🚀 Complete BankingApp.API Implementation Guide for Visual Studio 2022

Phase 1: Create New API Project (10 minutes)

Step 1: Create the Project

1. **Open Visual Studio 2022**
2. Click **"Create a new project"**
3. Search for **"ASP.NET Core Web API"**
4. Select it and click **Next**
5. **Project name:** `BankingApp.API`
6. **Location:** Choose where you want it (same folder as your other projects)
7. Click **Next**
8. **Framework:** .NET 8.0 (or .NET 6.0+ if you prefer)
9. **Authentication Type:** None
10. ✅ **Configure for HTTPS**
11. ✅ **Use controllers**
12. ✅ **Enable OpenAPI support**
13. Click **Create**

Step 2: Install NuGet Packages

1. **Right-click** `BankingApp.API` project in Solution Explorer
2. Select **"Manage NuGet Packages..."**

3. Click **"Browse"** tab
4. **Install these packages ONE BY ONE:**

```
Microsoft.AspNetCore.Authentication.JwtBearer
System.IdentityModel.Tokens.Jwt
BCrypt.Net-Next
Dapper
Npgsql
```

For each package:

- Type the name in search box
- Click the package
- Click **"Install"**
- Click **"OK"** if license dialog appears
- Wait for installation to complete

Phase 2: Set Up Project Structure (10 minutes)

Step 3: Create Folder Structure

1. **Right-click** your `BankingApp.API` project
2. **Add > New Folder** for each of these:
   - `Models`
   - `Controllers` (should already exist)
   - `Interfaces`
   - `Repositories`

Step 4: Delete Default Files

1. **Delete** `WeatherForecast.cs`
2. **Delete** `Controllers/WeatherForecastController.cs`

Phase 3: Create Your Models (15 minutes)

Step 5: Create UserModel.cs

1. **Right-click** `Models` folder
2. **Add > Class**
3. **Name:** `UserModel.cs`
4. Click **Add**
5. **Replace ALL content** with:

```csharp
namespace BankingApp.API.Models
{
    public class UserModel
    {
        public int UserId { get; set; }
        public string Username { get; set; } = string.Empty;
        public string Email { get; set; } = string.Empty;
        public string Password { get; set; } = string.Empty;
        public DateTime CreatedAt { get; set; }
    }
}
```

Step 6: Create AccountModel.cs

1. **Right-click** `Models` folder
2. **Add > Class**
3. **Name:** `AccountModel.cs`
4. **Replace ALL content** with:

```csharp
namespace BankingApp.API.Models
{
    public class AccountModel
    {
        public int AccountId { get; set; }
        public int UserId { get; set; }
        public string AccountNumber { get; set; } = string.Empty;
        public string AccountType { get; set; } = string.Empty;
        public decimal Balance { get; set; }
        public DateTime CreatedAt { get; set; }
    }
}
```

Step 7: Create TransactionModel.cs

1. **Right-click** `Models` folder
2. **Add > Class**
3. **Name:** `TransactionModel.cs`
4. **Replace ALL content** with:

```csharp
```

```
namespace BankingApp.API.Models
{
    public class TransactionModel
    {
        public int TransactionId { get; set; }
        public int AccountId { get; set; }
        public decimal Amount { get; set; }
        public string TransactionType { get; set; } = string.Empty;
        public string Description { get; set; } = string.Empty;
        public DateTime CreatedAt { get; set; }
    }
}
```

Phase 4: Create Repository Interfaces (10 minutes)

Step 8: Create IUserRepository.cs

1. **Right-click** `Interfaces` folder
2. **Add > Class**
3. **Name:** `IUserRepository.cs`
4. **Replace ALL content** with:

```csharp
```

```csharp
using BankingApp.API.Models;

namespace BankingApp.API.Interfaces
{
    public interface IUserRepository
    {
        Task<UserModel?> ValidateUserAsync(string username, string password);
        Task<UserModel?> GetUserByUsernameAsync(string username);
        Task<UserModel?> GetUserByEmailAsync(string email);
        Task<int> CreateUserAsync(UserModel user);
        Task<UserModel?> GetUserByIdAsync(int userId);
    }
}
```

Step 9: Create IAccountRepository.cs

1. **Right-click** `Interfaces` folder
2. **Add > Class**
3. **Name:** `IAccountRepository.cs`
4. **Replace ALL content** with:

```csharp
```

```csharp
using BankingApp.API.Models;

namespace BankingApp.API.Interfaces
{
    public interface IAccountRepository
    {
        Task<IEnumerable<AccountModel>> GetAccountsByUserIdAsync(int userId);
        Task<AccountModel?> GetAccountByIdAsync(int accountId);
        Task<int> CreateAccountAsync(AccountModel account);
        Task UpdateBalanceAsync(int accountId, decimal newBalance);
    }
}
```

Step 10: Create ITransactionRepository.cs

1. **Right-click** `Interfaces` folder
2. **Add > Class**
3. **Name:** `ITransactionRepository.cs`
4. **Replace ALL content** with:

```csharp
```

```
using BankingApp.API.Models;

namespace BankingApp.API.Interfaces
{
    public interface ITransactionRepository
    {
        Task<int> CreateTransactionAsync(TransactionModel transaction);
        Task<IEnumerable<TransactionModel>> GetTransactionsByAccountIdAsync(int accountId);
    }
}
```

Phase 5: Create Repository Implementations (20 minutes)

Step 11: Create UserRepository.cs

1. **Right-click** `Repositories` folder
2. **Add > Class**
3. **Name:** `UserRepository.cs`
4. **Replace ALL content** with:

```csharp
```

```
using Dapper;
using Npgsql;
using BankingApp.API.Interfaces;
using BankingApp.API.Models;

namespace BankingApp.API.Repositories
{
    public class UserRepository : IUserRepository
    {
        private readonly string _connectionString;

        public UserRepository(string connectionString)
        {
            _connectionString = connectionString;
        }

        public async Task<UserModel?> ValidateUserAsync(string username, string password)
        {
            using var connection = new NpgsqlConnection(_connectionString);
            var sql = "SELECT * FROM Users WHERE Username = @username";
            var user = await connection.QueryFirstOrDefaultAsync<UserModel>(sql, new { username });

            if (user != null && BCrypt.Net.BCrypt.Verify(password, user.Password))
            {
                return user;
            }
            return null;
        }

        public async Task<UserModel?> GetUserByUsernameAsync(string username)
        {
            using var connection = new NpgsqlConnection(_connectionString);
```

```
        var sql = "SELECT * FROM Users WHERE Username = @username";
        return await connection.QueryFirstOrDefaultAsync<UserModel>(sql, new { username });
    }

    public async Task<UserModel?> GetUserByEmailAsync(string email)
    {
        using var connection = new NpgsqlConnection(_connectionString);
        var sql = "SELECT * FROM Users WHERE Email = @email";
        return await connection.QueryFirstOrDefaultAsync<UserModel>(sql, new { email });
    }

    public async Task<int> CreateUserAsync(UserModel user)
    {
        using var connection = new NpgsqlConnection(_connectionString);
        var sql = @"INSERT INTO Users (Username, Email, Password, CreatedAt)
                VALUES (@Username, @Email, @Password, @CreatedAt)
                RETURNING UserId";
        return await connection.QuerySingleAsync<int>(sql, user);
    }

    public async Task<UserModel?> GetUserByIdAsync(int userId)
    {
        using var connection = new NpgsqlConnection(_connectionString);
        var sql = "SELECT * FROM Users WHERE UserId = @userId";
        return await connection.QueryFirstOrDefaultAsync<UserModel>(sql, new { userId });
    }
  }
}
```

Step 12: Create AccountRepository.cs

1. **Right-click** Repositories folder

2. **Add > Class**

3. **Name:** `AccountRepository.cs`

4. **Replace ALL content** with:

```csharp
```

```
using Dapper;
using Npgsql;
using BankingApp.API.Interfaces;
using BankingApp.API.Models;

namespace BankingApp.API.Repositories
{
    public class AccountRepository : IAccountRepository
    {
        private readonly string _connectionString;

        public AccountRepository(string connectionString)
        {
            _connectionString = connectionString;
        }

        public async Task<IEnumerable<AccountModel>> GetAccountsByUserIdAsync(int userId)
        {
            using var connection = new NpgsqlConnection(_connectionString);
            var sql = "SELECT * FROM Accounts WHERE UserId = @userId";
            return await connection.QueryAsync<AccountModel>(sql, new { userId });
        }

        public async Task<AccountModel?> GetAccountByIdAsync(int accountId)
        {
            using var connection = new NpgsqlConnection(_connectionString);
            var sql = "SELECT * FROM Accounts WHERE AccountId = @accountId";
            return await connection.QueryFirstOrDefaultAsync<AccountModel>(sql, new { accountId });
        }

        public async Task<int> CreateAccountAsync(AccountModel account)
        {
```

```
        using var connection = new NpgsqlConnection(_connectionString);
        var sql = @"INSERT INTO Accounts (UserId, AccountNumber, AccountType, Balance, CreatedAt)
                VALUES (@UserId, @AccountNumber, @AccountType, @Balance, @CreatedAt)
                RETURNING AccountId";
        return await connection.QuerySingleAsync<int>(sql, account);
    }

    public async Task UpdateBalanceAsync(int accountId, decimal newBalance)
    {
        using var connection = new NpgsqlConnection(_connectionString);
        var sql = "UPDATE Accounts SET Balance = @newBalance WHERE AccountId = @accountId";
        await connection.ExecuteAsync(sql, new { newBalance, accountId });
    }
  }
}
```

Step 13: Create TransactionRepository.cs

1. **Right-click** `Repositories` folder
2. **Add > Class**
3. **Name:** `TransactionRepository.cs`
4. **Replace ALL content** with:

```csharp
```

```
using Dapper;
using Npgsql;
using BankingApp.API.Interfaces;
using BankingApp.API.Models;

namespace BankingApp.API.Repositories
{
    public class TransactionRepository : ITransactionRepository
    {
        private readonly string _connectionString;

        public TransactionRepository(string connectionString)
        {
            _connectionString = connectionString;
        }

        public async Task<int> CreateTransactionAsync(TransactionModel transaction)
        {
            using var connection = new NpgsqlConnection(_connectionString);
            var sql = @"INSERT INTO Transactions (AccountId, Amount, TransactionType, Description, CreatedAt)
                    VALUES (@AccountId, @Amount, @TransactionType, @Description, @CreatedAt)
                    RETURNING TransactionId";
            return await connection.QuerySingleAsync<int>(sql, transaction);
        }

        public async Task<IEnumerable<TransactionModel>> GetTransactionsByAccountIdAsync(int accountId)
        {
            using var connection = new NpgsqlConnection(_connectionString);
            var sql = "SELECT * FROM Transactions WHERE AccountId = @accountId ORDER BY CreatedAt DESC";
            return await connection.QueryAsync<TransactionModel>(sql, new { accountId });
        }
```

```
  }
}
```

Phase 6: Configure appsettings.json (5 minutes)

Step 14: Update appsettings.json

1. **Find** `appsettings.json` in your project root
2. **Double-click** to open it
3. **Replace ALL content** with:

```json
```

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Host=localhost;Database=BankingAppDB;Username=your_username;Password=your_p
  },
  "JwtSettings": {
    "Secret": "YourSuperSecretKeyThatShouldBeAtLeast256BitsLongForSecurity!123456789",
    "Issuer": "BankingAppAPI",
    "Audience": "BankingAppUser",
    "ExpiryMinutes": 30
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

⚠️ **IMPORTANT:** Change `your_username` and `your_password` to your actual PostgreSQL credentials!

Phase 7: Configure Program.cs (10 minutes)

Step 15: Update Program.cs

1. **Find** `Program.cs` in your project root
2. **Replace ALL content** with:

```csharp
```

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using BankingApp.API.Interfaces;
using BankingApp.API.Repositories;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// Add CORS
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAll", policy =>
    {
        policy.AllowAnyOrigin()
              .AllowAnyMethod()
              .AllowAnyHeader();
    });
});

// Database Connection String
builder.Services.AddSingleton<string>(provider =>
    builder.Configuration.GetConnectionString("DefaultConnection")!);

// Repository Registration (Dependency Injection)
builder.Services.AddScoped<IUserRepository, UserRepository>();
builder.Services.AddScoped<IAccountRepository, AccountRepository>();
builder.Services.AddScoped<ITransactionRepository, TransactionRepository>();
```

```
// JWT Authentication Configuration
var config = builder.Configuration;
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = config["JwtSettings:Issuer"],
            ValidAudience = config["JwtSettings:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(config["JwtSettings:Secret"]!))
        };
    });

builder.Services.AddAuthorization();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseCors("AllowAll");
```

```
// ⚠️ IMPORTANT: Authentication MUST come before Authorization
app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();

app.Run();
```

Phase 8: Create Controllers (15 minutes)

Step 16: Create AuthenticationController.cs

1. **Right-click** `Controllers` folder
2. **Add > Controller...**
3. Select **"API Controller - Empty"**
4. **Name:** `AuthenticationController`
5. **Replace ALL content** with:

```csharp
```

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using BankingApp.API.Interfaces;
using BankingApp.API.Models;

namespace BankingApp.API.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class AuthenticationController : ControllerBase
    {
        private readonly IUserRepository _userRepository;
        private readonly IConfiguration _config;

        public AuthenticationController(IUserRepository userRepository, IConfiguration config)
        {
            _userRepository = userRepository;
            _config = config;
        }

        [HttpPost("login")]
        public async Task<IActionResult> Login(LoginRequest request)
        {
            try
            {
                var user = await _userRepository.ValidateUserAsync(request.Username, request.Password);

                if (user == null)
                {
```

```
            return Unauthorized(new { message = "Invalid username or password" });
        }

        var token = GenerateJwtToken(user);

        return Ok(new LoginResponse
        {
            Token = token,
            UserId = user.UserId,
            Username = user.Username,
            Email = user.Email,
            ExpiresAt = DateTime.UtcNow.AddMinutes(int.Parse(_config["JwtSettings:ExpiryMinutes"]!))
        });
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = "An error occurred during login", error = ex.Message });
    }
}

[HttpPost("register")]
public async Task<IActionResult> Register(RegisterRequest request)
{
    try
    {
        var existingUser = await _userRepository.GetUserByUsernameAsync(request.Username);
        if (existingUser != null)
        {
            return BadRequest(new { message = "Username already exists" });
        }

        var existingEmail = await _userRepository.GetUserByEmailAsync(request.Email);
```

```
            if (existingEmail != null)
            {
                return BadRequest(new { message = "Email already exists" });
            }

            var newUser = new UserModel
            {
                Username = request.Username,
                Email = request.Email,
                Password = BCrypt.Net.BCrypt.HashPassword(request.Password),
                CreatedAt = DateTime.UtcNow
            };

            var userId = await _userRepository.CreateUserAsync(newUser);
            newUser.UserId = userId;

            var token = GenerateJwtToken(newUser);

            return Created("", new LoginResponse
            {
                Token = token,
                UserId = newUser.UserId,
                Username = newUser.Username,
                Email = newUser.Email,
                ExpiresAt = DateTime.UtcNow.AddMinutes(int.Parse(_config["JwtSettings:ExpiryMinutes"]!))
            });
        }
        catch (Exception ex)
        {
            return StatusCode(500, new { message = "An error occurred during registration", error = ex.Message })
        }
    }
```

```
    private string GenerateJwtToken(UserModel user)
    {
        var claims = new[]
        {
            new Claim(JwtRegisteredClaimNames.Sub, user.UserId.ToString()),
            new Claim(JwtRegisteredClaimNames.Email, user.Email),
            new Claim("username", user.Username),
            new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
        };

        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["JwtSettings:Secret"]!));
        var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

        var token = new JwtSecurityToken(
            issuer: _config["JwtSettings:Issuer"],
            audience: _config["JwtSettings:Audience"],
            claims: claims,
            expires: DateTime.UtcNow.AddMinutes(int.Parse(_config["JwtSettings:ExpiryMinutes"]!)),
            signingCredentials: credentials
        );

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

public class LoginRequest
{
    public string Username { get; set; } = string.Empty;
    public string Password { get; set; } = string.Empty;
}
```

```
    public class RegisterRequest
    {
        public string Username { get; set; } = string.Empty;
        public string Email { get; set; } = string.Empty;
        public string Password { get; set; } = string.Empty;
    }

    public class LoginResponse
    {
        public string Token { get; set; } = string.Empty;
        public int UserId { get; set; }
        public string Username { get; set; } = string.Empty;
        public string Email { get; set; } = string.Empty;
        public DateTime ExpiresAt { get; set; }
    }
}
```

Step 17: Create AccountsController.cs

1. **Right-click** `Controllers` folder
2. **Add > Controller...**
3. Select **"API Controller - Empty"**
4. **Name:** `AccountsController`
5. **Replace ALL content** with:

```csharp
```

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;
using BankingApp.API.Interfaces;
using BankingApp.API.Models;

namespace BankingApp.API.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    [Authorize] // 🔒 All endpoints require JWT authentication
    public class AccountsController : ControllerBase
    {
        private readonly IAccountRepository _accountRepository;
        private readonly ITransactionRepository _transactionRepository;

        public AccountsController(IAccountRepository accountRepository, ITransactionRepositi
        {
            _accountRepository = accountRepository;
            _transactionRepository = transactionRepository;
        }

        [HttpGet]
        public async Task<IActionResult> GetUserAccounts()
        {
            try
            {
                var userId = GetCurrentUserId();
                if (userId == 0) return Unauthorized();

                var accounts = await _accountRepository.GetAccountsByUserIdAsync(userId);
                return Ok(accounts);
```

```
        }
        catch (Exception ex)
        {
            return StatusCode(500, new { message = "Error retrieving accounts", error = ex.Message });
        }
    }

    [HttpGet("{accountId}")]
    public async Task<IActionResult> GetAccount(int accountId)
    {
        try
        {
            var userId = GetCurrentUserId();
            if (userId == 0) return Unauthorized();

            var account = await _accountRepository.GetAccountByIdAsync(accountId);

            if (account == null)
                return NotFound(new { message = "Account not found" });

            if (account.UserId != userId)
                return Forbid("You don't have access to this account");

            return Ok(account);
        }
        catch (Exception ex)
        {
            return StatusCode(500, new { message = "Error retrieving account", error = ex.Message });
        }
    }

    [HttpPost]
```

```
public async Task<IActionResult> CreateAccount(CreateAccountRequest request)
{
    try
    {
        var userId = GetCurrentUserId();
        if (userId == 0) return Unauthorized();

        var newAccount = new AccountModel
        {
            UserId = userId,
            AccountNumber = GenerateAccountNumber(),
            AccountType = request.AccountType,
            Balance = request.InitialBalance,
            CreatedAt = DateTime.UtcNow
        };

        var accountId = await _accountRepository.CreateAccountAsync(newAccount);
        newAccount.AccountId = accountId;

        return Created("", newAccount);
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = "Error creating account", error = ex.Message });
    }
}

[HttpPost("{accountId}/deposit")]
public async Task<IActionResult> Deposit(int accountId, TransactionRequest request)
{
    try
    {
```

```
        var userId = GetCurrentUserId();
        if (userId == 0) return Unauthorized();

        var account = await _accountRepository.GetAccountByIdAsync(accountId);
        if (account == null || account.UserId != userId)
            return BadRequest(new { message = "Invalid account" });

        if (request.Amount <= 0)
            return BadRequest(new { message = "Amount must be greater than zero" });

        var transaction = new TransactionModel
        {
            AccountId = accountId,
            Amount = request.Amount,
            TransactionType = "Deposit",
            Description = request.Description ?? "ATM Deposit",
            CreatedAt = DateTime.UtcNow
        };

        var transactionId = await _transactionRepository.CreateTransactionAsync(transaction);
        await _accountRepository.UpdateBalanceAsync(accountId, account.Balance + request.Amount);

        transaction.TransactionId = transactionId;

        return Ok(new { message = "Deposit successful", transaction });
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = "Error processing deposit", error = ex.Message });
    }
}
```

```
[HttpPost("{accountId}/withdraw")]
public async Task<IActionResult> Withdraw(int accountId, TransactionRequest request)
{
    try
    {
        var userId = GetCurrentUserId();
        if (userId == 0) return Unauthorized();

        var account = await _accountRepository.GetAccountByIdAsync(accountId);
        if (account == null || account.UserId != userId)
            return BadRequest(new { message = "Invalid account" });

        if (request.Amount <= 0)
            return BadRequest(new { message = "Amount must be greater than zero" });

        if (account.Balance < request.Amount)
            return BadRequest(new { message = "Insufficient funds" });

        var transaction = new TransactionModel
        {
            AccountId = accountId,
            Amount = -request.Amount,
            TransactionType = "Withdrawal",
            Description = request.Description ?? "ATM Withdrawal",
            CreatedAt = DateTime.UtcNow
        };

        var transactionId = await _transactionRepository.CreateTransactionAsync(transaction);
        await _accountRepository.UpdateBalanceAsync(accountId, account.Balance - request.Amount);

        transaction.TransactionId = transactionId;
```

```
        return Ok(new { message = "Withdrawal successful", transaction });
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = "Error processing withdrawal", error = ex.Message });
    }
}

[HttpGet("{accountId}/transactions")]
public async Task<IActionResult> GetTransactionHistory(int accountId)
{
    try
    {
        var userId = GetCurrentUserId();
        if (userId == 0) return Unauthorized();

        var account = await _accountRepository.GetAccountByIdAsync(accountId);
        if (account == null || account.UserId != userId)
            return BadRequest(new { message = "Invalid account" });

        var transactions = await _transactionRepository.GetTransactionsByAccountIdAsync(accountId);
        return Ok(transactions);
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = "Error retrieving transactions", error = ex.Message });
    }
}

// Helper methods
private int GetCurrentUserId()
{
```

```
        var userIdClaim = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
        return int.TryParse(userIdClaim, out int userId) ? userId : 0;
    }

    private string GenerateAccountNumber()
    {
        var random = new Random();
        var accountNumber = "";
        for (int i = 0; i < 16; i++)
        {
            accountNumber += random.Next(0, 10);
        }
        return accountNumber;
    }
  }

  public class CreateAccountRequest
  {
    public string AccountType { get; set; } = "Checking";
    public decimal InitialBalance { get; set; } = 0;
  }

  public class TransactionRequest
  {
    public decimal Amount { get; set; }
    public string? Description { get; set; }
  }
}
```

Phase 9: Build and Test (10 minutes)

Step 18: Build the Project

1. **Build > Build Solution** (or Ctrl+Shift+B)
2. **Fix any red squiggly errors** if they appear
3. Make sure **"Build succeeded"** appears in output

Step 19: Test Your API

1. **Press F5** to run your project
2. Your browser should open to `https://localhost:XXXX/swagger`
3. You should see **AuthenticationController** and **AccountsController** in Swagger UI

Phase 10: Database Setup (IMPORTANT!)

Step 20: Create PostgreSQL Tables

Your API won't work without these tables! Run this SQL in your PostgreSQL database:

```sql
```

```sql
CREATE TABLE Users (
    UserId SERIAL PRIMARY KEY,
    Username VARCHAR(50) UNIQUE NOT NULL,
    Email VARCHAR(100) UNIQUE NOT NULL,
    Password VARCHAR(255) NOT NULL,
    CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE Accounts (
    AccountId SERIAL PRIMARY KEY,
    UserId INTEGER REFERENCES Users(UserId),
    AccountNumber VARCHAR(20) UNIQUE NOT NULL,
    AccountType VARCHAR(20) NOT NULL,
    Balance DECIMAL(10,2) DEFAULT 0,
    CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE Transactions (
    TransactionId SERIAL PRIMARY KEY,
    AccountId INTEGER REFERENCES Accounts(AccountId),
    Amount DECIMAL(10,2) NOT NULL,
    TransactionType VARCHAR(20) NOT NULL,
    Description TEXT,
    CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

🎉 Congratulations!

You now have a complete Banking API with JWT authentication!

What You Can Test:

1. **Register a user:** POST `/api/authentication/register`
2. **Login:** POST `/api/authentication/login`
3. **Create account:** POST `/api/accounts` (requires token)
4. **Deposit money:** POST `/api/accounts/{id}/deposit` (requires token)
5. **Withdraw money:** POST `/api/accounts/{id}/withdraw` (requires token)
6. **View transactions:** GET `/api/accounts/{id}/transactions` (requires token)

Next Steps:

- Test each endpoint in Swagger UI
- Connect this to your ATM simulator frontend
- Add more banking features!

Need help with testing or connecting to your frontend? Let me know! 🚀