# Microservices

**1️⃣ UserService (Authentication & User Management) 👤**

🔹 **Handles**: User registration, authentication (JWT), role-based access control (RBAC).  
🔹 **Tech Stack**:

* **.NET 9 Web API**
* **Identity Framework**
* **MS SQL Server** (for structured user data)
* **MongoDB** (for user activity logs or session data)

🔹 **Endpoints**:

* **POST /api/users/register** → Register a new user
* **POST /api/users/login** → Authenticate user & generate JWT
* **GET /api/users/{id}** → Get user details

📌 **Use Case**: A traffic control officer logs into the system to monitor and control intersections.

**2️⃣ TrafficService (Traffic Light Control & AI Processing) 🚦**

🔹 **Handles**:

* Smart traffic light control based on real-time data
* AI-based optimization of signal timings
* Emergency vehicle prioritization

🔹 **Tech Stack**:

* **.NET 9 Web API**
* **ML.NET** (for AI-based traffic prediction)
* **RabbitMQ** (for real-time alerts)
* **MongoDB** (to store intersection and signal history for AI processing)

🔹 **Endpoints**:

* **GET /api/traffic/status/{intersectionId}** → Get current light status
* **POST /api/traffic/update** → Change traffic light timing
* **POST /api/traffic/emergency** → Prioritize emergency vehicle passage

📌 **Use Case**: AI detects high congestion at an intersection and dynamically adjusts the traffic lights.

**3️⃣ VehicleService (Vehicle Tracking & Traffic Analytics) 🚗**

🔹 **Handles**:

* GPS-based vehicle tracking
* Logs traffic flow data
* Provides analytics on vehicle movements

🔹 **Tech Stack**:

* **.NET 9 Web API**
* **MS SQL Server** (for structured vehicle data)
* **Redis** (for caching frequently accessed vehicle data)
* **MongoDB** (for real-time vehicle tracking logs and unstructured vehicle-related data)

🔹 **Endpoints**:

* **POST /api/vehicles/register** → Register a new vehicle
* **GET /api/vehicles/{licensePlate}** → Get vehicle details
* **GET /api/vehicles/realtime/{location}** → Get real-time vehicle count at a location

📌 **Use Case**: The system detects an unusual number of vehicles in an area, indicating a traffic jam.

**4️⃣ NotificationService (Alerts & Messaging) 🔔**

🔹 **Handles**:

* Sends alerts when accidents or high congestion occur
* Notifies emergency services about blocked roads
* Uses RabbitMQ for event-driven notifications

🔹 **Tech Stack**:

* **.NET 9 Worker Service**
* **RabbitMQ** (for message queuing)
* **MongoDB** (for storing historical notifications and events)

🔹 **Functionality**:

* Listens for **TrafficJamDetected** event from **TrafficService**
* Sends notifications via email/SMS

📌 **Use Case**: If an accident is detected, a notification is sent to the traffic control center and emergency services.

**5️⃣ GatewayAPI (Ocelot API Gateway) 🚪**

🔹 **Handles**:

* Acts as a single entry point for all microservices
* API routing, authentication, and load balancing

🔹 **Tech Stack**:

* **Ocelot API Gateway**

🔹 **Functionality**:

* Routes /users/\* requests to **UserService**
* Routes /traffic/\* requests to **TrafficService**
* Routes /vehicles/\* requests to **VehicleService**

📌 **Use Case**: A mobile app calls /traffic/status/123 and gets routed to **TrafficService** transparently.

**6️⃣ Infrastructure Services**

A. **SQL Server (Persistent Storage)** 🔹 **Handles**:

* Stores user data, vehicle data, and traffic logs (for structured relational data)

B. **RabbitMQ (Message Broker)** 🔹 **Handles**:

* Asynchronous event-driven communication between services
* Example: **TrafficService** detects congestion → sends event to **NotificationService**

C. **Redis (Caching)** 🔹 **Handles**:

* Caches frequently accessed traffic and vehicle data for fast retrieval
* Reduces SQL Server load

D. **MongoDB (Document-Based NoSQL Database)** 🔹 **Handles**:

* Stores unstructured or semi-structured data like traffic light signal history, user activity logs, real-time vehicle data, etc.
* Example: **TrafficService** stores traffic light patterns and AI-generated data in MongoDB.

**🌐 Microservices Communication Flow**

1️⃣ **VehicleService** detects high congestion and sends an event to **RabbitMQ**.  
2️⃣ **TrafficService** listens to the event and adjusts traffic lights.  
3️⃣ **NotificationService** picks up the event and sends an alert.  
4️⃣ The **API Gateway** ensures smooth communication across all services.

**Updated Database Use Case with MongoDB**

* **MongoDB** enhances flexibility for handling dynamic, unstructured data (vehicle tracking, traffic light patterns, real-time logs).
* **MS SQL Server** remains for relational data (e.g., user data, vehicle registrations).
* **Redis** ensures that high-frequency queries, such as current vehicle counts or traffic light status, are served quickly.

# Project Plan

To properly implement your system, you should start with the **UserService** as it forms the foundation for the rest of the system. The other services, like TrafficService, VehicleService, NotificationService, and the GatewayAPI, all rely on user authentication and management in some way (for example, user roles, access control, etc.).

Here's the recommended order to start building your services:

**1️⃣ UserService (Authentication & User Management) 👤**

* **Why start here?**  
  The UserService handles authentication (JWT), user registration, and role-based access control (RBAC), which is a fundamental part of ensuring the security and proper functionality of your system. All other services that require user-related data, such as traffic control officers, will need to interact with the UserService.
* **Tech Stack:**
  + .NET 9 Web API
  + Identity Framework
  + MS SQL Server (for structured user data)
  + MongoDB (for user activity logs or session data)
* **Endpoints:**
  + POST /api/users/register: Register a new user.
  + POST /api/users/login: Authenticate user & generate JWT.
  + GET /api/users/{id}: Get user details.

**Step 1**: Set up the UserService as a REST API.  
**Step 2**: Define database models for MS SQL Server (user info) and MongoDB (user activity logs).  
**Step 3**: Implement user registration, login with JWT, and role-based access control.  
**Step 4**: Once the UserService is up, other services can be developed to interact with this API for authentication.

**2️⃣ TrafficService (Traffic Light Control & AI Processing) 🚦**

* **Why second?**  
  TrafficService is one of the core microservices in the system. It controls the traffic lights, integrates AI for optimization, and processes traffic-related data. This service can utilize MongoDB for storing traffic light patterns and AI-generated data.
* **Tech Stack:**
  + .NET 9 Web API
  + ML.NET (for AI-based traffic prediction)
  + RabbitMQ (for real-time alerts)
  + MongoDB (to store intersection and signal history for AI processing)
* **Endpoints:**
  + GET /api/traffic/status/{intersectionId}: Get the current light status.
  + POST /api/traffic/update: Change traffic light timing.
  + POST /api/traffic/emergency: Prioritize emergency vehicle passage.

**Step 1**: Set up the TrafficService as a REST API.  
**Step 2**: Integrate ML.NET for AI processing and prediction.  
**Step 3**: Implement RabbitMQ for event-driven notifications and MongoDB to store intersection and signal history.  
**Step 4**: This service will use the UserService for role-based access if needed (for authorized personnel).

**3️⃣ VehicleService (Vehicle Tracking & Traffic Analytics) 🚗**

* **Why third?**  
  VehicleService tracks vehicle movements and provides traffic analytics. It integrates with both Redis (for caching vehicle data) and MongoDB (for storing unstructured or real-time data), making it a good candidate to work on after TrafficService.
* **Tech Stack:**
  + .NET 9 Web API
  + MS SQL Server (for structured vehicle data)
  + Redis (for caching frequently accessed vehicle data)
  + MongoDB (for real-time vehicle tracking logs and unstructured vehicle-related data)
* **Endpoints:**
  + POST /api/vehicles/register: Register a new vehicle.
  + GET /api/vehicles/{licensePlate}: Get vehicle details.
  + GET /api/vehicles/realtime/{location}: Get real-time vehicle count at a location.

**Step 1**: Set up the VehicleService as a REST API.  
**Step 2**: Implement Redis caching for frequently accessed data.  
**Step 3**: Use MongoDB for storing vehicle tracking logs and unstructured data.  
**Step 4**: Integrate with TrafficService for real-time vehicle tracking and event handling.

**4️⃣ NotificationService (Alerts & Messaging) 🔔**

* **Why fourth?**  
  NotificationService sends alerts and messages based on events such as traffic congestion, accidents, or roadblocks. It interacts with RabbitMQ and MongoDB for event-driven communication and storing historical notifications.
* **Tech Stack:**
  + .NET 9 Worker Service
  + RabbitMQ (for message queuing)
  + MongoDB (for storing historical notifications and events)
* **Functionality:**
  + Listens for events such as TrafficJamDetected from TrafficService.
  + Sends notifications via email/SMS.

**Step 1**: Set up the NotificationService as a Worker Service.  
**Step 2**: Implement RabbitMQ for asynchronous event-driven messaging.  
**Step 3**: Store notifications and event history in MongoDB.  
**Step 4**: Once this is in place, it will react to events from TrafficService and VehicleService.

**5️⃣ GatewayAPI (Ocelot API Gateway) 🚪**

* **Why last?**  
  The API Gateway acts as the entry point for all microservices and handles routing, load balancing, and authentication. It should be implemented last, as it needs to route requests to the services that are already set up and running.
* **Tech Stack:**
  + Ocelot API Gateway
* **Functionality:**
  + Routes /users/\* requests to UserService.
  + Routes /traffic/\* requests to TrafficService.
  + Routes /vehicles/\* requests to VehicleService.

**Step 1**: Set up Ocelot API Gateway and configure routes for all microservices.  
**Step 2**: Configure JWT authentication and load balancing for the API Gateway.  
**Step 3**: Once the other services are in place, this will route requests to the appropriate services.

**Infrastructure Setup**

After setting up the core microservices, you should focus on the infrastructure services:

1. **SQL Server** (for relational storage)
2. **RabbitMQ** (for asynchronous communication)
3. **Redis** (for caching)
4. **MongoDB** (for NoSQL data storage)

These infrastructure services should be configured and deployed early on to ensure smooth integration with the microservices.

**Summary of Steps:**

1. **Start with UserService** for authentication and user management.
2. **Develop TrafficService** with AI-based traffic control.
3. **Build VehicleService** for vehicle tracking and analytics.
4. **Set up NotificationService** to handle alerts and messaging.
5. **Implement API Gateway** (Ocelot) for routing requests to services.

By following this sequence, you ensure that each service can be built and tested in a logical flow, with dependencies managed as you go.

# User Service

### ****User Service Breakdown: Steps to Implement****

Let's dive into the UserService implementation process step-by-step, focusing on the tech stack, environment setup, and essential steps.

### ****1️⃣ Prerequisites & Environment Setup****

Before starting development, ensure the following prerequisites and environment setups are in place:

#### **Tech Stack for UserService:**

* **.NET 9 Web API**: To build the RESTful API for user management.
* **Identity Framework**: To handle authentication, user roles, and role-based access control (RBAC).
* **MS SQL Server**: For relational storage of user data (like username, password, email).
* **MongoDB**: For storing unstructured user-related data like activity logs or session data.
* **Docker**: To containerize the service and ensure it runs consistently across environments.
* **RabbitMQ** (Optional): If the service will communicate with other services asynchronously (e.g., sending welcome email, etc.).

#### **Environment Setup:**

1. **IDE**: Visual Studio or Visual Studio Code (VS Code) for .NET development.
2. **.NET 9 SDK**: Ensure you have the latest version of the .NET 9 SDK installed.
3. **Docker**: Docker Desktop or Docker Engine to build and run containers.
4. **Databases**:
   * **SQL Server**: Ensure SQL Server (Docker or local installation) is available for persistent storage.
   * **MongoDB**: Set up MongoDB, either locally or using a Docker container.
5. **RabbitMQ**: If using RabbitMQ for event-driven messaging between microservices, ensure that RabbitMQ is set up and accessible.

### ****2️⃣ Service Design: Structure of UserService****

Now, let's break down the steps to design and implement the UserService:

#### **Step 1: Create a New .NET 9 Web API Project**

* Open Visual Studio or VS Code and create a new **ASP.NET Core Web API** project using **.NET 9**.
* Use the following commands (if using the command line):

bash

CopyEdit

dotnet new webapi -n UserService

cd UserService

#### **Step 2: Configure Entity Framework (EF Core) for SQL Server**

* Install the required packages for Entity Framework Core with SQL Server support:

bash

CopyEdit

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

* Create a **DbContext** class to manage the connection to SQL Server:

csharp

CopyEdit

public class UserServiceDbContext : DbContext

{

public UserServiceDbContext(DbContextOptions<UserServiceDbContext> options) : base(options) { }

public DbSet<User> Users { get; set; }

// Add more DbSets as needed

}

* Add connection string in appsettings.json:

json

CopyEdit

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=UserServiceDb;User Id=sa;Password=your\_password;"

}

#### **Step 3: Setup Identity Framework for Authentication & Authorization**

* Install the necessary Identity packages:

bash

CopyEdit

dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore

* Configure the identity services:

csharp

CopyEdit

public class ApplicationUser : IdentityUser

{

public string Role { get; set; } // Optional: Add more properties for user data

}

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) { }

}

* Configure Identity in Program.cs:

csharp

CopyEdit

builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddIdentity<ApplicationUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

#### **Step 4: Implement JWT Authentication for User Login**

* Install JWT authentication middleware:

bash

CopyEdit

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer

* Configure JWT in Program.cs:

csharp

CopyEdit

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidIssuer = "YourIssuer",

ValidAudience = "YourAudience",

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKey"))

};

});

* Add endpoint for login in UserController.cs:

csharp

CopyEdit

[HttpPost("login")]

public async Task<IActionResult> Login([FromBody] LoginRequest model)

{

var user = await \_userManager.FindByNameAsync(model.Username);

if (user == null || !await \_userManager.CheckPasswordAsync(user, model.Password))

{

return Unauthorized();

}

var token = GenerateJwtToken(user); // Implement GenerateJwtToken

return Ok(new { Token = token });

}

#### **Step 5: User Registration and Role Management**

* Implement registration and role assignment:

csharp

CopyEdit

[HttpPost("register")]

public async Task<IActionResult> Register([FromBody] RegisterRequest model)

{

var user = new ApplicationUser { UserName = model.Username, Email = model.Email };

var result = await \_userManager.CreateAsync(user, model.Password);

if (!result.Succeeded) return BadRequest(result.Errors);

await \_userManager.AddToRoleAsync(user, "User"); // Or assign other roles

return Ok();

}

#### **Step 6: Integrating MongoDB (For User Activity or Session Logs)**

* Install MongoDB packages:

bash

CopyEdit

dotnet add package MongoDB.Driver

* Set up MongoDB client and models:

csharp

CopyEdit

public class MongoDbSettings

{

public string ConnectionString { get; set; }

public string DatabaseName { get; set; }

}

public class UserActivityLog

{

public string UserId { get; set; }

public DateTime Timestamp { get; set; }

public string Activity { get; set; }

}

public class UserActivityService

{

private readonly IMongoCollection<UserActivityLog> \_activityCollection;

public UserActivityService(IOptions<MongoDbSettings> mongoDbSettings)

{

var mongoClient = new MongoClient(mongoDbSettings.Value.ConnectionString);

var mongoDatabase = mongoClient.GetDatabase(mongoDbSettings.Value.DatabaseName);

\_activityCollection = mongoDatabase.GetCollection<UserActivityLog>("UserActivityLogs");

}

public async Task LogActivityAsync(string userId, string activity)

{

var activityLog = new UserActivityLog { UserId = userId, Timestamp = DateTime.UtcNow, Activity = activity };

await \_activityCollection.InsertOneAsync(activityLog);

}

}

#### **Step 7: Dockerize the Service**

1. **Dockerfile for UserService:**
   * Create a Dockerfile in the UserService folder:

dockerfile

CopyEdit

# Use the official .NET SDK image as a base image

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

WORKDIR /app

EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

WORKDIR /src

COPY ["UserService/UserService.csproj", "UserService/"]

RUN dotnet restore "UserService/UserService.csproj"

COPY . .

WORKDIR "/src/UserService"

RUN dotnet build "UserService.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "UserService.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "UserService.dll"]

1. **.dockerignore File**:
   * Create a .dockerignore file to exclude unnecessary files:

plaintext

CopyEdit

bin/

obj/

### ****3️⃣ Testing****

* Test the **UserService** API using tools like **Postman** or **Swagger**.
* Test **registration**, **login**, and **role management** functionality.
* Ensure **JWT token generation** and validation work properly.
* Test **MongoDB** integration for storing user activity logs.

### ****4️⃣ Docker Compose Setup****

Update the docker-compose.yml to include **UserService** and **MongoDB**:

yaml

CopyEdit

version: '3.8'

services:

user-service:

build:

context: ./services/user-service

environment:

- ASPNETCORE\_ENVIRONMENT=Development

- ConnectionStrings\_\_DefaultConnection=Server=sqlserver;Database=UserServiceDb;User Id=sa;Password=your\_password;

- MongoDbSettings\_\_ConnectionString=mongodb://mongo:27017

- MongoDbSettings\_\_DatabaseName=UserServiceDb

ports:

- "5000:80"

depends\_on:

- sqlserver

- mongo

mongo:

image: mongo:latest

volumes:

- mongo-data:/data/db

ports:

- "27017:27017"

sqlserver:

image: mcr.microsoft.com/mssql/server

environment:

- ACCEPT\_EULA=Y

- SA\_PASSWORD=your\_password

ports:

- "1433:1433"

volumes:

mongo-data:

### ****5️⃣ Final Touches****

* **API Documentation**: Integrate Swagger for easy API documentation and testing.
* **Logging & Monitoring**: Use libraries like **Serilog** for logging and **Prometheus** for metrics.
* **Security**: Ensure security best practices for handling sensitive data (e.g., password hashing, JWT storage).

This breakdown covers the core steps to get the **UserService** up and running. Once completed, you can move on to the other services.

# Traffic Service

### ****Traffic Service Breakdown: Steps to Implement****

Let's dive into the **TrafficService** implementation process step-by-step. We'll cover the tech stack, environment setup, and essential steps to get it up and running.

### ****1️⃣ Prerequisites & Environment Setup****

Before starting development, ensure the following prerequisites and environment setups are in place:

#### **Tech Stack for TrafficService:**

* **.NET 9 Web API**: To build the RESTful API for traffic control and AI processing.
* **ML.NET**: For AI-based traffic prediction and optimization (e.g., dynamic traffic light adjustments based on congestion).
* **MongoDB**: To store intersection and traffic signal history for AI-based processing.
* **RabbitMQ**: For event-driven communication between services (e.g., notifying **NotificationService** about congestion).
* **Docker**: For containerizing the service to run consistently across environments.

#### **Environment Setup:**

1. **IDE**: Visual Studio or Visual Studio Code (VS Code) for .NET development.
2. **.NET 9 SDK**: Ensure you have the latest version of the .NET 9 SDK installed.
3. **Docker**: Docker Desktop or Docker Engine to build and run containers.
4. **MongoDB**: Set up MongoDB, either locally or using a Docker container (MongoDB will store traffic signal patterns and AI-generated traffic data).
5. **RabbitMQ**: If using RabbitMQ for event-driven messaging, ensure that RabbitMQ is set up and accessible.

### ****2️⃣ Service Design: Structure of TrafficService****

Now, let's break down the steps to design and implement the TrafficService:

#### **Step 1: Create a New .NET 9 Web API Project**

* Open Visual Studio or VS Code and create a new **ASP.NET Core Web API** project using **.NET 9**.
* Use the following commands (if using the command line):

bash

CopyEdit

dotnet new webapi -n TrafficService

cd TrafficService

#### **Step 2: Configure Entity Framework (EF Core) for SQL Server** (Optional)

If you plan to store structured traffic data in a SQL Server database (e.g., historical data), you'll need to set up **Entity Framework** to connect to SQL Server.

* Install the required packages for Entity Framework Core with SQL Server support:

bash

CopyEdit

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

* Create a **DbContext** class to manage the connection to SQL Server (if applicable):

csharp

CopyEdit

public class TrafficServiceDbContext : DbContext

{

public TrafficServiceDbContext(DbContextOptions<TrafficServiceDbContext> options) : base(options) { }

public DbSet<TrafficSignal> TrafficSignals { get; set; }

// Add more DbSets as needed

}

* Add connection string in appsettings.json:

json

CopyEdit

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=TrafficServiceDb;User Id=sa;Password=your\_password;"

}

#### **Step 3: Setup MongoDB for Storing Traffic Data**

For storing traffic signal patterns and history, MongoDB is more suitable because it allows flexible, document-based storage.

* Install MongoDB packages:

bash

CopyEdit

dotnet add package MongoDB.Driver

* Configure MongoDB in Program.cs:

csharp

CopyEdit

builder.Services.Configure<MongoDbSettings>(builder.Configuration.GetSection("MongoDbSettings"));

builder.Services.AddSingleton<TrafficSignalService>();

* Create the MongoDB settings model (MongoDbSettings.cs):

csharp

CopyEdit

public class MongoDbSettings

{

public string ConnectionString { get; set; }

public string DatabaseName { get; set; }

}

* Create the **TrafficSignal** model for MongoDB:

csharp

CopyEdit

public class TrafficSignal

{

public string IntersectionId { get; set; }

public string SignalPattern { get; set; } // AI-generated pattern

public DateTime Timestamp { get; set; }

}

* Create the **TrafficSignalService** to interact with MongoDB:

csharp

CopyEdit

public class TrafficSignalService

{

private readonly IMongoCollection<TrafficSignal> \_trafficSignals;

public TrafficSignalService(IOptions<MongoDbSettings> mongoDbSettings)

{

var mongoClient = new MongoClient(mongoDbSettings.Value.ConnectionString);

var mongoDatabase = mongoClient.GetDatabase(mongoDbSettings.Value.DatabaseName);

\_trafficSignals = mongoDatabase.GetCollection<TrafficSignal>("TrafficSignals");

}

public async Task InsertTrafficSignalAsync(TrafficSignal trafficSignal)

{

await \_trafficSignals.InsertOneAsync(trafficSignal);

}

public async Task<List<TrafficSignal>> GetTrafficSignalHistoryAsync(string intersectionId)

{

return await \_trafficSignals.Find(signal => signal.IntersectionId == intersectionId).ToListAsync();

}

}

#### **Step 4: Setup RabbitMQ for Event-Driven Communication**

In this service, RabbitMQ will be used for sending alerts about traffic conditions to the **NotificationService**.

* Install RabbitMQ NuGet package:

bash

CopyEdit

dotnet add package RabbitMQ.Client

* Create a **RabbitMqService** for handling message publishing:

csharp

CopyEdit

public class RabbitMqService

{

private readonly IConnection \_connection;

private readonly IModel \_channel;

public RabbitMqService(IOptions<RabbitMqSettings> rabbitMqSettings)

{

var factory = new ConnectionFactory() { HostName = rabbitMqSettings.Value.HostName };

\_connection = factory.CreateConnection();

\_channel = \_connection.CreateModel();

}

public void SendTrafficJamAlert(string message)

{

\_channel.QueueDeclare(queue: "trafficJamQueue", durable: false, exclusive: false, autoDelete: false, arguments: null);

var body = Encoding.UTF8.GetBytes(message);

\_channel.BasicPublish(exchange: "", routingKey: "trafficJamQueue", basicProperties: null, body: body);

}

}

* **RabbitMqSettings.cs**:

csharp

CopyEdit

public class RabbitMqSettings

{

public string HostName { get; set; }

}

#### **Step 5: Implement AI Traffic Control Logic (ML.NET)**

TrafficService should use **ML.NET** for real-time traffic light optimization. This involves training an ML model to predict the best traffic signal patterns based on real-time traffic data.

* Install ML.NET NuGet package:

bash

CopyEdit

dotnet add package Microsoft.ML

* Example of creating and using an ML model:

csharp

CopyEdit

public class TrafficPredictionModel

{

private readonly PredictionEngine<TrafficData, TrafficPrediction> \_predictionEngine;

public TrafficPredictionModel(PredictionEngine<TrafficData, TrafficPrediction> predictionEngine)

{

\_predictionEngine = predictionEngine;

}

public TrafficPrediction Predict(TrafficData inputData)

{

return \_predictionEngine.Predict(inputData);

}

}

public class TrafficData

{

public int IntersectionId { get; set; }

public int VehicleCount { get; set; }

public int CongestionLevel { get; set; }

}

public class TrafficPrediction

{

public string SignalPattern { get; set; }

}

#### **Step 6: Dockerize the Service**

1. **Dockerfile for TrafficService**:
   * Create a Dockerfile in the **TrafficService** folder:

dockerfile

CopyEdit

# Use the official .NET SDK image as a base image

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

WORKDIR /app

EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

WORKDIR /src

COPY ["TrafficService/TrafficService.csproj", "TrafficService/"]

RUN dotnet restore "TrafficService/TrafficService.csproj"

COPY . .

WORKDIR "/src/TrafficService"

RUN dotnet build "TrafficService.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "TrafficService.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "TrafficService.dll"]

1. **.dockerignore File**:
   * Create a .dockerignore file to exclude unnecessary files:

plaintext

CopyEdit

bin/

obj/

### ****3️⃣ Testing****

* Test the **TrafficService** API using tools like **Postman** or **Swagger**.
* Test **AI-based traffic prediction**, including the **Signal Pattern** generation.
* Ensure **RabbitMQ** event publishing works as expected.
* Test **MongoDB** integration for storing traffic signal history.

### ****4️⃣ Docker Compose Setup****

Update the docker-compose.yml to include **TrafficService**, **MongoDB**, and **RabbitMQ**:

yaml

CopyEdit

version: '3.8'

services:

traffic-service:

build:

context: ./services/traffic-service

environment:

- ASPNETCORE\_ENVIRONMENT=Development

- MongoDbSettings\_\_ConnectionString=mongodb://mongo:27017

- MongoDbSettings\_\_DatabaseName=TrafficServiceDb

- RabbitMqSettings\_\_HostName=rabbitmq

ports:

- "5001:80"

depends\_on:

- mongo

- rabbitmq

mongo:

image: mongo:latest

volumes:

- mongo-data:/data/db

ports:

- "27017:27017"

rabbitmq:

image: rabbitmq:management

ports:

- "15672:15672"

- "5672:5672"

volumes:

mongo-data:

### ****5️⃣ Final Touches****

* **API Documentation**: Integrate **Swagger** for easy API documentation and testing.
* **Logging & Monitoring**: Use libraries like **Serilog** for logging and **Prometheus** for metrics.
* **Security**: Ensure security best practices are followed, especially for sensitive data.

# Vehicle Service

### ****Vehicle Service Breakdown: Steps to Implement****

Now, let’s dive into the **VehicleService** breakdown. This service will handle vehicle tracking, traffic analytics, and provide endpoints to interact with vehicle data. Below are the steps to implement the service, along with the environment setup, tech stack, and essential components.

### ****1️⃣ Prerequisites & Environment Setup****

Before starting development, make sure the following environment setup and prerequisites are in place:

#### **Tech Stack for VehicleService:**

* **.NET 9 Web API**: To build the RESTful API for vehicle tracking and analytics.
* **MS SQL Server**: For storing structured vehicle data like registration, vehicle details, etc.
* **Redis**: For caching frequently accessed vehicle data (e.g., vehicle counts, status) to improve performance.
* **MongoDB**: For storing real-time vehicle tracking logs and unstructured vehicle-related data (optional).
* **Docker**: For containerizing the service for easier deployment and scaling.

#### **Environment Setup:**

1. **IDE**: Visual Studio or Visual Studio Code for .NET development.
2. **.NET 9 SDK**: Ensure you have the latest .NET 9 SDK installed.
3. **SQL Server**: Set up either a local instance or a Docker container running SQL Server for structured vehicle data.
4. **Redis**: Set up Redis, either locally or using a Docker container.
5. **MongoDB** (optional): If MongoDB is used, ensure it is set up.

### ****2️⃣ Service Design: Structure of VehicleService****

Here’s a breakdown of the steps to design and implement the **VehicleService**:

#### **Step 1: Create a New .NET 9 Web API Project**

* Open Visual Studio or VS Code and create a new **ASP.NET Core Web API** project using **.NET 9**.
* If using the command line:

bash

CopyEdit

dotnet new webapi -n VehicleService

cd VehicleService

#### **Step 2: Configure Entity Framework (EF Core) for SQL Server**

Vehicle data such as registration details will be stored in SQL Server, so configure **Entity Framework (EF Core)** to manage the connection to SQL Server.

* Install the necessary packages:

bash

CopyEdit

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

* Create a **DbContext** class to manage the connection to SQL Server:

csharp

CopyEdit

public class VehicleServiceDbContext : DbContext

{

public VehicleServiceDbContext(DbContextOptions<VehicleServiceDbContext> options) : base(options) { }

public DbSet<Vehicle> Vehicles { get; set; }

// You can add other DbSets if needed

}

* Add the connection string in appsettings.json:

json

CopyEdit

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=VehicleServiceDb;User Id=sa;Password=your\_password;"

}

#### **Step 3: Set Up Redis for Caching**

Redis will be used to cache frequently accessed vehicle data to optimize the service's performance.

* Install Redis NuGet package:

bash

CopyEdit

dotnet add package StackExchange.Redis

* In **Program.cs**, configure Redis:

csharp

CopyEdit

builder.Services.AddSingleton<IConnectionMultiplexer>(sp =>

{

var configuration = ConfigurationOptions.Parse("localhost:6379", true);

return ConnectionMultiplexer.Connect(configuration);

});

* Create a **CacheService** to interact with Redis:

csharp

CopyEdit

public class CacheService

{

private readonly IConnectionMultiplexer \_redis;

public CacheService(IConnectionMultiplexer redis)

{

\_redis = redis;

}

public async Task SetVehicleDataAsync(string vehicleId, string data)

{

var db = \_redis.GetDatabase();

await db.StringSetAsync(vehicleId, data, TimeSpan.FromMinutes(5)); // Cache data for 5 minutes

}

public async Task<string> GetVehicleDataAsync(string vehicleId)

{

var db = \_redis.GetDatabase();

return await db.StringGetAsync(vehicleId);

}

}

#### **Step 4: MongoDB for Real-Time Vehicle Tracking Logs (Optional)**

If you wish to store unstructured data related to real-time vehicle tracking, use MongoDB.

* Install MongoDB driver:

bash

CopyEdit

dotnet add package MongoDB.Driver

* Configure MongoDB settings in **Program.cs**:

csharp

CopyEdit

builder.Services.Configure<MongoDbSettings>(builder.Configuration.GetSection("MongoDbSettings"));

builder.Services.AddSingleton<VehicleTrackingService>();

* Create **MongoDbSettings**:

csharp

CopyEdit

public class MongoDbSettings

{

public string ConnectionString { get; set; }

public string DatabaseName { get; set; }

}

* Create the **VehicleTrackingService** to store real-time logs:

csharp

CopyEdit

public class VehicleTrackingService

{

private readonly IMongoCollection<VehicleTrackingLog> \_vehicleLogs;

public VehicleTrackingService(IOptions<MongoDbSettings> mongoDbSettings)

{

var mongoClient = new MongoClient(mongoDbSettings.Value.ConnectionString);

var mongoDatabase = mongoClient.GetDatabase(mongoDbSettings.Value.DatabaseName);

\_vehicleLogs = mongoDatabase.GetCollection<VehicleTrackingLog>("VehicleLogs");

}

public async Task LogVehicleMovementAsync(VehicleTrackingLog log)

{

await \_vehicleLogs.InsertOneAsync(log);

}

public async Task<List<VehicleTrackingLog>> GetVehicleLogsAsync(string vehicleId)

{

return await \_vehicleLogs.Find(log => log.VehicleId == vehicleId).ToListAsync();

}

}

* **VehicleTrackingLog** model:

csharp

CopyEdit

public class VehicleTrackingLog

{

public string VehicleId { get; set; }

public string Location { get; set; }

public DateTime Timestamp { get; set; }

public string Status { get; set; }

}

#### **Step 5: Create Vehicle Model and API Endpoints**

1. **Vehicle Model**:
   * This model will hold the vehicle's registration data.

csharp

CopyEdit

public class Vehicle

{

public string Id { get; set; }

public string LicensePlate { get; set; }

public string Model { get; set; }

public int Year { get; set; }

public string Owner { get; set; }

// Additional fields as required

}

1. **API Endpoints**:
   * The following endpoints are needed:
     + **POST** /api/vehicles/register: Register a new vehicle.
     + **GET** /api/vehicles/{licensePlate}: Get vehicle details.
     + **GET** /api/vehicles/realtime/{location}: Get real-time vehicle count at a location.
2. **Example VehicleController**:

csharp

CopyEdit

[Route("api/[controller]")]

[ApiController]

public class VehicleController : ControllerBase

{

private readonly VehicleServiceDbContext \_context;

private readonly CacheService \_cacheService;

private readonly VehicleTrackingService \_trackingService;

public VehicleController(VehicleServiceDbContext context, CacheService cacheService, VehicleTrackingService trackingService)

{

\_context = context;

\_cacheService = cacheService;

\_trackingService = trackingService;

}

[HttpPost("register")]

public async Task<IActionResult> RegisterVehicle([FromBody] Vehicle vehicle)

{

if (vehicle == null)

return BadRequest();

\_context.Vehicles.Add(vehicle);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetVehicleById), new { id = vehicle.Id }, vehicle);

}

[HttpGet("{licensePlate}")]

public async Task<IActionResult> GetVehicleById(string licensePlate)

{

var vehicle = await \_context.Vehicles.FirstOrDefaultAsync(v => v.LicensePlate == licensePlate);

if (vehicle == null)

return NotFound();

return Ok(vehicle);

}

[HttpGet("realtime/{location}")]

public async Task<IActionResult> GetRealTimeVehicleCount(string location)

{

// Example for caching real-time vehicle count

var cacheKey = $"vehicleCount:{location}";

var cachedData = await \_cacheService.GetVehicleDataAsync(cacheKey);

if (cachedData != null)

return Ok(cachedData);

// If data not cached, query and cache

var vehicleCount = await \_context.Vehicles.CountAsync(v => v.Location == location);

await \_cacheService.SetVehicleDataAsync(cacheKey, vehicleCount.ToString());

return Ok(vehicleCount);

}

}

#### **Step 6: Dockerize the VehicleService**

1. **Dockerfile for VehicleService**:
   * Create a Dockerfile in the **VehicleService** folder:

dockerfile

CopyEdit

# Use the official .NET SDK image as a base image

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

WORKDIR /app

EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

WORKDIR /src

COPY ["VehicleService/VehicleService.csproj", "VehicleService/"]

RUN dotnet restore "VehicleService/VehicleService.csproj"

COPY . .

WORKDIR "/src/VehicleService"

RUN dotnet build "VehicleService.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "VehicleService.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "VehicleService.dll"]

1. **docker-compose.yaml for VehicleService**:
   * Add the **vehicle-service** configuration to the docker-compose.yaml file.

yaml

CopyEdit

version: '3.4'

services:

vehicle-service:

build:

context: .

dockerfile: VehicleService/Dockerfile

ports:

- "5002:80"

depends\_on:

- mongo

- redis

- rabbitmq

### ****3️⃣ Final Touches****

* **Testing**: Write unit tests for your controllers, services, and repository layers.
* **Logging**: Implement logging using libraries like **Serilog** to capture errors, info logs, and performance metrics.
* **Monitoring**: Integrate **Prometheus** and **Grafana** for real-time metrics and service health checks.
* **Swagger**: Document the API using **Swagger** to generate interactive API docs.

# Notification Service

### ****Notification Service Breakdown: Steps to Implement****

The **NotificationService** handles alerts and messaging for the traffic system. This service will listen for events like traffic jams, accidents, and other system notifications, and send them to users (e.g., traffic control centers, emergency services) through different channels (email, SMS, etc.).

Here’s a breakdown of the steps to implement the **NotificationService**, including environment setup, tech stack, and implementation.

### ****1️⃣ Prerequisites & Environment Setup****

Before we start, make sure that you have the following environment setup and prerequisites in place:

#### **Tech Stack for NotificationService:**

* **.NET 9 Worker Service**: To build a background worker service that listens for events and sends notifications.
* **RabbitMQ**: For message queuing and event-driven architecture. The NotificationService will listen for messages/events from the TrafficService.
* **MongoDB**: To store historical notification logs and events (optional).
* **Email/SMS API** (e.g., Twilio for SMS, SendGrid for email): For sending notifications to users.
* **Docker**: For containerizing the service for easier deployment.

#### **Environment Setup:**

1. **IDE**: Visual Studio or Visual Studio Code for .NET development.
2. **.NET 9 SDK**: Ensure you have the latest .NET 9 SDK installed.
3. **RabbitMQ**: Set up RabbitMQ either locally or in Docker for handling event-driven messages.
4. **Email/SMS API**: Set up email/SMS services like Twilio (for SMS) or SendGrid (for email) and configure their API keys.
5. **MongoDB** (optional): Set up MongoDB to store event logs and historical data.

### ****2️⃣ Service Design: Structure of NotificationService****

#### **Step 1: Create a New .NET 9 Worker Service Project**

Start by creating a new Worker Service project, which is designed for background tasks.

* Use Visual Studio to create a **Worker Service** project, or run the following command:

bash

CopyEdit

dotnet new worker -n NotificationService

cd NotificationService

#### **Step 2: Install Required NuGet Packages**

* **RabbitMQ Client**: For interacting with RabbitMQ.

bash

CopyEdit

dotnet add package RabbitMQ.Client

* **Email/SMS API Client**: Depending on the service you choose (e.g., SendGrid for email, Twilio for SMS), you’ll need to install their respective NuGet packages.
  + For **SendGrid**:

bash

CopyEdit

dotnet add package SendGrid

* + For **Twilio** (for SMS):

bash

CopyEdit

dotnet add package Twilio

* **MongoDB Client** (Optional): For storing historical notifications in MongoDB.

bash

CopyEdit

dotnet add package MongoDB.Driver

#### **Step 3: Create Configuration for Services**

In **appsettings.json**, add the configuration details for RabbitMQ, Email/SMS API, and MongoDB (if used).

json

CopyEdit

{

"RabbitMQ": {

"Host": "localhost",

"QueueName": "notifications"

},

"EmailService": {

"SendGridApiKey": "your\_sendgrid\_api\_key",

"FromEmail": "no-reply@example.com"

},

"TwilioService": {

"AccountSid": "your\_twilio\_account\_sid",

"AuthToken": "your\_twilio\_auth\_token",

"FromPhoneNumber": "+1234567890"

},

"MongoDbSettings": {

"ConnectionString": "mongodb://localhost:27017",

"DatabaseName": "NotificationService"

}

}

#### **Step 4: Create RabbitMQ Consumer**

The NotificationService will listen for messages from RabbitMQ. You’ll need a **RabbitMQ Consumer** to receive messages and trigger notifications.

* **Message Models**: Define the data models for the events that will be consumed from RabbitMQ.

csharp

CopyEdit

public class TrafficJamDetectedEvent

{

public string IntersectionId { get; set; }

public string Severity { get; set; }

public DateTime Timestamp { get; set; }

}

* **RabbitMQ Consumer**: Create a **RabbitMqService** class to connect to RabbitMQ and consume messages.

csharp

CopyEdit

public class RabbitMqService

{

private readonly IConfiguration \_configuration;

private readonly IConnection \_connection;

private readonly IModel \_channel;

public RabbitMqService(IConfiguration configuration)

{

\_configuration = configuration;

\_connection = new ConnectionFactory() { HostName = \_configuration["RabbitMQ:Host"] }.CreateConnection();

\_channel = \_connection.CreateModel();

}

public void StartListening(Action<string> onMessageReceived)

{

\_channel.QueueDeclare(queue: \_configuration["RabbitMQ:QueueName"], durable: false, exclusive: false, autoDelete: false, arguments: null);

var consumer = new EventingBasicConsumer(\_channel);

consumer.Received += (model, eventArgs) =>

{

var body = eventArgs.Body.ToArray();

var message = Encoding.UTF8.GetString(body);

onMessageReceived(message);

};

\_channel.BasicConsume(queue: \_configuration["RabbitMQ:QueueName"], autoAck: true, consumer: consumer);

}

}

#### **Step 5: Implement Notification Logic**

The **NotificationService** will send notifications (e.g., email/SMS) based on the events received from RabbitMQ.

1. **Email Notification Service**:
   * If you're using **SendGrid** to send emails, create a service to send emails:

csharp

CopyEdit

public class EmailNotificationService

{

private readonly string \_sendGridApiKey;

private readonly string \_fromEmail;

public EmailNotificationService(IConfiguration configuration)

{

\_sendGridApiKey = configuration["EmailService:SendGridApiKey"];

\_fromEmail = configuration["EmailService:FromEmail"];

}

public async Task SendEmailAsync(string toEmail, string subject, string message)

{

var client = new SendGridClient(\_sendGridApiKey);

var from = new EmailAddress(\_fromEmail, "Traffic Control System");

var to = new EmailAddress(toEmail);

var plainTextContent = message;

var htmlContent = $"<strong>{message}</strong>";

var msg = MailHelper.CreateSingleEmail(from, to, subject, plainTextContent, htmlContent);

var response = await client.SendEmailAsync(msg);

}

}

1. **SMS Notification Service**:
   * If you're using **Twilio** for SMS, create a service to send SMS:

csharp

CopyEdit

public class SmsNotificationService

{

private readonly string \_accountSid;

private readonly string \_authToken;

private readonly string \_fromPhoneNumber;

public SmsNotificationService(IConfiguration configuration)

{

\_accountSid = configuration["TwilioService:AccountSid"];

\_authToken = configuration["TwilioService:AuthToken"];

\_fromPhoneNumber = configuration["TwilioService:FromPhoneNumber"];

}

public void SendSms(string toPhoneNumber, string message)

{

var twilio = new TwilioRestClient(\_accountSid, \_authToken);

var messageOptions = new CreateMessageOptions(new PhoneNumber(toPhoneNumber))

{

From = new PhoneNumber(\_fromPhoneNumber),

Body = message

};

var msg = MessageResource.Create(messageOptions);

}

}

#### **Step 6: Integrating RabbitMQ with Notification Logic**

The **Worker** will listen for events on RabbitMQ and, depending on the event type (e.g., traffic jam detected), trigger notifications via email or SMS.

* **Worker Implementation**: Create a **NotificationWorker** class that consumes messages from RabbitMQ and sends notifications.

csharp

CopyEdit

public class NotificationWorker : BackgroundService

{

private readonly RabbitMqService \_rabbitMqService;

private readonly EmailNotificationService \_emailService;

private readonly SmsNotificationService \_smsService;

public NotificationWorker(RabbitMqService rabbitMqService, EmailNotificationService emailService, SmsNotificationService smsService)

{

\_rabbitMqService = rabbitMqService;

\_emailService = emailService;

\_smsService = smsService;

}

protected override Task ExecuteAsync(CancellationToken stoppingToken)

{

\_rabbitMqService.StartListening(async (message) =>

{

var eventDetails = JsonConvert.DeserializeObject<TrafficJamDetectedEvent>(message);

if (eventDetails != null)

{

var notificationMessage = $"Traffic jam detected at intersection {eventDetails.IntersectionId}. Severity: {eventDetails.Severity}.";

// Send email notification

await \_emailService.SendEmailAsync("control-center@example.com", "Traffic Jam Alert", notificationMessage);

// Send SMS notification (for emergency services)

\_smsService.SendSms("+1234567890", notificationMessage);

}

});

return Task.CompletedTask;

}

}

#### **Step 7: MongoDB for Storing Event Logs (Optional)**

* If you need to store the event logs, you can save the messages in MongoDB.

csharp

CopyEdit

public class MongoDbService

{

private readonly IMongoCollection<NotificationLog> \_logs;

public MongoDbService(IOptions<MongoDbSettings> settings)

{

var client = new MongoClient(settings.Value.ConnectionString);

var database = client.GetDatabase(settings.Value.DatabaseName);

\_logs = database.GetCollection<NotificationLog>("NotificationLogs");

}

public async Task LogNotificationAsync(NotificationLog log)

{

await \_logs.InsertOneAsync(log);

}

}

public class NotificationLog

{

public string Message { get; set; }

public DateTime Timestamp { get; set; }

public string Type { get; set; } // Email/SMS

}

### ****3️⃣ Final Touches****

* **Testing**: Write unit tests for the message consumption logic, notification sending, and MongoDB logging (if applicable).
* **Logging**: Implement logging with **Serilog** for tracking events, errors, and performance.
* **Health Checks**: Add health checks for RabbitMQ, MongoDB, and other dependencies.
* **Docker**: Dockerize the service to ensure it can run in containerized environments.

### ****Dockerfile for NotificationService****

Create a Dockerfile for your service, assuming it's based on a **.NET 9 Worker**:

dockerfile

CopyEdit

# Use the official .NET SDK image to build the application

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

WORKDIR /app

EXPOSE 80

# Use SDK image to build the application

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

WORKDIR /src

COPY ["NotificationService/NotificationService.csproj", "NotificationService/"]

RUN dotnet restore "NotificationService/NotificationService.csproj"

COPY . .

WORKDIR "/src/NotificationService"

RUN dotnet build "NotificationService.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "NotificationService.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "NotificationService.dll"]

### ****4️⃣ Conclusion****

Once you complete the steps above, you’ll have a fully functional NotificationService that listens for events from RabbitMQ, processes them, and sends out notifications through various channels like email and SMS. You can also store the event logs in MongoDB for historical purposes.

# API Gateway Service

### ****API Gateway Service Breakdown: Steps to Implement****

The **API Gateway Service** is responsible for acting as a single entry point for all microservices. It routes requests to the appropriate backend services, handles authentication (e.g., JWT tokens), and can manage load balancing. In this case, we will use **Ocelot API Gateway** in .NET to create the API Gateway.

Here’s a breakdown of the steps to implement the **API Gateway Service**, including environment setup, tech stack, and implementation.

### ****1️⃣ Prerequisites & Environment Setup****

Before starting, ensure that the following environment setup and prerequisites are in place:

#### **Tech Stack for API Gateway:**

* **Ocelot API Gateway**: Ocelot is a lightweight API Gateway for .NET applications.
* **JWT Authentication**: For secure access to microservices via token-based authentication.
* **Docker**: For containerizing the API Gateway and running it as part of a microservices architecture.

#### **Environment Setup:**

1. **IDE**: Visual Studio or Visual Studio Code for .NET development.
2. **.NET 9 SDK**: Ensure that you have the latest .NET 9 SDK installed.
3. **Ocelot**: Install the **Ocelot** NuGet package.

bash

CopyEdit

dotnet add package Ocelot

1. **Docker**: Docker is optional, but if you're working with containers, ensure Docker is installed for containerizing the API Gateway.
2. **JWT Authentication Server**: Ensure that you have a working **User Service** that handles authentication and issues JWT tokens.

### ****2️⃣ Service Design: Structure of API Gateway****

#### **Step 1: Create a New .NET 9 Web API Project for API Gateway**

Start by creating a new **Web API** project for the API Gateway.

* Use Visual Studio or run the following command to create a new project:

bash

CopyEdit

dotnet new webapi -n ApiGateway

cd ApiGateway

#### **Step 2: Install Ocelot NuGet Package**

Install the **Ocelot** NuGet package to enable API Gateway functionality.

bash

CopyEdit

dotnet add package Ocelot

#### **Step 3: Configure Ocelot in the API Gateway**

The core of the API Gateway functionality lies in routing incoming requests to the correct microservices. This is done by configuring **Ocelot** in the **Startup.cs** (or **Program.cs** in .NET 6/7+) file.

In **Program.cs**, configure services and middleware:

csharp

CopyEdit

using Ocelot.DependencyInjection;

using Ocelot.Middleware;

var builder = WebApplication.CreateBuilder(args);

// Add Ocelot services

builder.Services.AddOcelot();

var app = builder.Build();

// Configure the Ocelot middleware

app.UseOcelot().Wait();

app.Run();

#### **Step 4: Configure Ocelot Routes**

Create an **ocelot.json** file in the root of the project directory to configure the routes. This configuration tells the API Gateway how to route requests to various microservices.

Here’s an example configuration for **Ocelot** routing:

json

CopyEdit

{

"ReRoutes": [

{

"DownstreamPathTemplate": "/api/users/register",

"UpstreamPathTemplate": "/users/register",

"UpstreamHttpMethod": ["POST"],

"DownstreamScheme": "http",

"DownstreamHostAndPort": "user-service:80"

},

{

"DownstreamPathTemplate": "/api/users/login",

"UpstreamPathTemplate": "/users/login",

"UpstreamHttpMethod": ["POST"],

"DownstreamScheme": "http",

"DownstreamHostAndPort": "user-service:80"

},

{

"DownstreamPathTemplate": "/api/traffic/status/{intersectionId}",

"UpstreamPathTemplate": "/traffic/status/{intersectionId}",

"UpstreamHttpMethod": ["GET"],

"DownstreamScheme": "http",

"DownstreamHostAndPort": "traffic-service:80"

},

{

"DownstreamPathTemplate": "/api/vehicles/register",

"UpstreamPathTemplate": "/vehicles/register",

"UpstreamHttpMethod": ["POST"],

"DownstreamScheme": "http",

"DownstreamHostAndPort": "vehicle-service:80"

},

{

"DownstreamPathTemplate": "/api/vehicles/{licensePlate}",

"UpstreamPathTemplate": "/vehicles/{licensePlate}",

"UpstreamHttpMethod": ["GET"],

"DownstreamScheme": "http",

"DownstreamHostAndPort": "vehicle-service:80"

},

{

"DownstreamPathTemplate": "/api/notification/send",

"UpstreamPathTemplate": "/notification/send",

"UpstreamHttpMethod": ["POST"],

"DownstreamScheme": "http",

"DownstreamHostAndPort": "notification-service:80"

}

],

"GlobalConfiguration": {

"BaseUrl": "http://localhost:5000"

}

}

In this example, each microservice has its corresponding route defined. For example, requests to /users/register will be forwarded to the **UserService** running on port 80, and requests to /traffic/status/{intersectionId} will be forwarded to the **TrafficService**.

* **DownstreamHostAndPort**: The address of the downstream service. This could be a Docker container name if using Docker Compose.
* **UpstreamPathTemplate**: The path exposed to the clients (e.g., /users/register).
* **DownstreamPathTemplate**: The actual path on the downstream service (e.g., /api/users/register).

#### **Step 5: Enable JWT Authentication**

Since the **API Gateway** is responsible for routing requests to microservices, it should validate incoming requests for authentication using **JWT tokens**.

1. **Install JWT NuGet package**:

bash

CopyEdit

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer

1. **Configure JWT Authentication in Program.cs**:

csharp

CopyEdit

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.Authority = "https://localhost:5001"; // Your Identity Server URL

options.Audience = "api\_gateway";

options.RequireHttpsMetadata = false;

});

1. **Secure Routes**: Add authorization to protect specific routes in the API Gateway.

csharp

CopyEdit

app.UseAuthentication();

app.UseAuthorization();

1. **Secure the Routes in Ocelot Configuration**: Add the AuthenticationOptions to the routes that require authentication in the **ocelot.json** configuration.

json

CopyEdit

{

"ReRoutes": [

{

"DownstreamPathTemplate": "/api/users/register",

"UpstreamPathTemplate": "/users/register",

"UpstreamHttpMethod": ["POST"],

"DownstreamScheme": "http",

"DownstreamHostAndPort": "user-service:80",

"AuthenticationOptions": {

"Required": true

}

},

...

]

}

This will ensure that only authenticated requests (those with valid JWT tokens) can access protected routes.

#### **Step 6: Additional Features (Optional)**

1. **Logging and Monitoring**: Implement logging using **Serilog** or **NLog** to capture all incoming requests and responses.
2. **Rate Limiting**: Add rate-limiting using Ocelot’s built-in support to prevent abuse or overload.
3. **Load Balancing**: Ocelot supports load balancing out of the box. You can configure this by specifying multiple downstream services.
4. **Caching**: Implement caching if needed, so that frequently accessed routes can be served faster.

### ****3️⃣ Final Touches****

Once you have the basic API Gateway functionality implemented:

1. **Testing**: Write unit tests to verify the JWT authentication flow and ensure requests are being correctly routed to the downstream services.
2. **Logging**: Ensure proper logging for troubleshooting and tracking.
3. **Health Checks**: Add health checks to monitor the status of the microservices and ensure the API Gateway is functioning correctly.
4. **Dockerization**: Containerize the API Gateway for deployment in Kubernetes or Docker Compose.

### ****Dockerfile for API Gateway****

Here’s an example **Dockerfile** to build and containerize the API Gateway:

dockerfile

CopyEdit

# Use the official .NET SDK image to build the application

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

WORKDIR /app

EXPOSE 80

# Use SDK image to build the application

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

WORKDIR /src

COPY ["ApiGateway/ApiGateway.csproj", "ApiGateway/"]

RUN dotnet restore "ApiGateway/ApiGateway.csproj"

COPY . .

WORKDIR "/src/ApiGateway"

RUN dotnet build "ApiGateway.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "ApiGateway.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "ApiGateway.dll"]

### ****4️⃣ Conclusion****

By completing the steps above, you’ll have a fully functional **API Gateway** using **Ocelot** that can route requests to the appropriate microservices, handle JWT authentication, and optionally implement features like rate-limiting, load balancing, and logging. The API Gateway acts as a secure and efficient entry point into the microservices architecture.