We preferred the Monolithic system architecture approach in our project. If we explain a little bit of monolithic applications, they are designed to handle multiple related tasks. These are typically complex applications involving several tightly coupled functions. Monolithic architecture means that the software is designed as self-contained. We can also say that it is formed as a "single piece" in line with a standard. The components in this architecture are designed as interdependent rather than loosely coupled. Monolithic architecture is the development of all functionality in a self-sufficient application under one roof.

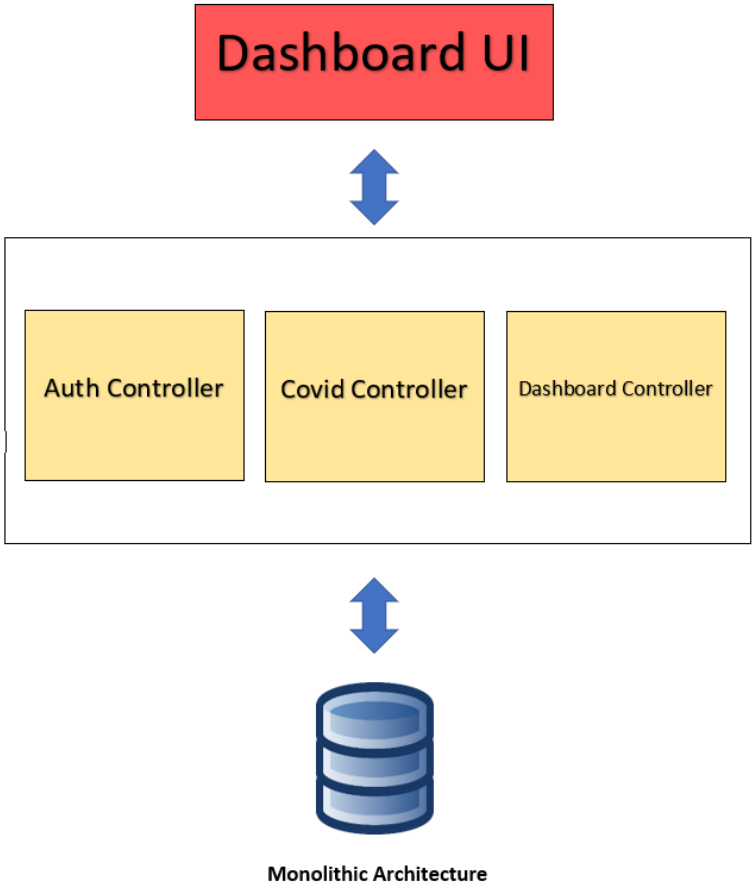
To talk about the advantages of Monolithic architecture;

Manageability and monitoring are easy.

It is easy to develop and maintain for small projects. Application can be developed quickly.

The interoperability of the functionalities is consistent.

Transaction management is easy.



Monolithic Architecture

Some of the design patterns we used in our project.

MVC

The entire backend part of our project is made in Model View Controller architecture.

Dependency injection

We used this design pattern in the process of including the objects available in .net technology to our project.

Decorator pattern

This design pattern is used to map the routing system in the web service in our project and to determine the routing criteria.

Singleton pattern

A single object created using this design pattern is used in the whole system so that the system does not create an object to connect the database each time and does not create security problems.

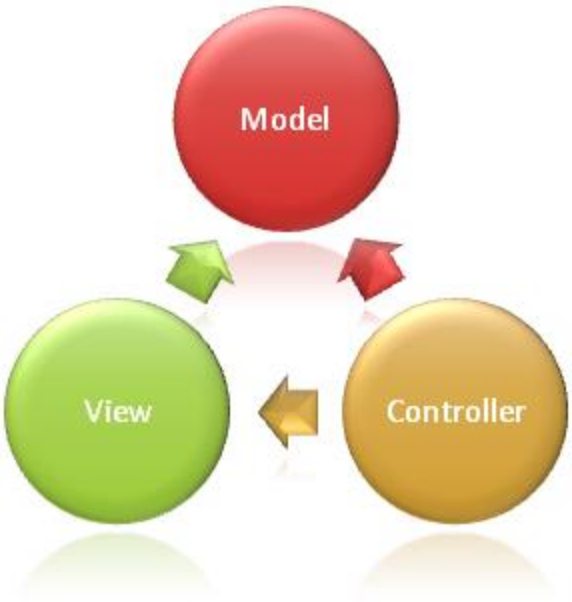
If it is necessary to give brief information about these.

ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

What is the MVC pattern?

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something (model, view, or controller) that has a single job. It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, an object containing business logic must be modified every time the user interface is changed. This often introduces errors and requires the retesting of business logic after every minimal user interface change.

What is ASP.NET Core MVC

The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core.

ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enables a clean separation of concerns. It gives you full control over markup, supports TDD-friendly development and uses the latest web standards.

Model Responsibilities

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application. Strongly-typed views typically use ViewModel types designed to contain the data to display on that view. The controller creates and populates these ViewModel instances from the model.

View Responsibilities

Views are responsible for presenting content through the user interface. They use the Razor view engine to embed .NET code in HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a View Component, ViewModel, or view template to simplify the view.

Controller Responsibilities

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

Dependency injection

In software engineering, **dependency injection** is a technique in which an object receives other objects that it depends on. These other objects are called dependencies. In the typical "using" relationship the receiving object is called a client and the passed (that is, "injected") object is called a service. The code that passes the service to the client can be many kinds of things and is called the injector. Instead of the client specifying which service it will use, the injector tells the client what service to use. The "injection" refers to the passing of a dependency (a service) into the object (a client) that would use it.

The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.

Dependency injection is one form of the broader technique of inversion of control. A client who wants to call some services should not have to know how to construct those services. Instead, the client delegates the responsibility of providing its services to external code (the injector). The client is not allowed to call the injector code; it is the injector that constructs the services. The injector then injects (passes) the services into the client which might already exist or may also be constructed by the injector. The client then uses the services. This means the client does not need to know about the injector, how to construct the services, or even which actual services it is using. The client only needs to know about the intrinsic interfaces of the services

because these define how the client may use the services. This separates the responsibility of "use" from the responsibility of "construction".

Singleton pattern

In software engineering, the **singleton pattern** is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system. The term comes from the mathematical concept of a singleton.

Critics consider the singleton to be an anti-pattern in that it is frequently used in scenarios where it is not beneficial, introduces unnecessary restrictions in situations where a sole instance of a class is not actually required, and introduces global state into an application.

Decorator pattern

In object-oriented programming, the **decorator pattern** is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern. The decorator pattern is structurally nearly identical to the chain of responsibility pattern, the difference being that in a chain of responsibility, exactly one of the classes handles the request, while for the decorator, all classes handle the request.