# Comprehensive Documentation for Car Maintenance API

This document provides a complete guide for building an ASP.NET Core Web API for a Car Maintenance system. It includes system analysis, repository design pattern, database implementation using Entity Framework Core (EF Core), Enhanced Entity-Relationship Diagram (EERD) with mapping, all features (including notifications, live chat, localization), authentication implementation, unit testing guide, API documentation using Swagger, and CI/CD pipeline integration. The project manages car details, owners, maintenance records, services, user authentication, notifications (e.g., email reminders), live chat (real-time support via SignalR), and localization for multi-language support.

The structure follows clean architecture principles with separation of concerns. This assumes .NET 8 or later, SQL Server (or SQLite for development), and tools like Visual Studio. The CI/CD pipeline uses GitHub Actions for automated build, test, and deployment.

You can copy-paste this into a Microsoft Word document for formatting (e.g., add headings, tables, diagrams, code highlighting).

<xaiArtifact artifact\_id="6ff1563f-1e11-4d68-a9b8-c9f7ba191820" artifact\_version\_id="9dcdb892-65e1-4187-8bce-087841706692" title="CarMaintenanceApi\_Documentation.md" contentType="text/markdown">

# Comprehensive Documentation for Car Maintenance API

## 1. System Analysis

### 1.1 Overview

The Car Maintenance API is a RESTful service for tracking vehicle maintenance. It allows users (e.g., mechanics, owners, admins) to manage cars, schedule services, record repairs, generate reports, send notifications, enable live chat, and support multiple languages. Key goals:

- Improve efficiency in tracking maintenance history.

- Reduce errors in scheduling and billing.

- Provide secure access with authentication.

- Notify users of upcoming maintenance or updates.

- Offer real-time chat for customer support.

- Support internationalization via localization.

- Document APIs clearly using Swagger.

- Automate build, test, and deployment with CI/CD.

### 1.2 Requirements

#### Functional Requirements:

- Manage Cars: CRUD operations for car details (make, model, year, VIN, etc.).

- Manage Owners: CRUD for car owners (name, contact, address).

- Manage Maintenance Records: Log services like oil changes, tire rotations, with dates, costs, and descriptions.

- Manage Services: Predefined service types (e.g., oil change, brake repair) with costs.

- Reporting: Get maintenance history for a car or owner.

- Search and Filtering: By date, car ID, owner, etc.

- Authentication: User registration, login, and token-based access.

- Notifications: Send email or push notifications for maintenance reminders or updates.

- Live Chat: Real-time messaging between users (e.g., owner and mechanic) using WebSockets.

- Localization: Support multiple languages (e.g., English, Spanish) for API responses and messages.

- API Documentation: Provide interactive API docs via Swagger UI.

- CI/CD: Automate build, test, and deployment processes.

#### Non-Functional Requirements:

- Performance: Handle up to 1000 records efficiently; real-time chat with low latency.

- Security: Use JWT for authentication, hash passwords; secure WebSockets.

- Scalability: Use repository pattern for easy DB switching; SignalR for chat scaling.

- Data Integrity: Enforce relationships (e.g., a maintenance record must link to a car).

- Tech Stack: ASP.NET Core API, EF Core for ORM, SQL Server, ASP.NET Core Identity for auth, Swashbuckle for Swagger, SignalR for live chat, MailKit for notifications, Resource files for localization, GitHub Actions for CI/CD.

- Testability: Include unit tests for key components.

- Reliability: Automated CI/CD ensures consistent deployments.

### 1.3 Use Cases

- Actor: Owner

- View my car's maintenance history.

- Schedule a new service.

- Receive notifications for due maintenance.

- Chat with mechanic in real-time.

- Actor: Mechanic

- Add a new maintenance record.

- Update service status.

- Send notifications to owners.

- Chat with owners.

- Actor: Admin

- Manage service types.

- Generate reports.

- Actor: Any User

- Register a new account.

- Login to obtain JWT.

- Switch language for localized responses.

- Actor: Developer

- Access interactive API documentation via Swagger UI.

- Rely on CI/CD for automated testing and deployment.

### 1.4 Assumptions and Constraints

- One car can have multiple owners (historical), but primary owner for simplicity.

- Notifications via email (extendable to push).

- Live chat is basic (one-to-one or group; stored in DB).

- Localization for strings in responses (e.g., error messages).

- CI/CD uses GitHub Actions; deploys to Azure App Service.

- Currency: USD for costs.

- Database: Relational (SQL).

- Users must authenticate for protected endpoints.

- Swagger UI accessible at `/swagger`.

## 2. Repository Design Pattern

The Repository Pattern abstracts data access, promoting separation of concerns. It acts as an in-memory collection for entities, hiding EF Core details. We'll use a Generic Repository for reusability, plus specific repositories for complex queries.

### 2.1 Benefits

- Testability: Mock repositories for unit tests.

- Flexibility: Switch from EF Core to another ORM easily.

- Clean Code: Business logic in services, data ops in repositories.

### 2.2 Implementation Structure

- \*\*IRepository<T>\*\*: Generic interface for CRUD.

- \*\*Repository<T>\*\*: Implements IRepository<T> using DbContext.

- \*\*IUnitOfWork\*\*: Manages multiple repositories and saves changes.

- \*\*UnitOfWork\*\*: Implements IUnitOfWork, injecting DbContext.

```csharp

// Interfaces/IRepository.cs

public interface IRepository<T> where T : class

{

Task<T> GetByIdAsync(int id);

Task<IEnumerable<T>> GetAllAsync();

Task AddAsync(T entity);

void Update(T entity);

void Delete(T entity);

}

// Interfaces/IUnitOfWork.cs

public interface IUnitOfWork : IDisposable

{

IRepository<Car> Cars { get; }

IRepository<Owner> Owners { get; }

IRepository<MaintenanceRecord> MaintenanceRecords { get; }

IRepository<ServiceType> ServiceTypes { get; }

IRepository<ChatMessage> ChatMessages { get; }

Task<int> CompleteAsync();

}

// Repositories/Repository.cs

public class Repository<T> : IRepository<T> where T : class

{

private readonly AppDbContext \_context;

private DbSet<T> \_dbSet;

public Repository(AppDbContext context)

{

\_context = context;

\_dbSet = \_context.Set<T>();

}

public async Task<T> GetByIdAsync(int id) => await \_dbSet.FindAsync(id);

public async Task<IEnumerable<T>> GetAllAsync() => await \_dbSet.ToListAsync();

public async Task AddAsync(T entity) => await \_dbSet.AddAsync(entity);

public void Update(T entity) => \_dbSet.Update(entity);

public void Delete(T entity) => \_dbSet.Remove(entity);

}

// Repositories/UnitOfWork.cs

public class UnitOfWork : IUnitOfWork

{

private readonly AppDbContext \_context;

public IRepository<Car> Cars { get; }

public IRepository<Owner> Owners { get; }

public IRepository<MaintenanceRecord> MaintenanceRecords { get; }

public IRepository<ServiceType> ServiceTypes { get; }

public IRepository<ChatMessage> ChatMessages { get; }

public UnitOfWork(AppDbContext context)

{

\_context = context;

Cars = new Repository<Car>(context);

Owners = new Repository<Owner>(context);

MaintenanceRecords = new Repository<MaintenanceRecord>(context);

ServiceTypes = new Repository<ServiceType>(context);

ChatMessages = new Repository<ChatMessage>(context);

}

public async Task<int> CompleteAsync() => await \_context.SaveChangesAsync();

public void Dispose() => \_context.Dispose();

}

```

## 3. Implement DB using EF Core

### 3.1 Setup

- Install packages: `Microsoft.EntityFrameworkCore`, `Microsoft.EntityFrameworkCore.SqlServer`, `Microsoft.EntityFrameworkCore.Tools`, `Microsoft.AspNetCore.Identity.EntityFrameworkCore`, `Microsoft.AspNetCore.Authentication.JwtBearer`, `Microsoft.IdentityModel.Tokens`, `System.IdentityModel.Tokens.Jwt`, `Swashbuckle.AspNetCore`, `Microsoft.AspNetCore.SignalR`, `MailKit`, `Microsoft.Extensions.Localization`.

- Connection String in appsettings.json: `"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=CarMaintenanceDb;Trusted\_Connection=True;"`

### 3.2 Entities

```csharp

// Models/AppUser.cs

public class AppUser : IdentityUser

{

// Custom props if needed, e.g., public string FullName { get; set; }

}

// Models/Car.cs

public class Car

{

public int Id { get; set; }

public string Make { get; set; }

public string Model { get; set; }

public int Year { get; set; }

public string VIN { get; set; }

public int OwnerId { get; set; }

public Owner Owner { get; set; }

public ICollection<MaintenanceRecord> MaintenanceRecords { get; set; }

}

// Models/Owner.cs

public class Owner

{

public int Id { get; set; }

public string Name { get; set; }

public string ContactNumber { get; set; }

public string Address { get; set; }

public ICollection<Car> Cars { get; set; }

}

// Models/MaintenanceRecord.cs

public class MaintenanceRecord

{

public int Id { get; set; }

public DateTime Date { get; set; }

public string Description { get; set; }

public decimal Cost { get; set; }

public int CarId { get; set; }

public Car Car { get; set; }

public int ServiceTypeId { get; set; }

public ServiceType ServiceType { get; set; }

}

// Models/ServiceType.cs

public class ServiceType

{

public int Id { get; set; }

public string Name { get; set; }

public decimal DefaultCost { get; set; }

public ICollection<MaintenanceRecord> MaintenanceRecords { get; set; }

}

// Models/ChatMessage.cs

public class ChatMessage

{

public int Id { get; set; }

public string SenderId { get; set; }

public string ReceiverId { get; set; }

public string Message { get; set; }

public DateTime Timestamp { get; set; }

public bool IsRead { get; set; }

public AppUser Sender { get; set; }

public AppUser Receiver { get; set; }

}

```

### 3.3 DbContext

```csharp

// Data/AppDbContext.cs

public class AppDbContext : IdentityDbContext<AppUser>

{

public DbSet<Car> Cars { get; set; }

public DbSet<Owner> Owners { get; set; }

public DbSet<MaintenanceRecord> MaintenanceRecords { get; set; }

public DbSet<ServiceType> ServiceTypes { get; set; }

public DbSet<ChatMessage> ChatMessages { get; set; }

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

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<Car>()

.HasOne(c => c.Owner)

.WithMany(o => o.Cars)

.HasForeignKey(c => c.OwnerId)

.OnDelete(DeleteBehavior.Cascade);

modelBuilder.Entity<MaintenanceRecord>()

.HasOne(m => m.Car)

.WithMany(c => c.MaintenanceRecords)

.HasForeignKey(m => m.CarId)

.OnDelete(DeleteBehavior.Cascade);

modelBuilder.Entity<MaintenanceRecord>()

.HasOne(m => m.ServiceType)

.WithMany(s => s.MaintenanceRecords)

.HasForeignKey(m => m.ServiceTypeId)

.OnDelete(DeleteBehavior.Restrict);

modelBuilder.Entity<ChatMessage>()

.HasOne(m => m.Sender)

.WithMany()

.HasForeignKey(m => m.SenderId)

.OnDelete(DeleteBehavior.NoAction);

modelBuilder.Entity<ChatMessage>()

.HasOne(m => m.Receiver)

.WithMany()

.HasForeignKey(m => m.ReceiverId)

.OnDelete(DeleteBehavior.NoAction);

modelBuilder.Entity<ServiceType>().HasData(

new ServiceType { Id = 1, Name = "Oil Change", DefaultCost = 50.00M },

new ServiceType { Id = 2, Name = "Brake Repair", DefaultCost = 200.00M }

);

modelBuilder.Entity<IdentityRole>().HasData(

new IdentityRole { Name = "Admin", NormalizedName = "ADMIN" },

new IdentityRole { Name = "Mechanic", NormalizedName = "MECHANIC" },

new IdentityRole { Name = "Owner", NormalizedName = "OWNER" }

);

}

}

```

### 3.4 Migrations

Run `Add-Migration InitialCreate`, `Add-Migration AddIdentity`, `Add-Migration AddChatMessages`, and `Update-Database`.

## 4. EERD & Mapping

### 4.1 Enhanced Entity-Relationship Diagram (EERD)

- \*\*Entities\*\*:

- AppUser (Id PK, UserName, Email, PasswordHash, etc.)

- IdentityRole (Id PK, Name)

- Owner (Id PK, Name, ContactNumber, Address)

- Car (Id PK, Make, Model, Year, VIN, OwnerId FK)

- ServiceType (Id PK, Name, DefaultCost)

- MaintenanceRecord (Id PK, Date, Description, Cost, CarId FK, ServiceTypeId FK)

- ChatMessage (Id PK, SenderId FK, ReceiverId FK, Message, Timestamp, IsRead)

- Other Identity tables (UserRoles, UserClaims, etc.)

- \*\*Relationships\*\*:

- AppUser M:N IdentityRole (via UserRoles junction).

- Owner 1:N Car (One owner owns many cars; Car has one owner. Cascade delete.)

- Car 1:N MaintenanceRecord (One car has many records; Record belongs to one car. Cascade delete.)

- ServiceType 1:N MaintenanceRecord (One type used in many records; Record uses one type. Restrict delete.)

- AppUser 1:N ChatMessage (as Sender; NoAction delete).

- AppUser 1:N ChatMessage (as Receiver; NoAction delete).

ASCII Diagram:

```

AppUser ----(M:N)---- IdentityRole

| \

| \ (1:N as Sender)

| \

| ChatMessage

| /

| / (1:N as Receiver)

| /

Id (PK)

...

Owner ----(1:N)---- Car ----(1:N)---- MaintenanceRecord ----(N:1)---- ServiceType

| | |

Id (PK) Id (PK) Id (PK) Id (PK)

Name Make Date Name

ContactNumber Model Description DefaultCost

Address Year Cost

VIN CarId (FK)

OwnerId (FK) ServiceTypeId (FK)

```

### 4.2 Mapping

```csharp

// Profiles/MappingProfile.cs

public class MappingProfile : Profile

{

public MappingProfile()

{

CreateMap<Car, CarDto>();

CreateMap<CarDto, Car>();

CreateMap<Owner, OwnerDto>();

CreateMap<OwnerDto, Owner>();

CreateMap<MaintenanceRecord, MaintenanceRecordDto>();

CreateMap<MaintenanceRecordDto, MaintenanceRecord>();

CreateMap<ServiceType, ServiceTypeDto>();

CreateMap<ServiceTypeDto, ServiceType>();

CreateMap<ChatMessage, ChatMessageDto>();

CreateMap<ChatMessageDto, ChatMessage>();

}

}

// DTOs/CarDto.cs (example)

public class CarDto

{

public int Id { get; set; }

public string Make { get; set; }

public string Model { get; set; }

public int Year { get; set; }

public string VIN { get; set; }

public int OwnerId { get; set; }

}

```

## 5. All Features

### 5.1 Project Structure

```

CarMaintenanceApi/

├── Controllers/

│ ├── AuthController.cs

│ ├── CarsController.cs

│ ├── OwnersController.cs

│ ├── MaintenanceRecordsController.cs

│ ├── ServiceTypesController.cs

│ ├── NotificationsController.cs

├── Hubs/

│ ├── ChatHub.cs

├── Data/

│ └── AppDbContext.cs

├── Interfaces/

│ ├── IRepository.cs

│ ├── IUnitOfWork.cs

│ ├── IAuthService.cs

│ ├── INotificationService.cs

├── Models/

│ ├── AppUser.cs

│ ├── Car.cs

│ ├── Owner.cs

│ ├── MaintenanceRecord.cs

│ ├── ServiceType.cs

│ ├── ChatMessage.cs

├── DTOs/

│ ├── CarDto.cs

│ ├── OwnerDto.cs

│ ├── MaintenanceRecordDto.cs

│ ├── ServiceTypeDto.cs

│ ├── LoginDto.cs

│ ├── RegisterDto.cs

│ ├── TokenDto.cs

│ ├── ChatMessageDto.cs

│ ├── NotificationDto.cs

├── Repositories/

│ ├── Repository.cs

│ ├── UnitOfWork.cs

├── Services/

│ ├── AuthService.cs

│ ├── CarService.cs

│ ├── NotificationService.cs

├── Profiles/

│ └── MappingProfile.cs

├── Resources/

│ ├── AppResources.resx

│ ├── AppResources.es.resx

├── .github/workflows/

│ └── ci-cd.yml // New for CI/CD

├── Program.cs

├── appsettings.json

└── CarMaintenanceApi.csproj

```

### 5.2 Services Layer

```csharp

// Services/INotificationService.cs

public interface INotificationService

{

Task SendEmailAsync(string email, string subject, string message);

}

// Services/NotificationService.cs

public class NotificationService : INotificationService

{

private readonly IConfiguration \_config;

public NotificationService(IConfiguration config)

{

\_config = config;

}

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

{

var smtpServer = \_config["Email:SmtpServer"];

var smtpPort = int.Parse(\_config["Email:SmtpPort"]);

var smtpUser = \_config["Email:SmtpUser"];

var smtpPass = \_config["Email:SmtpPass"];

var mimeMessage = new MimeMessage();

mimeMessage.From.Add(new MailboxAddress("Car Maintenance", smtpUser));

mimeMessage.To.Add(new MailboxAddress("", email));

mimeMessage.Subject = subject;

mimeMessage.Body = new TextPart("plain") { Text = message };

using var client = new SmtpClient();

await client.ConnectAsync(smtpServer, smtpPort, SecureSocketOptions.StartTls);

await client.AuthenticateAsync(smtpUser, smtpPass);

await client.SendAsync(mimeMessage);

await client.DisconnectAsync(true);

}

}

```

Existing services remain.

### 5.3 DTOs

```csharp

// DTOs/ChatMessageDto.cs

public class ChatMessageDto

{

public string SenderId { get; set; }

public string ReceiverId { get; set; }

public string Message { get; set; }

public DateTime Timestamp { get; set; }

}

// DTOs/NotificationDto.cs

public class NotificationDto

{

public string Email { get; set; }

public string Subject { get; set; }

public string Message { get; set; }

}

```

### 5.4 Authentication and SignalR Configuration

In Program.cs:

```csharp

services.AddIdentity<AppUser, IdentityRole>()

.AddEntityFrameworkStores<AppDbContext>()

.AddDefaultTokenProviders();

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(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"]))

};

});

services.AddAuthorization();

services.AddSignalR();

services.AddScoped<IAuthService, AuthService>();

services.AddScoped<INotificationService, NotificationService>();

services.AddAutoMapper(typeof(MappingProfile));

services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "Car Maintenance API", Version = "v1" });

c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme

{

Description = "JWT Authorization header using the Bearer scheme. Example: \"Authorization: Bearer {token}\"",

Name = "Authorization",

In = ParameterLocation.Header,

Type = SecuritySchemeType.Http,

Scheme = "bearer",

BearerFormat = "JWT"

});

c.AddSecurityRequirement(new OpenApiSecurityRequirement

{

{

new OpenApiSecurityScheme

{

Reference = new OpenApiReference

{

Type = ReferenceType.SecurityScheme,

Id = "Bearer"

}

},

new string[] {}

}

});

});

var supportedCultures = new[] { "en", "es" };

var localizationOptions = new RequestLocalizationOptions().SetDefaultCulture("en")

.AddSupportedCultures(supportedCultures)

.AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);

app.UseAuthentication();

app.UseAuthorization();

app.UseSwagger();

app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Car Maintenance API v1"));

app.MapHub<ChatHub>("/chatHub");

```

appsettings.json:

```json

{

"Jwt": {

"Key": "${JWT\_KEY}",

"Issuer": "https://yourapi.com",

"Audience": "https://yourapi.com"

},

"ConnectionStrings": {

"DefaultConnection": "${SQL\_CONNECTION\_STRING}"

},

"Email": {

"SmtpServer": "${SMTP\_SERVER}",

"SmtpPort": "${SMTP\_PORT}",

"SmtpUser": "${SMTP\_USER}",

"SmtpPass": "${SMTP\_PASS}"

}

}

```

### 5.5 Controllers

```csharp

// Controllers/NotificationsController.cs

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

[ApiController]

public class NotificationsController : ControllerBase

{

private readonly INotificationService \_notificationService;

private readonly IStringLocalizer<AppResources> \_localizer;

public NotificationsController(INotificationService notificationService, IStringLocalizer<AppResources> localizer)

{

\_notificationService = notificationService;

\_localizer = localizer;

}

/// <summary>

/// Sends a notification email.

/// </summary>

[Authorize]

[HttpPost("send")]

public async Task<ActionResult> Send(NotificationDto dto)

{

try

{

await \_notificationService.SendEmailAsync(dto.Email, dto.Subject, dto.Message);

return Ok(\_localizer["Success"]);

}

catch (Exception ex)

{

return BadRequest(\_localizer["Error", ex.Message]);

}

}

}

```

Existing controllers (e.g., CarsController, AuthController) remain with added localization support.

### 5.6 Live Chat with SignalR

```csharp

// Hubs/ChatHub.cs

public class ChatHub : Hub

{

private readonly IUnitOfWork \_unitOfWork;

public ChatHub(IUnitOfWork unitOfWork)

{

\_unitOfWork = unitOfWork;

}

public async Task SendMessage(string receiverId, string message)

{

var senderId = Context.UserIdentifier;

var chatMessage = new ChatMessage

{

SenderId = senderId,

ReceiverId = receiverId,

Message = message,

Timestamp = DateTime.UtcNow,

IsRead = false

};

await \_unitOfWork.ChatMessages.AddAsync(chatMessage);

await \_unitOfWork.CompleteAsync();

await Clients.User(receiverId).SendAsync("ReceiveMessage", senderId, message);

}

}

```

### 5.7 Localization

Resources/AppResources.resx (English):

- Key: "Success", Value: "Operation completed successfully"

- Key: "Error", Value: "Error: {0}"

- Key: "ErrorNotFound", Value: "Not Found"

Resources/AppResources.es.resx (Spanish):

- Key: "Success", Value: "Operación completada con éxito"

- Key: "Error", Value: "Error: {0}"

- Key: "ErrorNotFound", Value: "No Encontrado"

## 6. Deployment and Best Practices

- Host on Azure App Service.

- Use HTTPS.

- Optimize queries with .Include().

- Handle concurrency with timestamps.

- Security: Use environment variables for secrets, validate inputs, rate limiting.

- SignalR: Use Redis backplane for scaling.

- Swagger: Restrict access in production.

## 7. Unit Testing Guide

### 7.1 Setup

- Test project: `CarMaintenanceApi.Tests`.

- Packages: `xunit`, `xunit.runner.visualstudio`, `Moq`, `Microsoft.EntityFrameworkCore.InMemory`, `Microsoft.AspNetCore.Mvc.Testing`.

### 7.2 Testing New Features

Add tests for NotificationService and ChatHub.

```csharp

// UnitTests/NotificationTests.cs

public class NotificationTests

{

[Fact]

public async Task SendEmailAsync\_SendsEmail()

{

var mockConfig = new Mock<IConfiguration>();

mockConfig.Setup(c => c["Email:SmtpServer"]).Returns("smtp.example.com");

mockConfig.Setup(c => c["Email:SmtpPort"]).Returns("587");

mockConfig.Setup(c => c["Email:SmtpUser"]).Returns("user@example.com");

mockConfig.Setup(c => c["Email:SmtpPass"]).Returns("password");

var service = new NotificationService(mockConfig.Object);

await service.SendEmailAsync("test@example.com", "Test", "Message");

// Assert via integration or mock SMTP client

}

}

```

## 8. API Documentation with Swagger

- Access: `https://localhost:<port>/swagger`.

- Includes new endpoints (e.g., `/api/Notifications/send`).

- Notes WebSocket endpoint `/chatHub` for chat.

## 9. CI/CD Pipeline with GitHub Actions

### 9.1 Introduction

The CI/CD pipeline automates building, testing, and deploying the API using GitHub Actions. It ensures code quality and reliable deployments to Azure App Service.

### 9.2 Setup

1. Create a GitHub repository for the project.

2. Store secrets in GitHub (Settings > Secrets and variables > Actions):

- `AZURE\_WEBAPP\_PUBLISH\_PROFILE`: Azure App Service publish profile.

- `JWT\_KEY`: JWT secret key.

- `SQL\_CONNECTION\_STRING`: Database connection string.

- `SMTP\_SERVER`, `SMTP\_PORT`, `SMTP\_USER`, `SMTP\_PASS`: Email settings.

3. Add `.github/workflows/ci-cd.yml`:

```yaml

name: CI/CD Pipeline

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Setup .NET

uses: actions/setup-dotnet@v3

with:

dotnet-version: '8.0.x'

- name: Restore dependencies

run: dotnet restore

- name: Build

run: dotnet build --configuration Release --no-restore

- name: Run tests

run: dotnet test --no-build --verbosity normal

deploy:

runs-on: ubuntu-latest

needs: build

if: github.event\_name == 'push' && github.ref == 'refs/heads/main'

steps:

- uses: actions/checkout@v3

- name: Setup .NET

uses: actions/setup-dotnet@v3

with:

dotnet-version: '8.0.x'

- name: Restore dependencies

run: dotnet restore

- name: Build

run: dotnet build --configuration Release --no-restore

- name: Publish

run: dotnet publish --configuration Release --output ./publish

- name: Deploy to Azure

uses: azure/webapps-deploy@v2

with:

app-name: 'your-app-name'

publish-profile: ${{ secrets.AZURE\_WEBAPP\_PUBLISH\_PROFILE }}

package: ./publish

```

### 9.3 Workflow Explanation

- Triggers: Runs on push or pull request to `main`.

- Build Job:

- Checks out code.

- Sets up .NET 8.

- Restores dependencies.

- Builds in Release mode.

- Runs unit tests.

- Deploy Job (on push to main):

- Publishes app.

- Deploys to Azure App Service using publish profile.

### 9.4 Best Practices

- Secure secrets in GitHub Actions.

- Use environment-specific configs (e.g., appsettings.Production.json).

- Add test coverage reporting (e.g., Coverlet).

- Use staging environment before production.

- Monitor deployments via Azure Portal.

</xaiArtifact>

This is the full updated documentation. Test the CI/CD pipeline by pushing to GitHub and verify deployment on Azure. Expand as needed!