MODERN ARCHITECT SERIES

Modular Minimal API .NET 10

A Step-by-Step Guide to Building Professional, Scalable, and Production-Ready Backend Systems

WRITTEN BY
Afandy Lamusu

FIRST EDITION
2026

Modular Minimal API .NET 10

Afandy Lamusu

Table of Contents

Modular Minimal API .NET 10

A Step-by-Step Guide to Building Professional, Scalable, and Production-Ready Backend Systems

Written by Afandy Lamusu

First Edition | Published February 2026

Published by: Afandy.Lamusu Tech Publishing

GGKEY: UZR01BANXZX

AI-Assisted Content Disclosure

This work was developed with the assistance of artificial intelligence technologies (including Claude by Anthropic). The author, Afandy Lamusu, served as the primary architect, lead editor, and technical validator for all content. All technical concepts, code logic, and structural arrangements were directed and refined by the author to ensure accuracy and original expression.

Technical Disclaimer

The information in this book is sold “as is” and without warranty. While every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the software and hardware products described in it.

Trademarks: Microsoft, .NET, ASP.NET, C#, and Windows are trademarks of Microsoft Corporation. PostgreSQL is a trademark of the PostgreSQL Community Association. Docker is a trademark of Docker, Inc. All other trademarks are the property of their respective owners.

Preface

Welcome to the Future of .NET Backend Development

In the fast-evolving landscape of software engineering, the journey from a beginner to a professional architect is filled with complex choices. Which framework should I use? Which database is the best? How should I structure my code so it doesn't become a "Big Ball of Mud"?

This book, **"Modular Minimal API .NET 10"**, was written to answer those questions.

Our goal is simple: to provide a clear, step-by-step, and beginner-friendly guide to building modern backend systems using the latest and greatest tools in the .NET ecosystem.

Why This Book?

There are countless tutorials and books on .NET, but they often fall into one of two traps: they are either too simple, leaving you with code that isn't production-ready, or they are too complex, overwhelming you with enterprise patterns before you've even mastered the basics.

This book finds the "Sweet Spot." We use **Minimal APIs** for simplicity, **PostgreSQL** for professional data management, and the **Modular Monolith** architecture to ensure your application can grow and scale as your needs change.

What You Will Learn

- **Modern .NET 10:** How to use the latest features of C# and the .NET runtime.
- **Clean Architecture:** How to separate your business logic from your infrastructure.
- **Modular Monolith:** How to build a large application as a set of independent, well-organized modules.

- **Real-World Skills:** JWT Authentication, Structured Logging, Global Error Handling, and Database Best Practices.
- **The Path to Microservices:** When and how to take your modules and turn them into standalone services.

Who Is This For?

- **Beginner to early-intermediate .NET developers.**
- **Junior backend developers** looking to understand architectural patterns.
- **Students** who want to build a portfolio with professional-grade code.
- **Anyone** curious about the Modular Monolith approach.

Let's Get Started

Software engineering is a craft. It's about building things that last and solving problems that matter. I'm excited to be your guide on this journey.

Grab your coffee, fire up your terminal, and let's start building!

Afandy Lamusu
February 2026

Modular Minimal API .NET 10

Section 1: Introduction to Modern .NET Backend

Welcome, future architect! I am thrilled to have you here. If you are reading this, you've likely heard the buzz about .NET, or perhaps you're looking to level up your skills from "just writing code" to "designing systems." You are in exactly the right place.

In this first section, we are going to set the stage. Before we touch a single line of code, we need to understand the what, the why, and the how of the tools we'll be using. Think of this as the foundation of your house. If the foundation is shaky, the most beautiful windows and the strongest roof won't save the building.

So, grab a coffee (or your favorite beverage), and let's dive into the world of modern .NET backend development.

What is .NET 10?

You might be wondering, "Why .NET 10? What happened to the old .NET Framework?"

Let's take a quick trip down memory lane. For years, .NET was "Windows-only." It was heavy, tied to the operating system, and often felt "enterprise-y" and complex. Then came .NET Core—a cross-platform, high-performance revolution. It allowed us to run .NET on Linux, macOS, and in Docker containers.

.NET 10 represents the latest peak of this evolution. It is a unified platform. Whether you are building web apps, mobile apps, desktop software, or cloud-native services, you are using the same high-performance runtime and libraries.

For us, .NET 10 means: - **Performance:** It is incredibly fast. Some of the fastest web benchmarks in the world are held by .NET. - **Simplicity:** With every version, Microsoft has worked to reduce "boilerplate" (the boring, repetitive code you have to write just to get things started). - **Cross-Platform:** You can develop on a Mac and deploy to a Linux

server in the cloud without breaking a sweat. - **Modern C#:** C# 14 (which comes with .NET 10) is a joy to write. It's expressive, safe, and powerful.

Under the Hood: The .NET 10 Runtime

When we say "Performance," we aren't just throwing around buzzwords. .NET 10 introduces significant improvements to the **JIT (Just-In-Time) Compiler** and the **Garbage Collector (GC)**. - **Dynamic PGO (Profile-Guided Optimization):** The runtime watches your code as it runs and recompiles "hot paths" (code that runs frequently) to be even faster. It's like having a mechanic tuning your car engine while you are driving. - **Native AOT (Ahead-of-Time) Compilation:** For scenarios where startup time is critical (like Serverless functions), .NET 10 allows you to compile your app directly to native machine code, removing the need for the JIT entirely. - **DATAS (Dynamic Adaptation To Application Sizes):** The GC in .NET 10 is smarter about memory limits, making it much friendlier to containerized environments like Kubernetes where memory is strictly capped.

> 💡 **Pro Tip** Don't worry about memorizing every new feature of .NET 10. As a beginner, focus on the core concepts of the language and the framework. The "shiny new things" will make more sense once you have a solid grasp of the basics.

What is Minimal API?

In the past, building a web API in .NET involved a lot of ceremony. You needed Controllers, base classes, attributes, and complex routing configurations. While powerful, it was often "too much" for simple services or for those just starting out.

Minimal API was introduced to solve this. It's a way to build HTTP APIs with minimal code and ceremony[1]. It's "expressive" and "fluent."

Think of it this way:

- **Traditional Controllers:** Like a formal 5-course dinner. There are rules, specific plates for everything, and it takes time to set up.
- **Minimal API:** Like a high-end street food stall. It's fast, focused, and gives you exactly what you need without the extra fluff, but the quality is still top-notch.

Deep Dive: How Minimal APIs Work

Minimal APIs aren't just "Controllers without the class." They are built on top of a lower-level construct called the `RequestDelegate`.

When you write `app.MapGet("/", () => "Hello")`, you are directly configuring the ASP.NET Core **Middleware Pipeline**. 1. **Endpoint Routing:** The framework builds a "Route Tree" optimized for speed. 2. **Binding:** It automatically looks at your parameters (e.g., `(int id, HttpContext context)`) and injects the correct values. This is done using highly optimized generated code, avoiding the slower "Reflection" used by traditional Controllers. 3. **Execution:** The lambda function is executed directly.

This reduced overhead means Minimal APIs handle **more requests per second** and use **less memory** than Controllers.

Here is a quick look at how simple a Minimal API can be:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello, World!");

app.Run();
```

That's it! In just a few lines, you have a fully functional web server. As we move forward, we will see how this simple structure can scale to support massive, complex applications.

What is PostgreSQL?

Every great application needs a place to store its data. For this book, we have chosen **PostgreSQL** (often just called "Postgres").

Postgres is an open-source, object-relational database system. It has been around for over 30 years and has earned a reputation for being incredibly reliable, robust, and feature-rich.

Why PostgreSQL instead of SQL Server?

You might expect a .NET book to use Microsoft SQL Server. While SQL Server is a fantastic database, Postgres is often the better choice for modern developers, especially those just starting out:

1. **Truly Open Source:** Postgres is free and has no licensing costs. You can run it anywhere without worrying about "Enterprise Edition" fees.
2. **Docker Friendly:** It is extremely lightweight and easy to run in a Docker container (which we will do in Section 4).
3. **JSON Support (JSONB):** This is a killer feature. Postgres allows you to store unstructured JSON data in a column (`JSONB`) and index specific fields inside that JSON. This gives you the flexibility of NoSQL (like MongoDB) within a strict relational database.
4. **Industry Standard:** From tiny startups to giants like Instagram and Netflix, Postgres is everywhere. Learning it is a high-value skill.

> 🧠 **Architecture Insight** In modern development, we often want our infrastructure (like our database) to be as "portable" as possible. Using Postgres ensures that our app can run on any cloud provider (AWS, Azure, Google Cloud) or even on a cheap Linux server.

What is a Modular Monolith?

This is the "meat" of our architectural journey. To understand a Modular Monolith, we first have to talk about its two "rivals": the **Monolith** and **Microservices**.

The Traditional Monolith (The Big Ball of Mud)

Most beginners start here. You have one project, one database, and all your code is mixed together. - **The Problem:** As the app grows, it becomes a "Big Ball of Mud." If you change the code for "User Login," you might accidentally break the "Shopping Cart" because everything is tightly coupled.

Microservices (The Distributed Headache)

The industry went crazy for microservices a few years ago. The idea is to break the app into many tiny, independent services that talk to each other over the network. - **The Problem:** It is **complex**. You need to handle network failures, distributed transactions, service discovery, and complex deployment pipelines. For a small team or a beginner, it is often overkill.

The Modular Monolith (The Sweet Spot)

A **Modular Monolith** gives you the best of both worlds. It is **one single application** (one project to deploy, one database), but internally, it is strictly divided into **independent modules**.

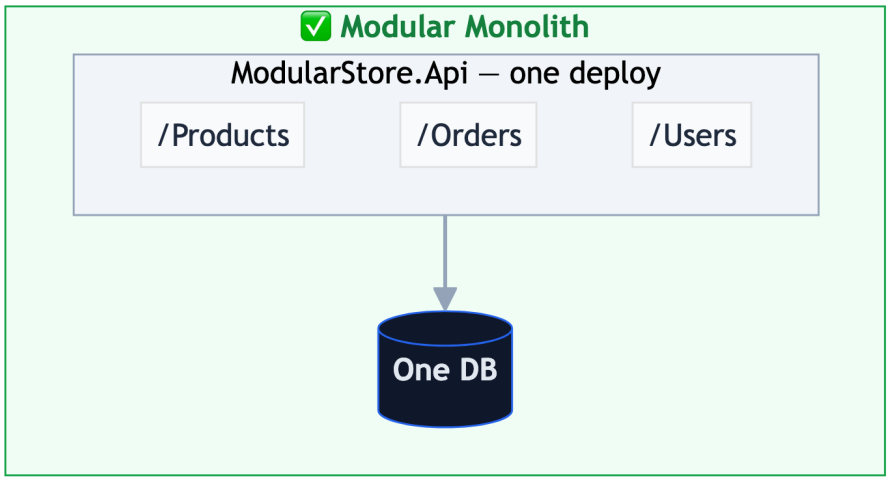
- The "Products" module doesn't know how the "Orders" module works.
- They communicate through well-defined "contracts" (interfaces).
- If you ever really need to move to microservices, you just "cut" the module out and move it. It's already designed to be independent!

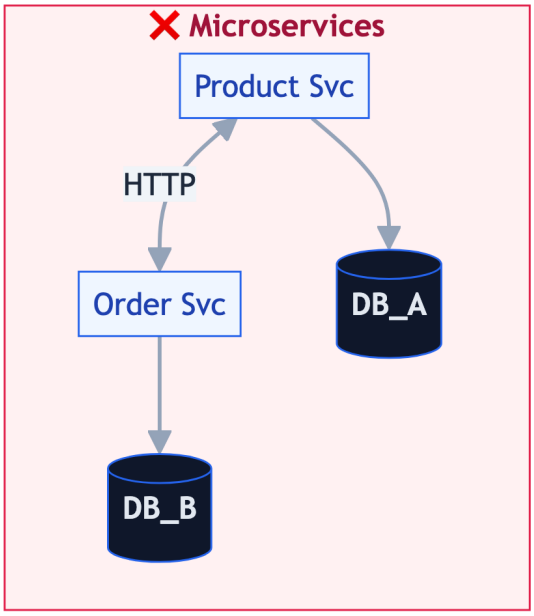
Modular Monolith vs Microservices: A Simple Comparison

Feature	Monolith	Modular Monolith	Microservices
Complexity	Low	Medium	High
Deployment	Simple (One app)	Simple (One app)	Hard (Many apps)
Scalability	Hard	Medium	Easy
Data Integrity	Easy (One DB)	Easy (One DB)	Hard (Many DBs)
Developer Speed	Fast (at start)	Sustainable	Slow (setup overhead)

Architecture at a Glance

The three approaches look very different when you visualize them:


✅ Modular Monolith
ModularStore.Api — one deploy
/Products
/Orders
/Users
One DB


❌ Microservices
Product Svc
HTTP
Order Svc
DB_A
DB_B

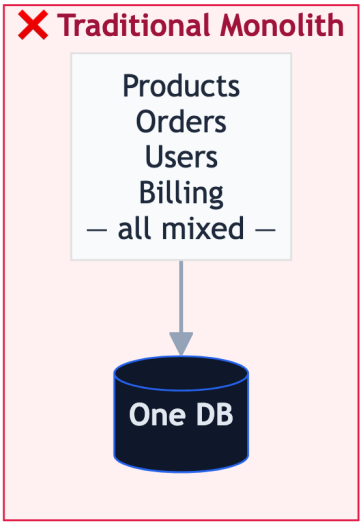

❌ Traditional Monolith
Products
Orders
Users
Billing
— all mixed —
One DB

Figure 1.1 — Architectural Approaches: Traditional Monolith vs Microservices vs Modular Monolith

When to use a Modular Monolith?

You should choose a Modular Monolith when: - You are starting a new project (Greenfield). - You want the speed of a monolith but the organization of microservices. - You have a small to medium-sized team. - You want to keep your deployment simple. - You want to be "Microservices-Ready" without the "Microservices-Pain."

Summary of Section 1

We've covered a lot of ground! We've identified our tech stack (.NET 10, Minimal API, Postgres) and our architectural strategy (Modular Monolith).

In the next section, we'll go even deeper into the principles of software architecture. We'll learn about "Separation of Concerns" and the "Dependency Rule"—the secret ingredients that keep our code clean and manageable.

Are you ready to become an architect? Let's move on!

> ⚠ **Common Mistake** Don't over-engineer! Many developers start a project and immediately think they need microservices because "Netflix uses them." Remember: Build for the requirements you have today, but design so you can change tomorrow. A Modular Monolith is the perfect way to do that.

Next: SECTION 2 – Understanding Software Architecture Fundamentals

Section 2 – Understanding Software Architecture Fundamentals

Before we start building our modular monolith, we need to talk about the "Blueprints." In the world of software, we call these blueprints **Architecture**.

Many beginners think that "architecture" is just for big companies or complex systems. That couldn't be further from the truth! Architecture is simply the art of making decisions early so that you don't suffer later. It's about organizing your code so that when your boss (or your future self) asks for a new feature, you don't have to rewrite the entire application.

In this section, we will break down the fundamental concepts that make up a professional .NET backend.

What is Software Architecture?

Imagine you are building a small shed in your backyard. You might not need a detailed blueprint. you can probably "wing it" with some wood, nails, and a hammer. If you mess up a measurement, it's not the end of the world.

Now, imagine you are building a **skyscraper**.

If you try to "wing it" with a skyscraper, it will collapse before you reach the third floor. You need to know where the plumbing goes, how the electricity is wired, and how the foundation will support the weight of 50 floors.

Software Architecture is the blueprint for your skyscraper.

It defines: 1. **Organization:** Where does the code live? 2. **Communication:** How do different parts of the app talk to each other? 3. **Constraints:** What are the "rules" that developers must follow?

> 🧠 **Architecture Insight** Good architecture is not about creating the "perfect" system. It's about creating a system that is **easy to change**. Requirements change constantly in the real world. If your code is easy to change, you have succeeded as an architect.

Layered Architecture (The Classic)

For decades, the standard way to build .NET apps was the **Layered Architecture** (also known as N-Tier Architecture). You've probably seen it before:

1. **Presentation Layer (API):** Handles HTTP requests and returns responses.
2. **Business Logic Layer (BLL):** Where the "rules" of your app live (e.g., "A discount is only applied if the cart is over $50").
3. **Data Access Layer (DAL):** Talks to the database.

It looks like this:

Figure 2.1 — Layered (N-Tier) Architecture: Presentation → Business Logic → Data Access → Database

The Problem with Traditional Layering

In this model, the **Business Logic depends on the Database**.

Think about that for a second. Your "Core Business Value" is tied to a specific technology (SQL Server, Postgres, etc.). If you want to change how you store data, or if you want to write a "Unit Test" for your business logic without a real database, it becomes very difficult. This is called **Tight Coupling**.

Clean Architecture (Simplified)

To fix the problems of layered architecture, smart engineers (like Robert C. Martin[2]) proposed **Clean Architecture**. The goal is simple: **Put the Business Logic at the center, and make everything else depend on it.**

In Clean Architecture, we think in circles, not layers. The most critical rule is: **dependencies only point inward.** The Domain is at the center and knows nothing about the outside world.

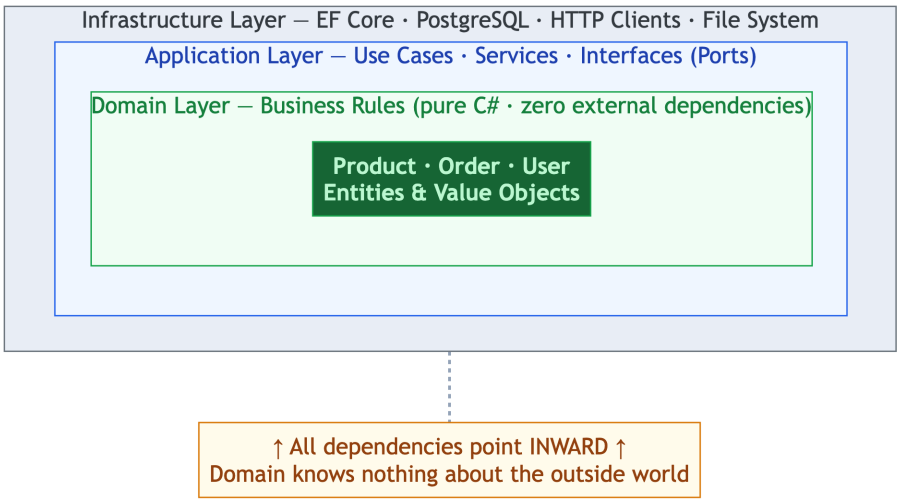


Figure 2.2 — Clean Architecture: Domain at the Center, All Dependencies Pointing Inward

The Dependency Rule & Inversion of Control (IoC)

This is the most important rule in this entire book: **Dependencies must only point inwards.**

- The **Domain** knows nothing about the outside world. It doesn't know about Postgres, it doesn't know about JSON, and it doesn't know about the Internet.
- The **Application** layer depends on the Domain.
- The **Infrastructure** and **API** layers depend on the Application and Domain.

But wait, if the `Application` layer needs to save data to a Database (which is in `Infrastructure`), doesn't it need to depend on `Infrastructure`?

This is where **Inversion of Control (IoC)** comes in. 1. The **Application Layer** defines an interface (e.g., `IProductRepository`). This is a "port." 2. The **Infrastructure Layer** implements that interface (e.g., `ProductRepository`). This is an "adapter." 3. At runtime, we "inject" the implementation into the application.

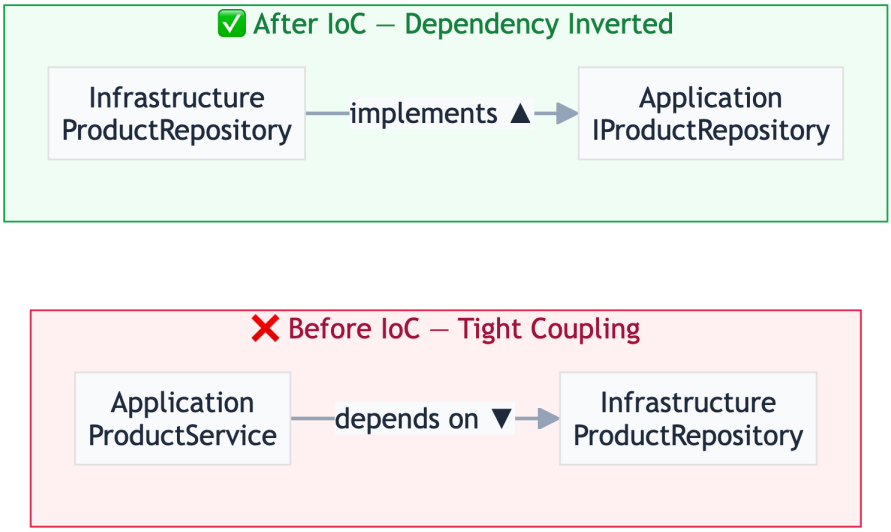


Figure 2.3 — Inversion of Control: Tight Coupling (Before) vs Dependency Inversion (After)

So, at compile time, `Infrastructure` depends on `Application` (to see the interface). The dependency arrow is reversed!

Concrete Code Walkthrough: IoC in Action

To make this concept crystal clear, let's look at how the code is structured.

1. The Domain Layer (The Interface) In this layer, we define what we need, but not how to do it. We use an **Interface**.

```
// ModularStore.Api/Modules/Products/Domain/IProductRepository.cs
namespace ModularStore.Api.Modules.Products.Domain;

public interface IProductRepository
{
    Task<IEnumerable<Product>> GetAllAsync();
}
```

2. The Application Layer (The Service) Our service (which holds the business logic) depends only on the interface. It has no idea that a database even exists.

```
// ModularStore.Api/Modules/Products/Application/ProductService.cs
namespace ModularStore.Api.Modules.Products.Application;

public class ProductService
{
    private readonly IProductRepository _repository;
```

```
    public ProductService(IProductRepository repository)
    {
        _repository = repository; // We "inject" the interface here
    }

    public async Task<IEnumerable<ProductResponse>> GetProductsAsync()
    {
        // We just call the method on the interface
        var products = await _repository.GetAllAsync();
        return products.Select(p => new ProductResponse(p.Id, p.Name, p.Price));
    }
}
```

3. The Infrastructure Layer (The Implementation) This layer is the only one that knows about the database. It "implements" the interface.

```
// ModularStore.Api/Modules/Products/Infrastructure/ProductRepository.cs
using Microsoft.EntityFrameworkCore;
using ModularStore.Api.Modules.Products.Domain;

namespace ModularStore.Api.Modules.Products.Infrastructure;

// Notice how this layer "points back" to the Domain layer
// to see the interface it needs to implement.
public class ProductRepository : IProductRepository
{
    private readonly ProductsDbContext _context;

    public ProductRepository(ProductsDbContext context) => _context = context;

    public async Task<IEnumerable<Product>> GetAllAsync()
    {
        // Here is the actual database logic using EF Core
        return await _context.Products.ToListAsync();
    }
}
```

> 💡 **The Result** At compile time, the `ProductService` (Application) does **not** depend on the `ProductRepository` (Infrastructure). In fact, it doesn't even know it exists! This is "Inversion of Control."

Separation of Concerns (SoC)

Separation of Concerns is a fancy way of saying: **"One piece of code should do one thing, and do it well."**

Think of a professional kitchen: - The **Chef** cooks the food (Business Logic). - The **Waiter** takes the order (API / Presentation). - The **Dishwasher** cleans the plates (Infrastructure).

If the Chef has to stop cooking to go wash dishes or wait tables, the kitchen slows down and eventually stops working. In code, if your "SaveUser" function is also validating the email format, calculating a discount, and sending an email, you have violated SoC.

Deep Dive: Domain Logic vs. Application Logic

It is often confusing to decide what goes where. Here is the litmus test:

Domain Logic (Enterprise Business Rules): Rules that are true regardless of whether this is a web app, a console app, or a mobile app. - Example: "A user cannot have a negative balance." - Example: "The total price is the sum of all item prices plus tax."

Application Logic (Application Business Rules): Rules that are specific to this application's workflow. - Example: "When a user registers, send them a welcome email." - Example: "When an order is placed, log the event to a file." - Example: "Authenticate the user using a JWT token."

If you switch from a Web API to a CLI tool, your **Domain Logic** stays exactly the same, but your **Application Logic** might change.

Defining Our Layers: Domain vs Infrastructure vs API

In our Modular Monolith, each module (like Products or Orders) will be divided into these internal layers. Let's define them clearly:

1. The Domain Layer (The Heart)

This is where your **Entities** and **Business Rules** live. - **Example:** A `Product` class with properties like `Name` and `Price`. - **Logic:** "A product price cannot be negative." - **Tools:** Just plain C# classes (often called POCOs - Plain Old CLR Objects). No NuGet packages allowed here if possible!

2. The Application Layer (The Brain)

This layer coordinates the work. It defines **what** the system should do. - **Example:** "Create a new product using the data from the request." - **Tools:** Interfaces (like `IProductRepository`), DTOs (Data Transfer Objects), and Handlers.

3. The Infrastructure Layer (The Tools)

This layer talks to the "Outside World." - **Example:** The actual code that uses Entity Framework Core to save a product to PostgreSQL. - **Example:** A service that sends an SMS via Twilio. - **Tools:** EF Core, PostgreSQL driver, File System access, 3rd party APIs.

4. The API Layer (The Face)

This is how users interact with your app. - **Example:** The Minimal API `MapPost` endpoint that receives JSON. - **Tools:** ASP.NET Core, Swagger/OpenAPI.

A Real-World Example: Adding a Product

Let's see how these layers work together when a user wants to add a new product to our store:

1. **API Layer:** Receives a POST request with JSON data. It converts the JSON into a `CreateProductRequest` object.
2. **Application Layer:** Receives the request. It says, "Okay, I need to create a `Product` entity."
3. **Domain Layer:** The `Product` entity is created. It checks: "Is the price valid?" If yes, it returns the object.
4. **Application Layer:** Now it says, "I need to save this. Hey `IProductRepository`, please save this product." (Note: It doesn't know how it's saved, just that it is saved).
5. **Infrastructure Layer:** The concrete implementation of the repository uses EF Core to write the data to **PostgreSQL**.
6. **API Layer:** Returns a `201 Created` response to the user.

> 💡 **Pro Tip** Notice how the "Application Layer" uses an **Interface** (*`IProductRepository`*). It doesn't care if the implementation uses Postgres, SQL

> Server, or even a simple text file. This is the power of **Dependency Injection**, which we will cover soon!

Summary of Section 2

You now understand the "why" behind software architecture. We are moving away from the "Big Ball of Mud" and toward a "Clean" and "Modular" design.

Key Takeaways: - **Architecture** makes code easy to change. - **The Dependency Rule:** Always point inwards toward the Domain. - **Separation of Concerns:** Keep your "Cooking," "Waiting," and "Cleaning" separate. - **Layers:** Domain (Heart), Application (Brain), Infrastructure (Tools), and API (Face).

In the next section, we will take these principles and apply them specifically to the **Modular Monolith** pattern. We'll see how to draw boundaries between different parts of our business.

> ⚠ **Common Mistake** Don't create layers just because "a book said so." Always ask: "Does this layer help me keep my code organized, or is it just making things more complicated?" For very small projects, Clean Architecture might be overkill. But for anything that needs to grow, it is a lifesaver.

---

1. Microsoft Learn, Minimal APIs Overview, https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis.

2. Robert C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design (Prentice Hall, 2017).