FROM DEV TO ARCHITECT

The Software Architecture Starter Kit for .NET Developers

BY KRISTIJAN KRALJ

2025

WHAT IS SOFTWARE ARCHITECTURE?

If you ask 100 developers a seemingly simple question:

➥ **“What is software architecture?”**

They will all probably give you different answers. That’s because “architecture” means different things to different developers.

Why?

Because they are all working on projects that are:

- Different in size,
- In a different domain,
- With different technical and non-technical requirements and constraints.

No wonder there would be so many answers.

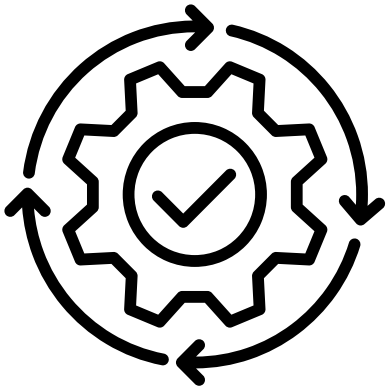
For example, the architecture is:

- The big picture
- The system overview
- Plans how to implement a solution
- Abstract view of the system
- Vision and strategy of software projects
- ...and so on.

But, even though the term architecture is hard to identify, you can think about architecture in two ways. As a noun and as a verb.

- **Architecture as a noun** - The software architecture is the structure of a system. It contains software elements, the externally visible properties of those elements, and relationships between the elements.

- **Architecture as a verb -** The software architecture is about translating the functional requirements, non-functional requirements, and constraints into a technical solution.



In the past few years, I have had to make many decisions that impacted the overall design of the system.

Even though I wasn't an architect, but rather a software developer.

Also, lately I have spent a lot of time reading about software architecture.

Now, you may wonder: "Why do I need to learn about software architecture? I'm a software developer, not an architect."

Well, if you are like me, you will be forced to think about the bigger picture for two reasons:

1. You are a senior developer (or someone who needs to make those decisions in your code)
2. There is no dedicated architect in your team.

So, to help you in learning more about software architecture, I have created this eBook. You will learn more about:

1. How to decide what non-functional requirements (scalability, performance...) are important for your application based on the business requirements you get
2. What are the most famous architectural styles (clean architecture, vertical slice, microservices...), and what are the differences
2. How and why to document your team's tech decisions in a simple, quick, but powerful way
2. 14 of my favorite software architecture resources for .NET developers (books, GitHub repos with fully developed applications...)

The eBook has plenty of visuals and practical examples.

Let's get started!

HOW TO DECIDE SOFTWARE ARCHITECTURE BASED ON REQUIREMENTS?

What's the first thing you get at the beginning of every software project?

➥ **The requirements.**

A description of how the finished application needs to behave.

Then, you need to analyze them. And make a wild guess at how long it will take to deliver the completed app.

Those requirements are often referred to as functional requirements.

But there is also another set of "invisible" requirements called non-functional requirements or architectural characteristics.

➥ **"Invisible" because they are not explicitly mentioned.**

But you still need to define them as they affect the overall user experience and operational sustainability of the system.

And ensure that the final product meets all expectations regarding performance, security, and other quality attributes.

There are a lot of architectural characteristics (non-functional requirements) that might affect how your system is shaped.

Too many to be covered in a single email.

But here are some that you have probably heard of:

Security - how secure the system is as a whole.

Availability - what percentage of time does the system need to be online (e.g. 99.999%)

Scalability - how well the system performs when the number of users increases.

Performance - how fast is the system?

Reliability - does the system need to be fail-safe? Is it mission-critical or impacts lives?

Testability - how easy is to test the system.

Deployability - how easy is to deploy the system.

Modularity - how separated are components.

Now, how do you define which architectural characteristics your app architecture needs to have?

By analyzing requirements.To make this as practical as possible, let's analyze some requirements to come up with architectural characteristics.

"We expect a burst in website visits while the offline event takes place."

- **Architectural characteristic:** Scalability - The system must be able to respond to a sharp increase in users visiting the site.

"We want to store the users' credit card information."

- **Architectural characteristic:** Security - Sensitive information like credit card details must lead to increased security of the system.

"We need to release the app to market in 6 months. However, the budget is limited."

- **Architectural characteristic:** Simplicity - If the project has time and money constraints, the architecture of the application should be as simple as possible to satisfy those constraints.

"The system must process 1,000 transactions per second."

- **Architectural characteristic:** Performance - It states how quickly the system must respond and process requests.

Even if you are not a software architect, here's why it's important to be aware of architectural characteristics:

- They directly impact the user experience and system performance, guiding your code to meet critical standards such as security, reliability, and efficiency.

- Understanding these requirements ensures the decisions you make align with the overall quality goals and sustainability of the project, leading to a more robust and user-friendly product.

Once you decide on the architectural characteristics, you may proceed to create a solution and start coding.

Next, you will learn about the different ways to structure your solution. And make a decision based on their benefits and drawbacks.

HOW TO ORGANIZE YOUR SOLUTION?

Ever tried solving a jigsaw puzzle?

A single puzzle doesn't tell you anything, but it's part of something bigger. And, when you assemble them correctly, they form a beautiful picture.

Well, software architecture is just like solving a jigsaw puzzle. Except you don't have a box where you can see the completed picture.

But you have different parts of your code you need to fit together to create a functional system.

You can split your code inside the .NET solution into different ways.

And the way the code is organized and deployed is defined by different architectural styles.

ARCHITECTURAL STYLES: THE 2 CATEGORIES

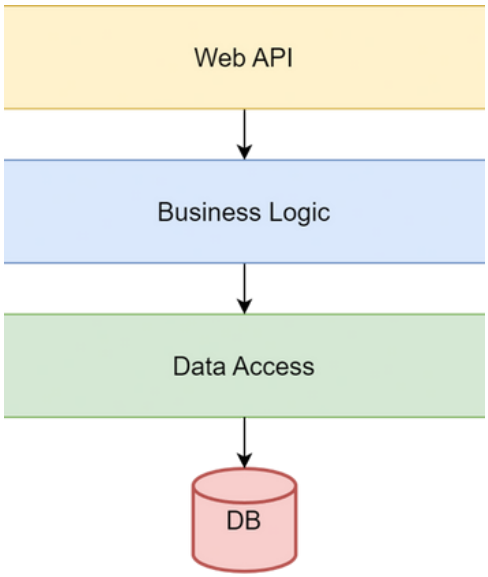
There are two categories by which you can classify an architectural style:

1. **Partitioning** - represents how you divide the code inside the application. The common ways are:
   - Partitioning by technical concerns - grouping code into layers (API, Business Logic, Data, Infrastructure).
   - Partitioning by domain (business) concerns - grouping code by features (Feature 1, Feature 2...)

2. **Deployment model** - represents how the system is deployed. You can deploy it in one of the two ways:
   - Monolith - all code is deployed as one single unit.
   - Distributed - the code is deployed in multiple units, which often results in multiple different services being deployed independently.

With those categories in mind, let's explain the most popular architecture styles used across the industry.

N-TIER ARCHITECTURE



- **Partitioning:** By technical concerns
- **Category:** Monolith

In N-tier architecture, also known as layered architecture, your code is divided into layers such as:

- Presentation,
- Business logic,
- Data access.

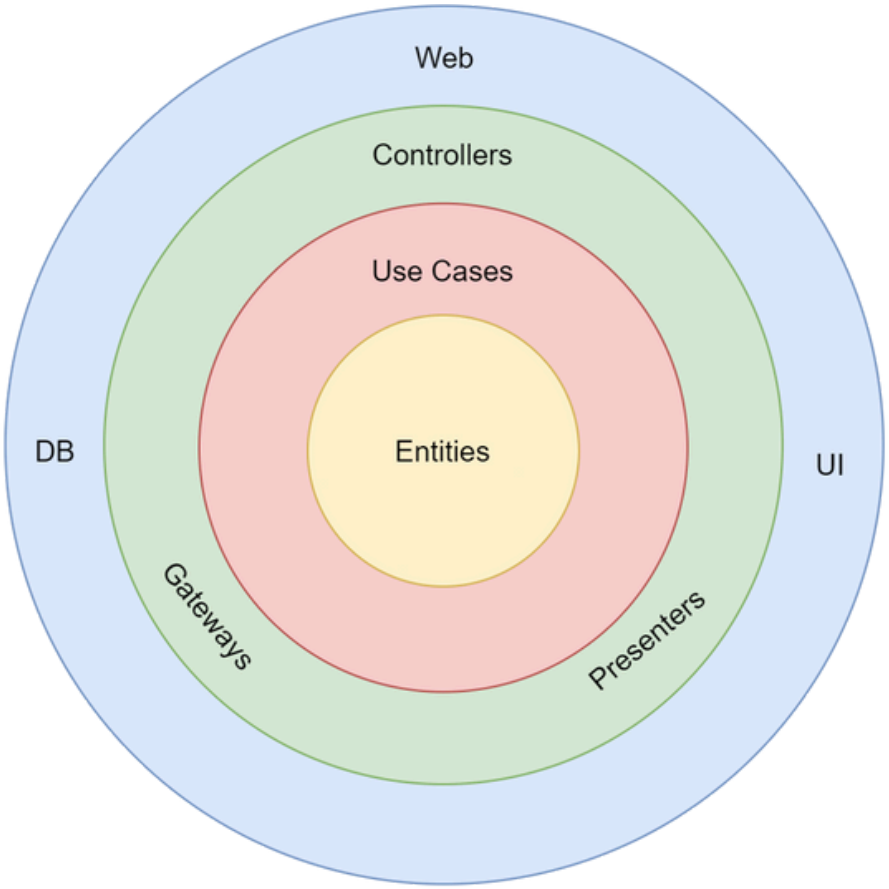
Each layer depends on the layer below it.

This separation helps manage complexity by enforcing a clear structure and separation of concerns.

- **The biggest benefit:** Easy to implement and use.
- **The biggest drawback:** Can easily introduce tight coupling between layers. And create what is known as the "Big Ball of Mud".

CLEAN ARCHITECTURE (ONION ARCHITECTURE)

- **Partitioning:** By technical concerns
- **Category:** Monolith



Clean architecture is designed to create a clear separation of concerns, making the codebase flexible, testable, and maintainable.

It organizes code into concentric circles, with each circle representing a different layer of the application:

- Entities
- Use cases
- Interface adapters
- Frameworks and drivers

It's similar to the N-Tier application since it uses layers to separate concerns.

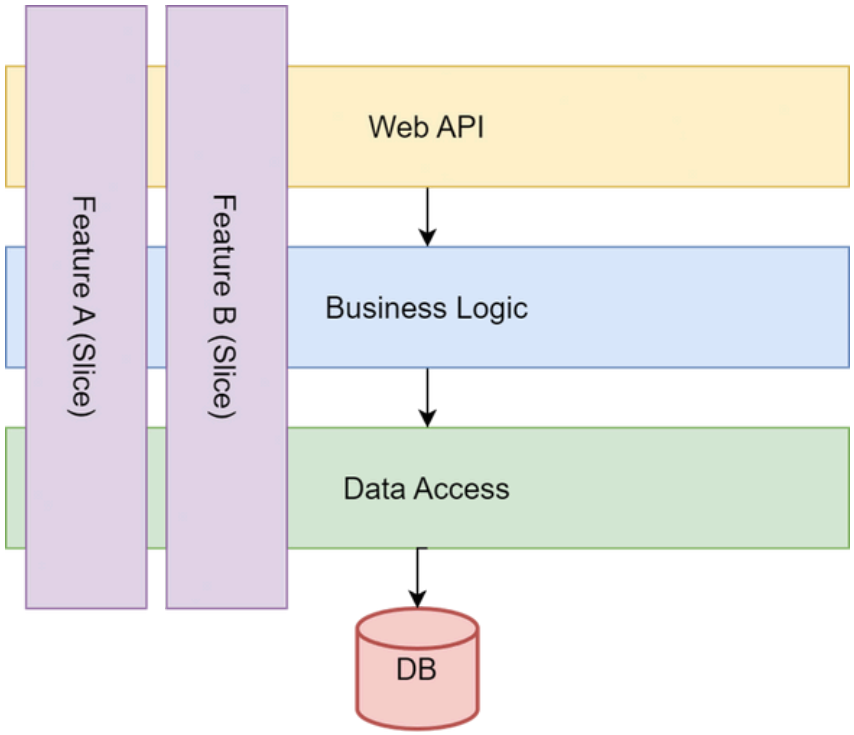
However, communication and dependency between layers is different:

- In N-Tier, each layer depends on the layer below.
- In Clean Architecture, everything depends on domain layers (Entities and Use Cases).

- **The biggest benefit:** Maintainability and testability, with a clear separation of concerns.
- **The biggest drawback:** Can be complex to set up initially and may require more upfront design effort.

VERTICAL SLICE ARCHITECTURE

- **Partitioning:** By domain
- **Category:** Monolith



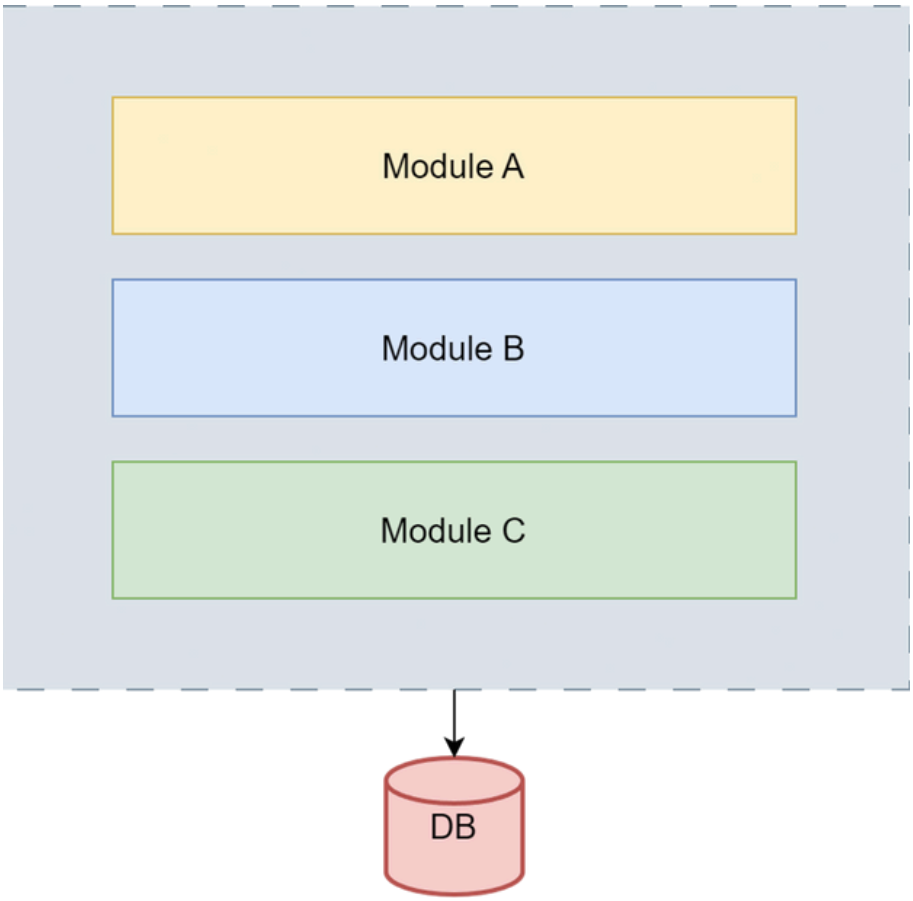
Vertical slice architecture takes a different approach. It organizes code around features or business use cases rather than technical layers.

Each slice contains all the necessary code for a single feature. This creates a self-contained group of code that can include everything from the API to the data storage.

- **The biggest benefit:** Easy to maintain and work in a team, since slices are independent.
- **The biggest drawback:** Similar functionality might be duplicated across slices, leading to redundancy and multiple implementations of the same thing.

MODULAR MONOLITH

- **Partitioning:** By domain
- **Category:** Monolith



Modular monolith architecture combines the benefits of layered/clean and vertical slice architectures. It structures the application into distinct modules, each representing a business domain, while deploying as a single unit.

The modules are most commonly organized into separate .NET projects inside a single solution.

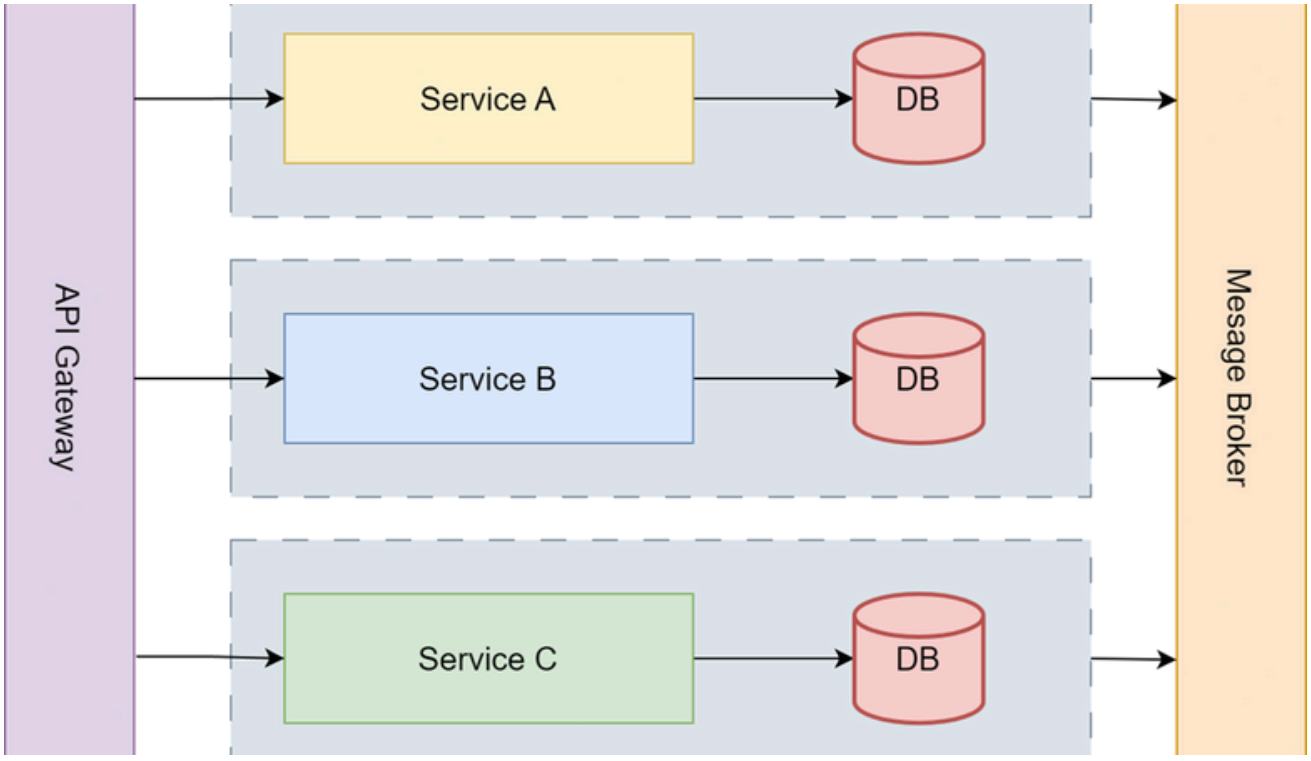
- **The biggest benefit:** modules have well-defined boundaries, making the system easier to maintain. Common functionalities can be shared across modules through a shared project.

- **The biggest drawback:** If you don't manage and split them correctly, modules can become tightly coupled, so you lose some of the benefits of this architecture.

MICROSERVICES

- **Partitioning:** By domain

- **Category:** Distributed



Microservices architecture takes the concept of partitioning further by deploying each domain module as an independent service.

This creates a distributed system where each service operates in its process and communicates with other services over a network.

- **The biggest benefit:** Services can be scaled independently. Failure in one service reduces the impact on the overall system.

- **The biggest drawback:** Complexity to build, deploy and maintain the whole system.

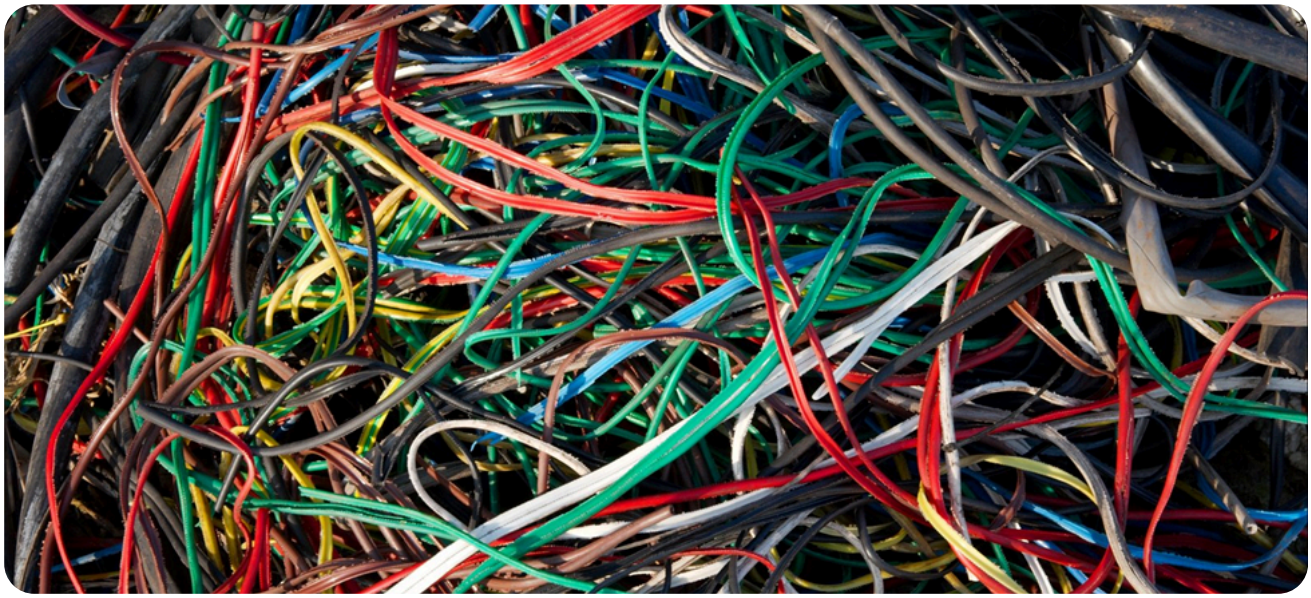
If you need code examples, don't worry.

At the end of the book, I'll share **popular GitHub repositories** that implement the architecture styles defined above.

HOW TO DOCUMENT YOUR TEAM'S TECH DECISIONS

They say that the best codebases are self-documenting.

But what they don't tell you is that most codebases look like this:



And while one of the best ways to document your code is automated tests, there is one type of document that I started to use recently.

↳ **And I like it a lot.**

It's Architecture Decision Record (ADR).

ADR is a document that describes a specific architectural decision.

You write one for every architectural decision you make. And over time, they form a log of past decisions.

But why is it crucial to document your architectural decisions?

Because it stores important decisions you make.

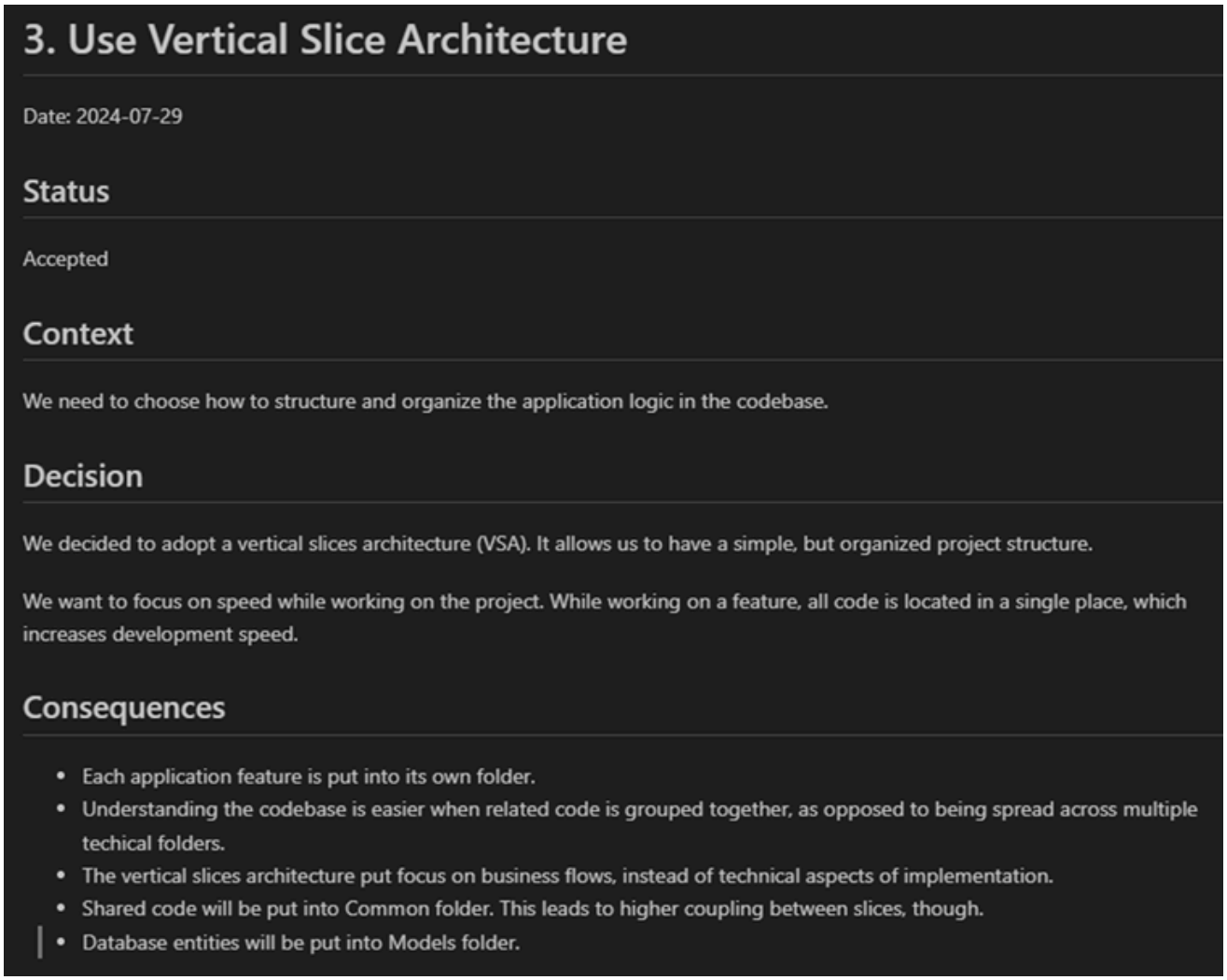
At the time of making a decision, you know why you made it.

↳ **At least I hope you do.**

But when you use ADR, anyone new who joins the project in the future can read and understand why the application looks and works the way it does.

It serves as a great reminder of past decisions you and your team made.

Here is an example of an ADR.



3. Use Vertical Slice Architecture

Date: 2024-07-29

Status

Accepted

Context

We need to choose how to structure and organize the application logic in the codebase.

Decision

We decided to adopt a vertical slices architecture (VSA). It allows us to have a simple, but organized project structure.

We want to focus on speed while working on the project. While working on a feature, all code is located in a single place, which increases development speed.

Consequences

- Each application feature is put into its own folder.
- Understanding the codebase is easier when related code is grouped together, as opposed to being spread across multiple techical folders.
- The vertical slices architecture put focus on business flows, instead of technical aspects of implementation.
- Shared code will be put into Common folder. This leads to higher coupling between slices, though.
- Database entities will be put into Models folder.

Let's go over the main sections:

- **Title** - It should be meaningful and short. A good title makes it easy to figure out what the ADR is about. For example, "3. Use Vertical Slice Architecture". After title, put the date when the ADR was created.

- **Status** - The status informs where the team stands on the architectural decision. If there is no particular decision-making process, you can put Accepted. And move on. But if there is a process, you can use the following status flow: Request for Comment → Proposed → Accepted.

- **Context** - This section explains the circumstances that led you to make a decision. The Context answers, "Why did we have to make this decision?"

- **Decision** - This section clearly explains the decision. Use authoritative voice in this part, e.g. "We will use..." (instead of weak language such as "We think" or "We believe"). The Decision section is also the place to explain why you are making the particular decision. In other words, provide justification.

- **Consequences** - Every decision has outcomes. In this section, you state the consequences of your decision. Both positive and negative. You can assess whether the positive outcomes outweigh the negative outcomes.

You may add other sections, such as Alternatives, Governance, and Notes, but the less you have to fill out, the less friction you have. And the more likely you will stick with this habit.

What are the benefits of using ADRs?

Besides storing significant architectural decisions, there are a few other benefits:

- Easy to start with - you don't need any tool to use ADR. Using simple text files is enough.

- Lightweight - you can express your architectural decision using a simple format.

What happens when architecture changes?

Sooner or later, the architecture will change. And the ADR may become obsolete. In that case, create a new ADR and communicate the change.

↪ But don't modify the existing ADRs.

They represent a chronological evolution of your app architecture.

Where to store ADRs?

You can store them in a team wiki or other shared documentation place. I prefer to store them in the GitHub repo, close to the code itself.

↪ I use the markdown format for writing ADRs.

The filename has the following convention:

- ADR number
- Lowercase title, separated by hyphens

↪ **Example:** 0003-use-vertical-slice-architecture.md

14 OF MY FAVORITE SOFTWARE ARCHITECTURE RESOURCES FOR .NET DEVELOPERS

When you are just starting as a developer, you are not expected to do much other than write code and complete tasks.

But when you progress in your career, you will get a chance to:

- Be involved in making decisions that impact the whole application architecture.
- Design and implement a whole feature within the app.
- Create a brand-new solution from scratch.

To be able to do that, you need to learn from:

- **Experience** - Doing is the best way to learn.
- **Practice** - Complete side projects or practice katas (more on this at the end).
- **Code examples** - Learn from articles with code examples or GitHub repositories.
- **Books** - There are thousands of developers who experienced the problems you have right now. Luckily, a few of them have written a book about the solutions.

The most important thing to become better at anything is experience.

The same is true for software architecture.

I can't give you the experience you need throughout this email.

But what I can give you is a collection of books, code examples, and ways to practice. So you get better at software architecture.

↳ **Let's dive into resources.**

BOOKS

> *"Sleep is good, he said, and books are better."*
> *- George R. R. Martin*

Here are some software architecture books that you will find helpful:

1. **Software Architecture for Developers** - In an easy-to-read manner, it explains what is the role of a software architect in a team and how it differs from a software developer.

2. **Head First Software Architecture** - If you have read other Head First books, you'll love this one. It's full of illustrations, images, and exercises.

3. **Fundamentals of Software Architecture: An Engineering Approach** - The focus is on architectural principles that are language/framework agnostic.

4. **Software Architecture: The Hard Parts** - This is a sequence to the previous book. The authors cover the parts that are left out from the first book.

5. **System Design Interview – An insider's guide** - It explains in a practical way how to design large systems and what to consider in terms of requirements. Covers how to design popular services, such as YouTube, Google Drive, URL Shortener, and Notification system. Fun fact: I used this as one of the resources to design a notification system in one of the previous projects.

6. **System Design Interview – An Insider's Guide: Volume 2** - Second part of the system design interview questions. It covers some more scenarios.

7. **Designing Data-Intensive Applications** - I haven't read it yet. But from what I read all over the internet, this is a Bible for backend developers. Many people claim it's the best book to read to become a better backend developer.

CODE EXAMPLES (GITHUB REPOSITORIES)

Some people collect watches. Some people collect NFTs. I collect useful GitHub repositories for .NET developers.

If you want to improve your architecture skills, here are 5 you're gonna love:

1. **Evolutionary architecture by example** - It shows how to evolve an application from a simple solution to advanced microservices.

2. **Modular monolith application with DDD** - Full modular monolith application with Domain Driven Design approach.

3. **.NET 9 starter kit with multitenancy support** - .NET 9 Starter Kit (Web API + Blazor Client) with Multitenancy Support.

4. **eCommerce microservice .NET application** - a microservices .NET application implementing an eCommerce site.

5. **Vertical slice architecture example** - Sample application showing how to implement vertical slice architecture.

PRACTICE

To get better at system design and software architecture, you need to practice in real-world scenarios.

But what can you do when you don't have a chance to do it at work?

You can use Software Architecture katas. Katas are software development exercises that help developers improve their skills through practice and repetition.

➥ **The site contains a list of many application scenarios that you need to design.**

Another great resource is KataLog, a collection of katas solved by winners and finalists from the O'Reilly Software Architecture Kata competition.