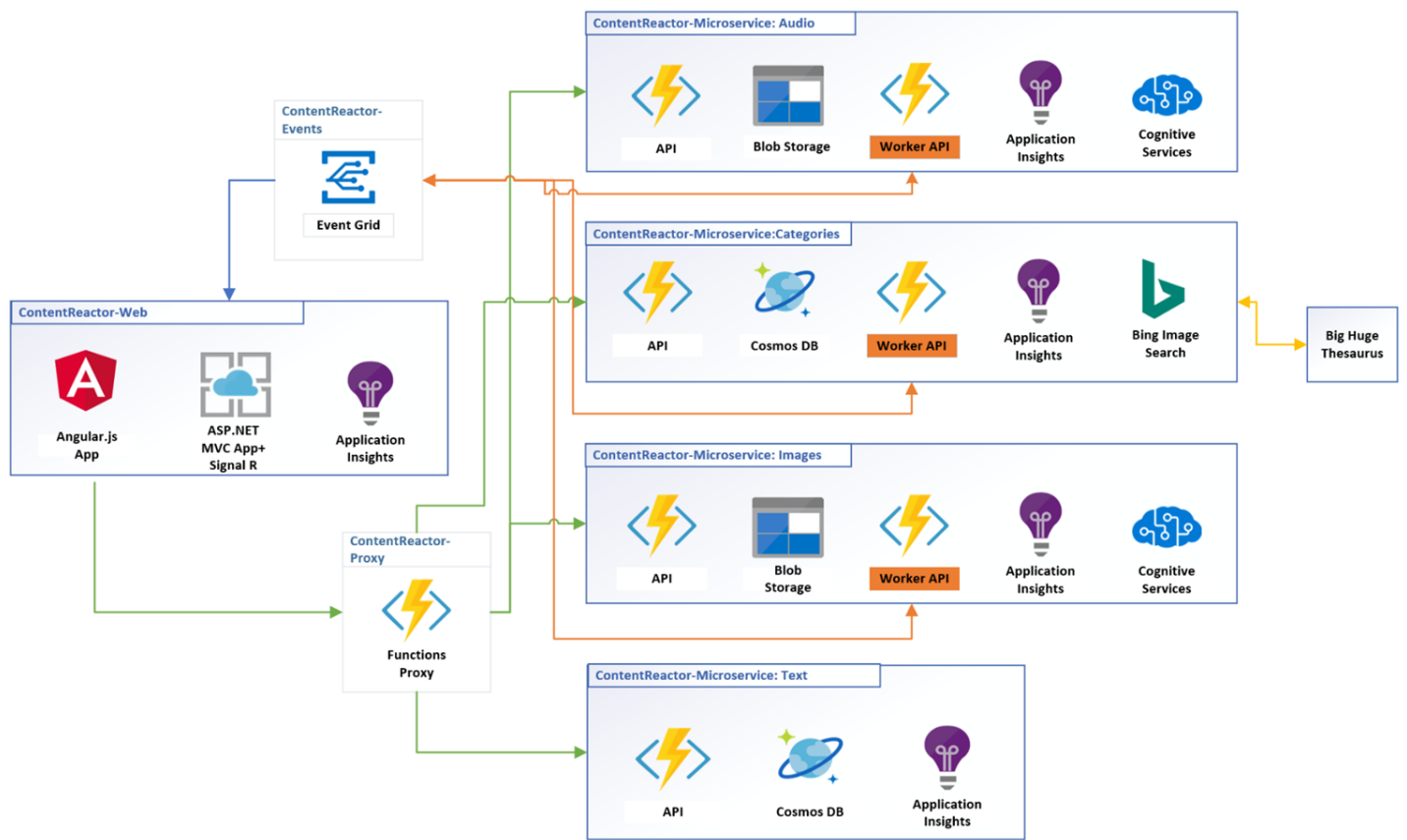
4. Serverless Deployment Pattern

In a serverless architecture, microservices are deployed as functions using platforms like AWS Lambda or Azure Functions. The platform automatically manages infrastructure, scaling, and load balancing. For example:

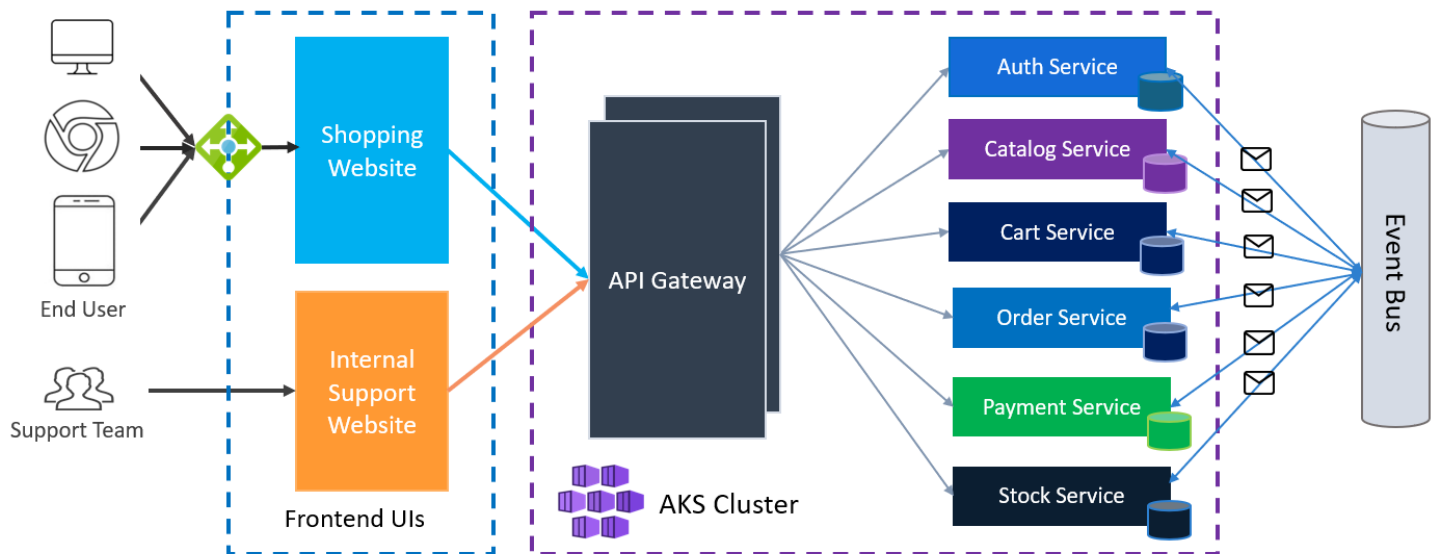
- **OrderProcessingService** is deployed as an Azure function that executes only when triggered by an event.

Advantages:

- **Reduced Infrastructure Management:** No need to manage servers, as the platform handles it automatically.
- **Cost Efficiency:** Pay only for the compute time used.
- **Automatic Scaling:** Services are automatically scaled based on demand.



AKS Deployment Using CI/CD



Creating AKS with ACR for Deployment

Create Kubernetes cluster ...

Basics Node pools Networking Integrations Monitoring Advanced Tags Review

Subscription * ⓘ Azure Subscription

Resource group * ⓘ (New) MicroservicesRg
[Create new](#)

Cluster details

Cluster preset configuration * ⓘ Production Economy
To quickly customize your Kubernetes cluster, choose one of the preset configurations above. You can modify these configurations at any time.
[Compare presets](#)

Kubernetes cluster name * ⓘ eshopflix

Region * ⓘ (Asia Pacific) Central India

Availability zones ⓘ None

AKS pricing tier ⓘ Standard

Kubernetes version * ⓘ 1.29.9 (default)

Automatic upgrade ⓘ Enabled with patch (recommended)

Automatic upgrade scheduler ⓘ Every week on Sunday (recommended)
Start on: Fri Oct 25 2024 00:00 +00:00 (Coordinated Universal Time)
[Edit schedule](#)

Node security channel type ⓘ Node Image

Security channel scheduler ⓘ Every week on Sunday (recommended)
Start on: Fri Oct 25 2024 00:00 +00:00 (Coordinated Universal Time)
[Edit schedule](#)

Choose between local accounts or Microsoft Entra ID for authentication and Azure RBAC or Kubernetes RBAC for your authorization needs.

Authentication and Authorization ⓘ Local accounts with Kubernetes RBAC

ⓘ Once the cluster is deployed, use the Kubernetes CLI to manage RBAC configurations. [Learn more](#)

Create Kubernetes cluster ...


Basics Node pools Networking Integrations Monitoring Advanced Tags Review

Node pools

In addition to the required primary node pool configured on the Basics tab, you can also add optional node pools to handle a variety of workloads [Learn more](#)

+ Add node pool  Delete

<input type="checkbox"/>	Name	Mode	Node size	OS SKU	Node count	Availat
<input type="checkbox"/>	agentpool	System	Standard_D8ds_v5 ...	Ubuntu	2 - 5	None
<input type="checkbox"/>	userpool	User	Standard_D8as_v4 ...	Ubuntu	0 - 25	None

 User node pool is recommended for production economy configuration with maximum of 25 nodes and a default value of 3 nodes.

Enable virtual nodes

Virtual nodes allow burstable scaling backed by serverless Azure Container Instances. [Learn more](#)

Enable virtual nodes  ☐

Node pool OS disk encryption

By default, all disks in AKS are encrypted at rest with Microsoft-managed keys. For additional control over encryption, you can supply your own keys using a disk encryption set backed by an Azure Key Vault. The disk encryption set will be used to encrypt the OS disks for all node pools in the cluster. [Learn more](#)

Encryption type (Default) Encryption at-rest with a platform-managed key

Create Kubernetes cluster ...

Basics Node pools **Networking** Integrations Monitoring Advanced Tags Review

Azure provides various networking controls to help manage and secure access to your Kubernetes cluster. [Learn more](#) ↗

Private access

Enable a private cluster to restrict worker node to API access, enhancing your Kubernetes workload's security and isolation.

Enable private cluster ⓘ ☐

Public access

Set authorized IP ranges ⓘ ☐

Container networking

Network configuration ⓘ

- ☒ **Azure CNI Overlay**
Assigns pod IP addresses from a private IP space. Best for scalability
- ☐ **Azure CNI Node Subnet**
Previously named Azure CNI. Assigns pod IP addresses from your host VNet. Best for workloads where pods must be reachable by other VNet resources
- ☐ **kubenet**
Older, route table-based Overlay with limited scalability. Not recommended for most clusters

Bring your own Azure virtual network ⓘ ☐

DNS name prefix * ⓘ

Enable Cilium dataplane and network policy ⓘ ☐

- Network policy * ⓘ
- ☒ **None**
Allow all ingress and egress traffic to the pods
 - ☐ **Calico**
Open-source networking solution. Best for large-scale deployments with strict security requirements
 - ☐ **Azure**
Native networking solution. Best for simpler deployments with basic security and networking requirements

Load balancer ⓘ **Standard**

Create Kubernetes cluster

Basics Node pools Networking **Integrations**

Connect your AKS cluster with additional services.

Microsoft Defender for Cloud

Microsoft Defender for Cloud provides unified security management for your workloads. [Learn more](#)

Your subscription is protected by Microsoft Defender for Cloud.

Azure Container Registry

Connect your cluster to an Azure Container Registry to enable image pull. [Learn more](#)

Container registry

None

[Create new](#)

Create container registry

Registry name *

eshopflix

.azurecr.io

Subscription

Azure Subscription

Resource group *

(New) MicroservicesRg

[Create new](#)

Region *

(Asia Pacific) Central India

Admin user * ⓘ

☐ Enable ☒ Disable

SKU * ⓘ

Standard

Availability zones ⓘ

☐

Create Kubernetes cluster

Basics Node pools Networking Integrations **Monitoring** Advanced Tags Review

Container Insights

Enable Container Logs



Azure monitor is recommended for production economy configuration.

Log Analytics workspace * ⓘ

(New) DefaultWorkspace-4cc400bc-8706-4682-b513-3a847ab32c7f-C...

[Create new](#)

Cost Preset * ⓘ

Standard

1 min collection frequency

No namespaces

Syslogs Disabled

Managed Prometheus

Managed Prometheus provides a highly available, scalable, and secure metrics platform to monitor your containerized workloads. [Learn more](#)

Enable Prometheus metrics



Azure Monitor workspace *

(New) defaultazuremonitorworkspace-cin

[Create new](#)

Create Kubernetes cluster ...

Basics Node pools Networking Integrations Monitoring **Advanced** Tags Review

Enable secret store CSI driver ⓘ ☐

Infrastructure resource group ⓘ

MC_MicroservicesRg_eshopflix_centralindia


[Edit](#)

Create Kubernetes cluster ...

Basics Node pools Networking Integrations Monitoring Advanced **Tags** Review + create

Tags are name/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups. [Learn more](#) ⓘ

Note that if you create tags and then change resource settings on other tabs, your tags will be automatically updated.

Name		Value		Resource	
Project	▼	:	eshopflix	▼	All resources selected ▼ 
	▼	:		▼	All resources selected ▼

CI/CD: Setup Service Connection

proshailendra / eshopflix19Oct / Settings / Service connections

Project Settings

eshopflix19Oct

General

- Overview
- Teams
- Permissions
- Notifications
- Service hooks
- Dashboards

Boards

- Project configuration
- Team configuration
- GitHub connections

Pipelines

- Agent pools
- Parallel jobs
- Settings
- Test management
- Release retention
- Service connections**
- XAML build services

Repos

Service connections

Convert your existing Azure Resource Manager service connections which use identity federation instead, for improved security and simplified maintenance.

Filter by keywords

- Azure Subscription (4cc400bc-8706-4682-b513-3a847ab32c7f)
- eshopflix

New Kubernetes service connection

Authentication method

- ☐ KubeConfig
- ☐ Service Account
- ☒ Azure Subscription

Azure Subscription

Azure Subscription (4cc400bc-8706-4682-b513-3a847ab32c7f) ▾

Cluster

eshopflix (MicroservicesRg) ▾

Namespace

default ▾

☐ Use cluster admin credentials

Details

Service connection name

eshop-flix-aks

Service Management Reference (optional)

Description (optional)

Security

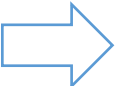
☒ Grant access permission to all pipelines

[Learn more](#)

[Troubleshoot](#)

Back Save

CI/CD: Setting for Kubernetes Cluster Deployment



KubectI

Kubernetes Cluster

Service connection type * ⓘ

Kubernetes Service Connection

Kubernetes service connection * ⓘ

eshop-flix-aks

Namespace ⓘ

default

Commands

Command ⓘ

apply

☐ Use configuration ⓘ

Arguments ⓘ

Secrets

Type of secret * ⓘ

dockerRegistry

Container registry type * ⓘ

Azure Container Registry

Azure subscription ⓘ

Azure Subscription(4cc400bc-8706-468...

Azure container registry ⓘ

eshopflix

Secret name ⓘ

acr-secret

☒ Force update secret ⓘ

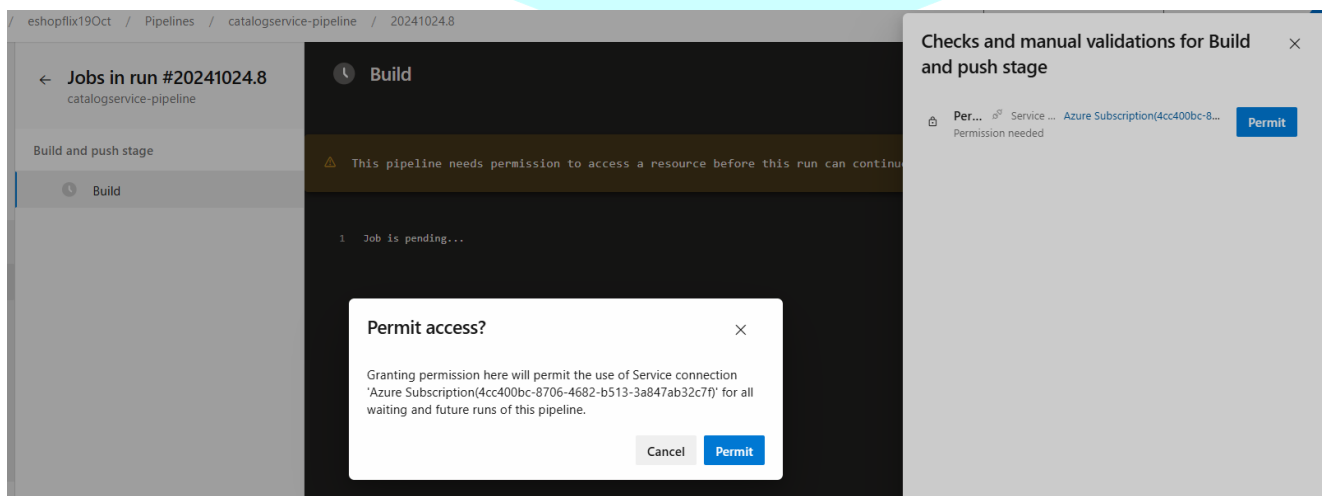
ConfigMaps

Advanced

About this task

Add

CI/CD: Permit Access to run Jobs



eshopflix19Oct / Pipelines / catalogservice-pipeline / 20241024.8

Jobs in run #20241024.8
catalogservice-pipeline

Build and push stage

Build

Build

This pipeline needs permission to access a resource before this run can continue

1 Job is pending...

Permit access?

Granting permission here will permit the use of Service connection 'Azure Subscription(4cc400bc-8706-4682-b513-3a847ab32c7f)' for all waiting and future runs of this pipeline.

Cancel Permit

Checks and manual validations for Build and push stage

Per... Service ... Azure Subscription(4cc400bc-8...
Permission needed

Permit

Contact Us

You can always reach out to us if you need help with this project. For getting help email us at hello@scholarhat.com or connect to our Discord's **#support** channel.