Unit 3
HTTP and ASP.NET Core

HTTP: Request and Response

- To communicate with a web server, the client, also known as the user agent, makes calls over the network using HTTP. So, when we talk about web applications or web services, we mean that they use HTTP to communicate between a client (often a web browser) and a server.
- A client makes an HTTP request for a resource, such as a page, uniquely identified by a Uniform Resource Locator (URL), and the server sends back an HTTP response.
- ASP.NET Core is a framework for building web applications that serve data from a server. One of the most common scenarios for web developers is building a web app that you can view in a web browser

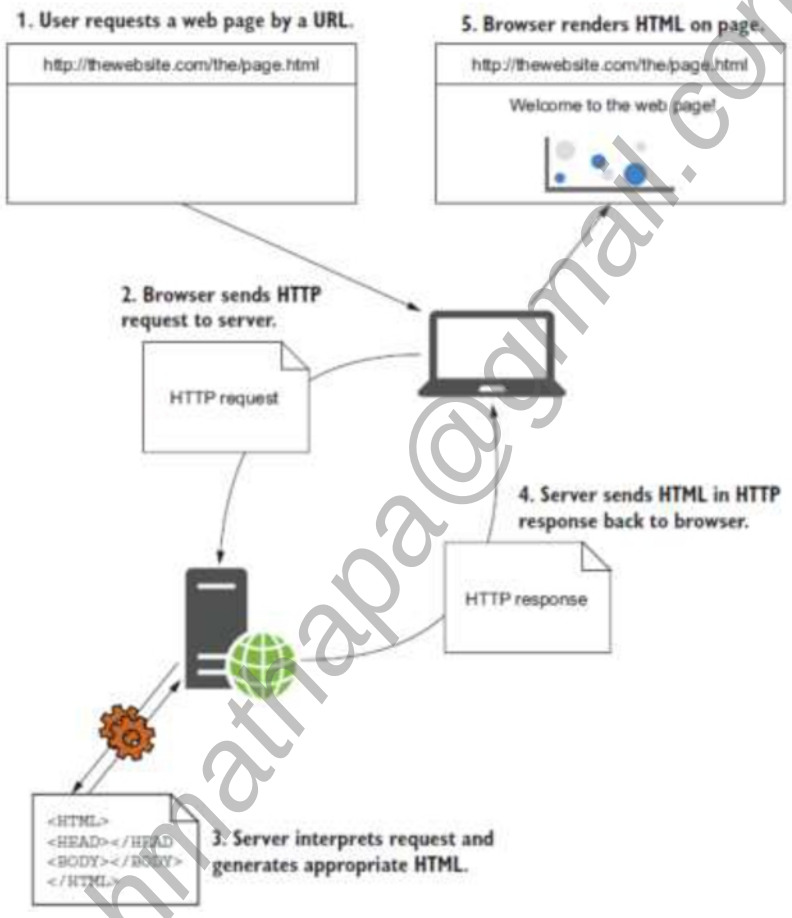
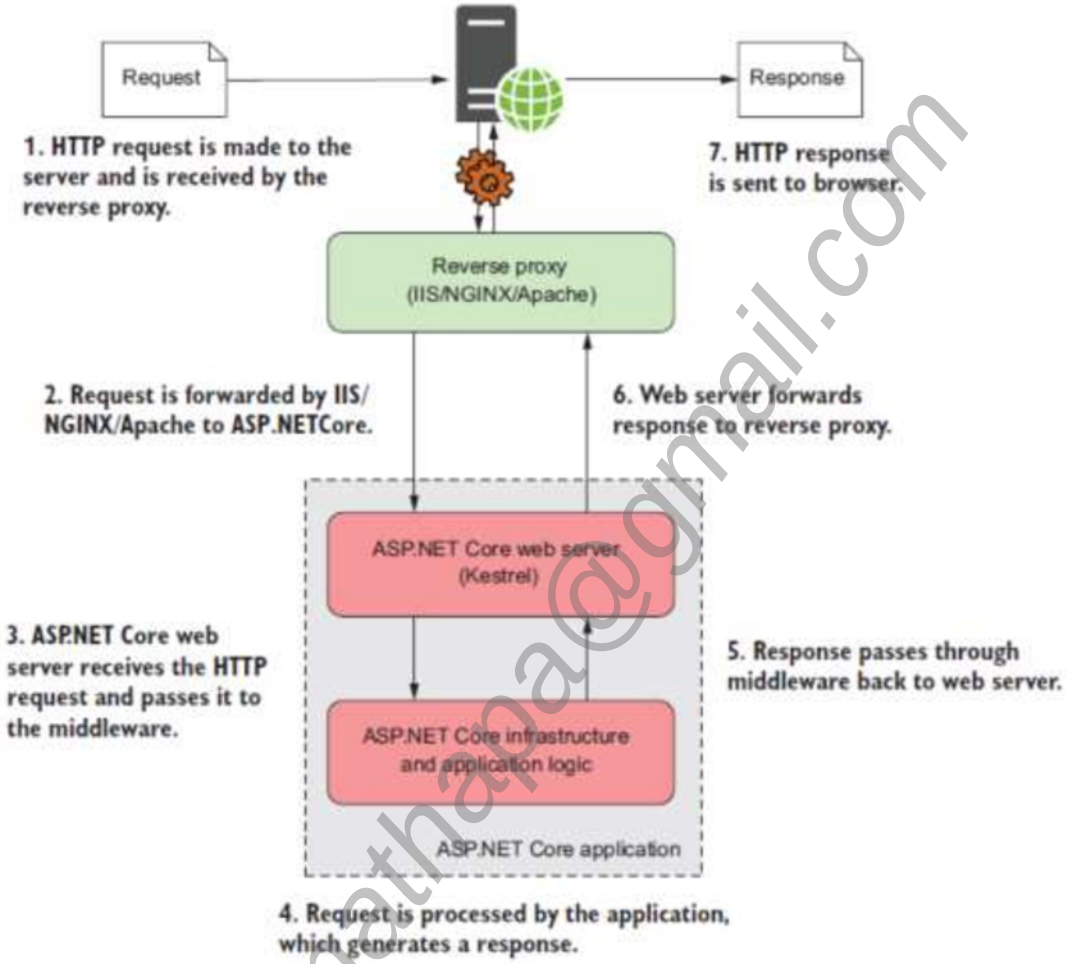


Fig: HTTP Request and Response for Webpage

- The high-level process we can expect from any web server is shown in figure above.
- The process begins when a user navigates to a website or types a URL in their browser. The URL or web address consists of a hostname and a path to some resource on the web app. Navigating to the address in your browser sends a request from the user's computer to the server on which the web app is hosted, using the HTTP protocol.
- The request passes through the internet, potentially to the other side of the world, until it finally makes its way to the server associated with the given hostname on which the web app is running.
- Once the server receives the request, it will check that it makes sense, and if it does, will generate an HTTP response. Depending on the request, this response could be a web page, an image, a JavaScript file, or a simple acknowledgment.
- As soon as the user's browser begins receiving the HTTP response, it can start displaying content on the screen, but the HTML page may also reference other pages and links on the server.
- To display the complete web page, instead of a static, colorless, raw HTML file, the browser must repeat the request process, fetching every referenced file.
- HTML, images, CSS for styling, and JavaScript files for extra behavior are all fetched using the exact same HTTP request process.

Processing a HTTP Request by ASP.NET Core

- When we build a web application with ASP.NET Core, browsers will still be using the same HTTP protocol to communicate with our application.
- ASP.NET Core itself encompasses everything that takes place on the server to handle a request, including verifying the request is valid, handling login details, and generating HTML.
- Just as with the generic web page example, the request process starts when a user's browser sends an HTTP request to the server, as shown in figure below.



- To the user, this process appears to be the same as for the generic HTTP request i.e. the user sent an HTTP request and received an HTTP response. All the differences are server-side, within our application.
- A request is received from a browser at the reverse proxy, which passes the request to the ASP.NET Core application, which runs a self-hosted web server also known as kestrel web server.
- The web server processes the request and passes it to the body of the application, which generates a response and returns it to the web server.
- The web server relays this to the reverse proxy, which sends the response to the browser.
- A reverse proxy is software responsible for receiving requests and forwarding them to the appropriate web server. The reverse proxy is exposed directly to the internet, whereas the underlying web server is exposed only to the proxy. This setup has several benefits, primarily security and performance for the web servers.
- A reverse-proxy server captures the request, before passing it to your application. In Windows, the reverse-proxy server will typically be IIS, and on Linux or macOS it might be NGINX or Apache. The same application can run behind various reverse proxies without modification.
- Implementing two web servers i.e. Reverse Proxy/External server and Core/Internal Web Server, while requesting in ASP.NET core application provides various advantage.
  1. The first one is the decoupling of your application from the underlying operating system. The same ASP.NET Core web server/Internal Web Server, Kestrel, can be cross-platform and used behind a variety of proxies without putting any constraints on a particular implementation. Alternatively, if you wrote a new ASP.NET Core web server, you could use that in place of Kestrel without needing to change anything else about your application.

2. Another benefit of a reverse proxy is that it can be hardened against potential threats from the public internet. They're often responsible for additional aspects, such as restarting a process that has crashed. Kestrel can stay as a simple HTTP server without having to worry about these extra features when it's used behind a reverse proxy. Think of it as a simple separation of concerns: Kestrel is concerned with generating HTTP responses; a reverse proxy is concerned with handling the connection to the internet, securities, and many others.

- Using a reverse proxy has a number of benefits:
  1. Security—Reverse proxies are specifically designed to be exposed to malicious internet traffic, so they're typically well-tested and battle-hardened.
  2. Performance—You can configure reverse proxies to provide performance improvements by aggressively caching responses to requests.
  3. Process management—An unfortunate reality is that apps sometimes crash. Some reverse proxies can act as a monitor/scheduler to ensure that if an app crashes, the proxy can automatically restart it.
  4. Support for multiple apps—It's common to have multiple apps running on a single server. Using a reverse proxy makes it easier to support this scenario by using the host name of a request to decide which app should receive the request.

.NET Core

- **.NET Core** is a **cross-platform** implementation, most versatile framework for building and running all kinds of software's including websites, services, desktop apps and console apps, Mobile Apps, Web APIs, Games etc. on Windows, Linux, and macOS.
- .NET Core is a cross-platform version of .NET for building websites, services, and console apps.
- It is open source on GitHub and accepts contribution from community.
- .NET Core is not limited to a single programming language and supports C#, VB.NET, F#, XAML, and TypeScript.

.NET Core Characteristics

1. NET Core is Free and Open Source.
   The .NET Core platform is free and open source. .NET Core source code project is available on Github. Any developer can get involved in .NET Core development. There are thousands of active developers participating in .NET Core development who are improving features, adding new features, and fixing bugs and issues.
2. .NET Core is Cross-platform.
   .NET Core supports and runs on Windows, macOS, and Linux operating systems.
3. .NET Core provides consistent programing model.
   .NET Core uses one consistent API model written in .NET Standard that is common to all .NET applications. The same API or library can be used with multiple platforms in multiple languages.
4. .NET Core is Modern.
   Unlike some older frameworks, .NET Core is designed to solve today's modern needs, including being mobile friendly, build once run it everywhere, scalable, and high performance. .NET Core is designed to build applications that target all kind of devices, including IoTs and gaming consoles.
5. .NET Core is Fast.
   .NET Core 3.0 is fast. Compared to the .NET Framework and .NET Core 2.2 and previous versions, .NET Core 3.0 is blazing fast. .NET Core is much faster than other server-side frameworks such as Java Servlet and Node.js.
6. .NET Core is Friendly.
   .NET Core is compatible with .NET Framework, Xamarin, and Mono, via .NET Standard. .NET Core also supports working with various popular Web frameworks and libraries such as React, Angular, and JavaScript. TypeScript is one of the key components of the .NET Core and Visual Studio ecosystem.
7. .NET Core Support CLI Tools
   .NET Core includes CLI tools for development and continuous integration
8. .NET Core supports Modular architecture
   .NET Core is released through NuGet Package manager in smaller assembly packages. .NET Framework is one large assembly (System.web) that contains most of the core functionalities. .NET Core is made available as smaller feature-centric packages. This modular approach enables the developers to optimize their app by including just those NuGet packages which they need in their app.

ASP.NET Core

- In November 2015, Microsoft released the 5.0 version of ASP.NET which get separated later and known as ASP.NET Core.
- Also, it is considered as an important redesign of ASP.NET with the feature of open-source and cross-platform.
- Before this version, ASP.NET is only considered as Windows-only version.
- ASP.NET Core was released in 2016.
- ASP.NET Core merges ASP.NET MVC, ASP.NET Web API, and ASP.NET Web Pages into one application framework.
- ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps.
- In simple terms, ASP.NET Core is a significant redesign of the framework ASP.NET. And with benefits like tighter security, improved performance, less coding, and so on, many enterprises have already adopted the new technology to build their applications.
- With ASP.NET Core, we can:
  - Build web apps and services, Internet of Things (IoT) apps, and mobile backends.
  - Use your favourite development tools on Windows, macOS, and Linux.
  - Deploy to the cloud or on-premises.
  - Run on .NET Core.

Benefits/Features of ASP.NET Core

1. ASP.NET Core is free
   ASP.NET Core is an open-source platform which is available free of cost. You can get some very good tools for free or you can use any other open-source IDE.
2. **Performance**
   It is fast and reliable performance. Dot NET Core is extremely fast, definitely faster than any other frameworks like Node.js, PHP or most of the java frameworks. **ASP.NET core allows us to include packages that we need for our application thereby reducing the request pipeline and improving the performance and scalability.**
3. **Modularity**
   The modularity is achieved as everything on the ASP.NET Core is decoupled from the underlying .NET platform. The web framework (ASP.NET Core) is implemented as modular NuGet packages. So, you can customize, update to whatever you want.
4. **Integrations with client-side frameworks**
   Client-side frameworks these days are getting way popular than the enterprise software. ASP.NET Core integrates with popular client-side frameworks and libraries, including Angular, React, and Bootstrap
5. **Cross Platform**
   Ability to develop and run on Windows, macOS, and Linux.
6. **IoC Container: ASP.NET Core includes the built in IoC container for automatic dependency injection.**
7. **Contributions from Open Source**
   Most of the performance improvements were made to Kestrel HTTP Server. Many of the contributions to performance improvements are from contributors to open source project on GitHub. So, developing on Open source means we should certainly see fixes and features would be way faster to production environments.
8. **Productivity**
   With libraries like WebApi, SignalR, EntityFramework, and language features like Linq, you can get such high productivity, those other platform developers can only dream of. We can take advantage of it to increase the productivity.
9. **Supports Cloud-Based Development**
   It is always advised to develop cloud-based applications. ASP.Net Core provides great support for cloud-based development of applications. Whether it is a large enterprise or small, Asp.net core offers the development of several types of web applications, Mobile backend, IoT apps, etc.
10. **Kestrel web server**
    Kestrel is a lightweight HTTP server that handles ASP.NET Core applications request pipeline.

With ASP.NET core, there is a Kestrel web server within the asp.net core application which handles the request. So, the IIS will hand off the request to the Kestrel web server and kestrel processes the request. Instead of IIS, for Mac users, they can have Nginx/Apache as a server to handle the requests and send it to kestrel.

Relationship between ASP.NET Core, ASP.NET, .NET Core and .NET Framework

- The development of ASP.NET Core was motivated by the desire to create a web framework with four main goals:
  1. To be run and developed cross-platform
  2. To have a modular architecture for easier maintenance
  3. To be developed completely as open source software
  4. To be applicable to current trends in web development, such as client-side applications and deploying to cloud environments
- ASP.NET development had always been focused, and dependent, on the Windows-only .NET Framework.
- For ASP.NET Core, Microsoft created a lightweight platform that runs on Windows, Linux, and mac OS called .NET Core as shown in figure below.

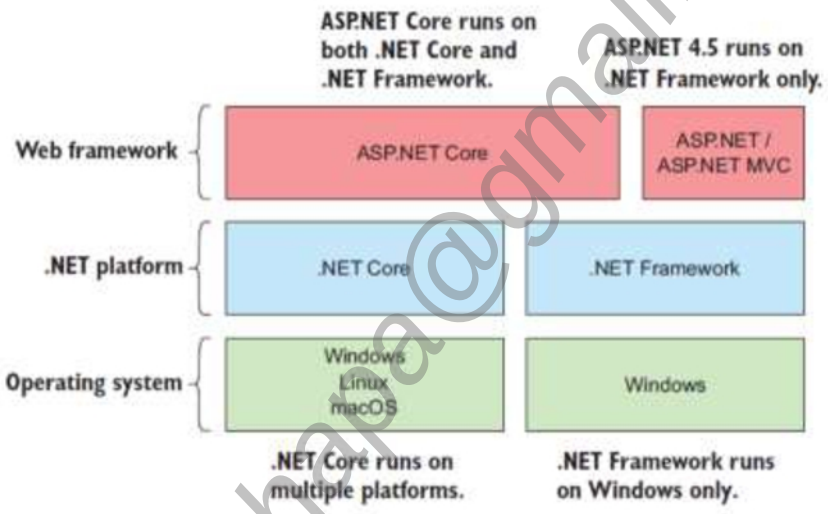


Figure 1.2 The relationship between ASP.NET Core, ASP.NET, .NET Core, and .NET Framework. ASP.NET Core runs on both .NET Framework and .NET Core, so it can run cross-platform. Conversely, ASP.NET runs on .NET Framework only, so is tied to the Windows OS.

- .NET Core shares many of the same APIs as .NET Framework, but it's smaller and currently only implements a subset of the features .NET Framework provides, with the goal of providing a simpler implementation and programming model.
- It's a completely new platform, rather than a fork of .NET Framework, though it uses similar code for many of its APIs.
- With .NET Core alone, it's possible to build console applications that run cross platform.
- Microsoft created ASP.NET Core to be an additional layer on top of console applications, such that converting to a web application involves adding and composing libraries, as shown in figure below.

You write a .NET Core console app that starts up an instance of an ASP.NET Core web server.

Microsoft provides, by default, a cross-platform web server called Kestrel.

Your web application logic is run by Kestrel. You'll use various libraries to enable features such as logging and HTML generation as required.

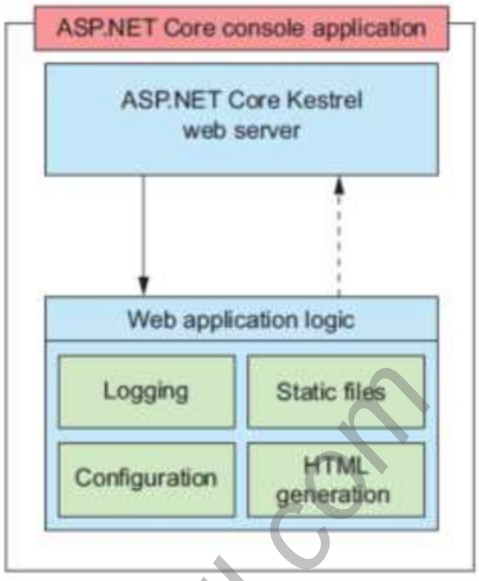


Fig: The ASP.NET Core Application Model

- By adding an ASP.NET Core web server to your .NET Core app, your application can run as a web application.
- ASP.NET Core is composed of many small libraries that you can choose from to provide your application with different features.
- You'll rarely need all the libraries available to you and you only add what you need.
- Some of the libraries are common and will appear in virtually every application you create, such as the ones for reading configuration files or performing logging.
- Other libraries build on top of these base capabilities to provide application-specific functionality, such as third-party logging-in via Facebook or Google.

Common Web Application Architectures: Monolithic and Micro Service Architecture

- In the simplest deployment model, most traditional .NET applications are deployed as **single units** running within a single IIS application domain. Such applications are also called as **monolithic application.** However more complex business applications can be benefited by **adding several layers** to make from some logical separation.
- The monolithic architecture is considered to be a traditional way of building applications.
- A monolithic application is built as a single and indivisible unit. Usually, such a solution comprises a client-side user interface, a server side-application (Business logic), and a database. It is unified and all the functions are managed and served in one place.
- Normally, monolithic applications have one large code base and lack modularity.
- If developers want to update or change something, they access the same code base. So, they make changes in the whole stack at once.
- Also, if such an application needs to scale horizontally, typically the entire application is duplicated across multiple servers or virtual machines.

Note: Horizontal scaling means that you scale by adding more machines into your pool of resources whereas Vertical scaling means that you scale by adding more power (CPU, RAM) to an existing machine.

- While a monolithic application is a single unified unit, a **micro services architecture** breaks it down into a collection of smaller independent units. These units carry out every application process as a separate service. So all the services have their own logic and the database as well as perform the specific functions.
- Micro service architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

- Within a micro services architecture, the entire functionality is split up into independently deployable modules which communicate with each other through defined methods called APIs (Application Programming Interfaces). Each service covers its own scope and can be updated, deployed, and scaled independently.
- Micro services are an accelerating trend these days. Indeed, micro services approach offers tangible benefits including an increase in scalability, flexibility, agility, and other significant advantages.

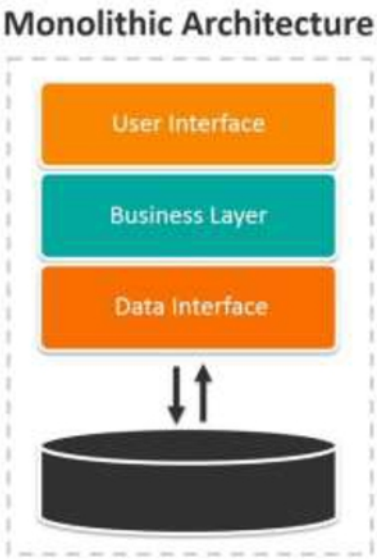


Fig: Traditional Monolithic Application

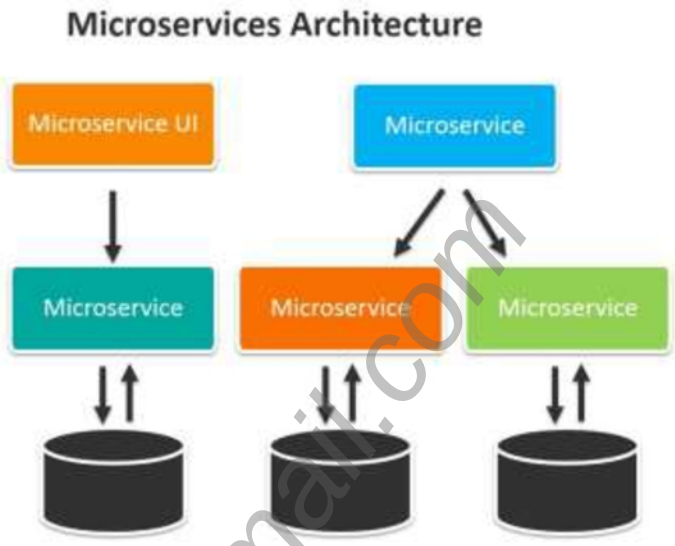


Fig: Modern Micro Services Based Application

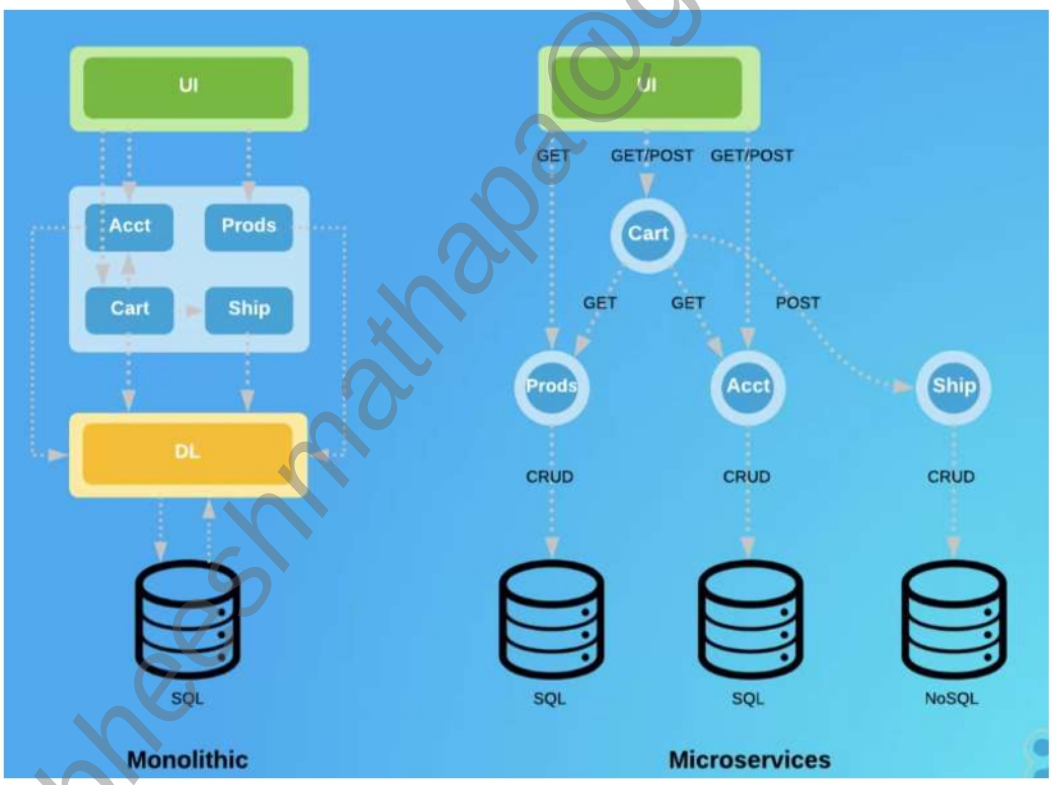


Fig: Example of Monolithic vs micro services based application

Advantage of the Monolithic Architecture

- Less cross-cutting concerns. Cross-cutting concerns are the concerns that affect the whole application such as logging, handling, caching, and performance monitoring. In a monolithic application, this area of functionality concerns only one application so it is easier to handle it. Cross-cutting concerns are parts of a program that rely on or must affect many other parts of the system. **For example:** logging, security and data transfer are the concerns which are needed in almost every module of an application, hence they are cross-cutting concerns.

- Easier debugging and testing. In contrast to the microservices architecture, monolithic applications are much easier to debug and test. Since a monolithic app is a single indivisible unit, you can run end-to-end testing much faster.
- Simple to deploy. Another advantage associated with the simplicity of monolithic apps is easier deployment. When it comes to monolithic applications, you do not have to handle many deployments – just one file or directory.
- Simple to develop. As long as the monolithic approach is a standard way of building applications, any engineering team has the right knowledge and capabilities to develop a monolithic application.

Weaknesses of the Monolithic Architecture

- Difficulty in Understanding. When a monolithic application scales up, it becomes too complicated to understand. Also, a complex system of code within one application is hard to manage.
- Difficulty in Making changes. It is harder to implement changes in such a large and complex application with highly tight coupling. Any code change affects the whole system so it has to be thoroughly coordinated. This makes the overall development process much longer.
- Scalability Issues. You cannot scale components independently, only the whole application.
- New technology barriers. It is extremely problematic to apply a new technology in a monolithic application because then the entire application has to be rewritten.

Advantage of the Microservice Architecture

- Independent components. Firstly, all the services can be deployed and updated independently, which gives more flexibility. Secondly, a bug in one microservice has an impact only on a particular service and does not influence the entire application. Also, it is much easier to add new features to a microservice application than a monolithic one.
- Easier understanding. Split up into smaller and simpler components, a microservice application is easier to understand and manage. You just concentrate on a specific service that is related to a business goal you have.
- Better scalability. Another advantage of the microservices approach is that each element can be scaled independently. So the entire process is more cost- and time-effective than with monoliths when the whole application has to be scaled even if there is no need in it. In addition, every monolith has limits in terms of scalability, so the more users you acquire, the more problems you have with your monolith. Therefore, many companies, end up rebuilding their monolithic architectures.
- Flexibility in choosing the technology. The engineering teams are not limited by the technology chosen from the start. They are free to apply various technologies and frameworks for each microservice.
- The higher level of agility. Any fault in a microservices application affects only a particular service and not the whole solution. So all the changes and experiments are implemented with lower risks and fewer errors.

Disadvantage of the Microservice Architecture

- Extra complexity. Since a microservices architecture is a distributed system, you have to choose and set up the connections between all the modules and databases. Also, as long as such an application includes independent services, all of them have to be deployed independently.
- System distribution. A microservices architecture is a complex system of multiple modules and databases so all the connections have to be handled carefully.
- Cross-cutting concerns. When creating a microservices application, you will have to deal with a number of cross-cutting concerns. They include externalized configuration, logging, metrics, health checks, and others.
- Testing. A multitude of independently deployable components makes testing a microservices-based solution much harder.

Layers/Tiers in Monolithic Architecture

- Layers represent logical separation within the application.
- As applications grow in complexity, one way to manage that complexity is to break up the application according to its responsibilities or concerns.
- This approach follows the separation of concerns principle and can help keep a growing codebase organized so that developers can easily find where certain functionality is implemented.
- Layered architecture offers a number of advantages beyond just code organization, though.
- By organizing code into layers, common low-level functionality can be reused throughout the application.

- This reuse is beneficial because it means less code needs to be written and because it can allow the application to standardize on a single implementation, following the don't repeat yourself (DRY) principle.
- However, when the application logic is **physically distributed** to separate servers or processes, these separate physical deployment targets are referred to as ***tiers*****.**
- The most common organization of application logic into layers is shown in Figure below

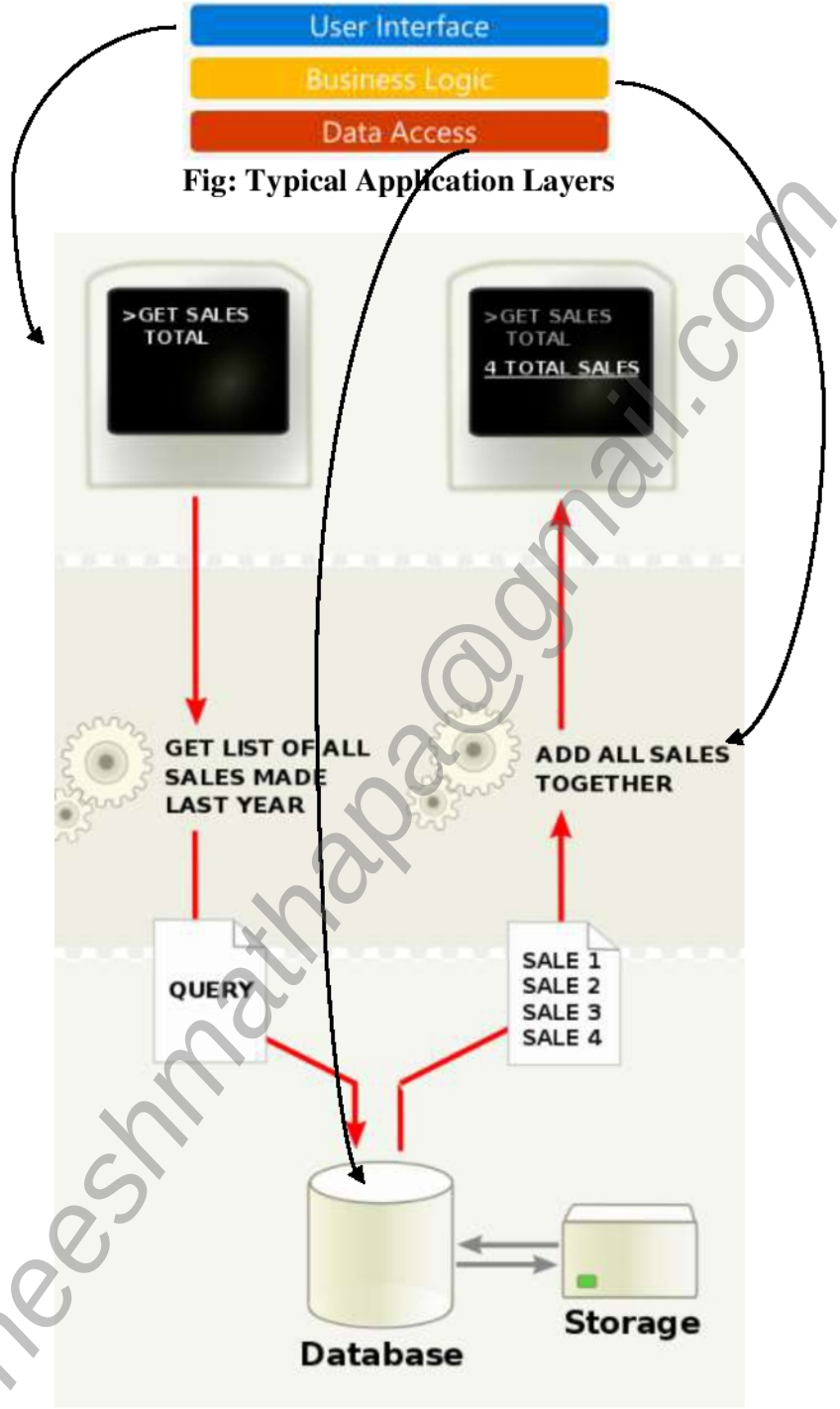


Fig: Typical Application Layers

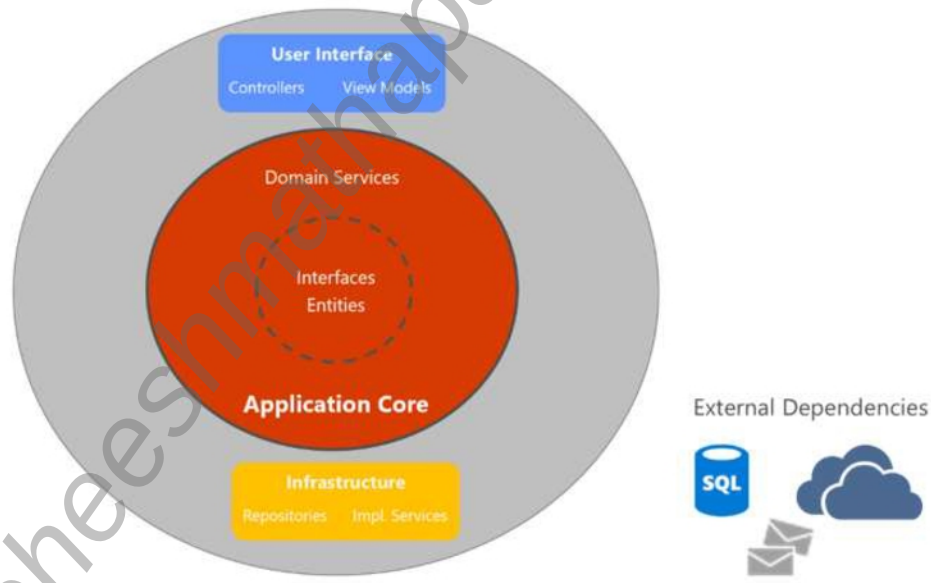
- These layers are frequently abbreviated as UI, BLL (Business Logic Layer), and DAL (Data Access Layer).
- Using this architecture, users make requests through the UI layer, which interacts only with the BLL. The BLL, in turn, can call the DAL for data access requests.
- The UI layer shouldn't make any requests to the DAL directly, nor should it interact with persistence directly through other means. Likewise, the BLL should only interact with persistence by going through the DAL. In this way, each layer has its own well-known responsibility.
- The topmost level of the application is the user interface the main function of the interface is to translate tasks and results to something the user can understand.
- Business logic or domain logic is the part of the program that encodes the real-world business rules that determine how data can be created, stored, and changed. This layers coordinates the application, processes

commands, makes logical decisions and performs calculations. It also moves and processed data between the UI and data access layer.

- The data tier is responsible for storing and retrieving the information from the database or file system. The information is then passed back to the logic tier for processing and then eventually back to the user.
- One disadvantage of this traditional layering approach is that compile-time dependencies run from the top to the bottom. That is, the UI layer depends on the BLL, which depends on the DAL. This means that the BLL, which usually holds the most important logic in the application, is dependent on data access implementation details (and often on the existence of a database). Testing business logic in such an architecture is often difficult, requiring a test database. The dependency inversion principle can be used to address this issue.

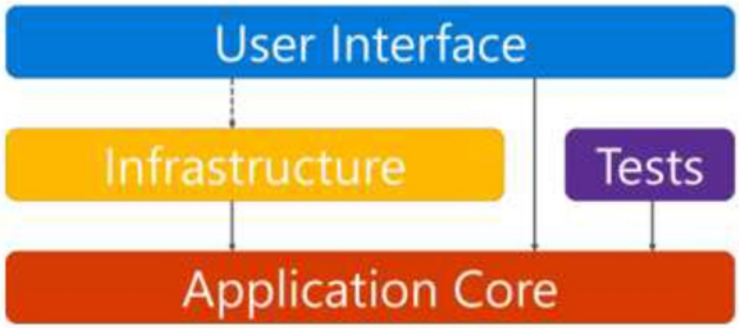
Clean Architecture

- Applications that follow the **Dependency Inversion Principle** as well as **the Domain-Driven Design (DDD)** principles tend to arrive at an architecture referred to as Clean Architecture.
- Domain-driven design (DDD) is the concept that the structure and language of software code (class names, class methods, class variables) should match the business domain. For example, if a software processes loan applications, it might have classes such as LoanApplication and Customer, and methods such as AcceptOffer and Withdraw.
- The Dependency Inversion Principle (DIP) states that **high level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions/interfaces**. *(Will be discussed in detail in coming chapter)*
- Clean architecture puts the business logic and application model at the center of the application.
- Instead of having business logic depend on data access or other infrastructure concerns, this dependency is inverted: infrastructure and implementation details depend on the Application Core.
- This functionality is achieved by defining abstractions, or interfaces, in the Application Core, which are then implemented by types defined in the Infrastructure layer.
- A common way of visualizing this architecture is to use a series of concentric circles, similar to an onion.
- Figure below shows the clean architecture layers/representation.



- In this diagram, dependencies flow toward the innermost circle. The Application Core takes its name from its position at the core of this diagram. And you can see on the diagram that the Application Core has no dependencies on other application layers. The application's entities and interfaces are at the very center. Just outside, but still in the Application Core, are domain services, which typically implement interfaces defined in the inner circle. Outside of the Application Core, both the UI and the Infrastructure layers (example such as data access layer, file system access layer, network access layer, etc.) depend on the Application Core, but not on one another (necessarily).
- Shifting from traditional, data-centric N-Tier architecture to a more domain-centric N-Tier architecture and potentially to the full application of Domain-Driven Design can yield great maintainability benefits for projects. The end result is a system that is loosely coupled, modular, and easily tested.

Clean Architecture Layers



- In the above figure, the solid arrows represent compile-time dependencies, while the dashed arrow represents a runtime-only dependency.
- With the clean architecture, the UI layer works with interfaces defined in the Application Core at compile time, and ideally shouldn't know about the implementation types defined in the Infrastructure layer.
- At run time, however, these implementation types are required for the app to execute, so they need to be present and wired up to the Application Core interfaces via dependency injection.
- Because the Application Core doesn't depend on Infrastructure, it's very easy to write automated unit tests and perform integration testing for such layer.
- In a Clean Architecture solution, each project has clear responsibilities. As such, certain types belong in each project and you'll frequently find folders corresponding to these types in the appropriate project.

Application Core

- The Application Core holds the **business model,** which includes **entities (Model Class), services, and interfaces.**
- These interfaces include abstractions for operations that will be performed using Infrastructure, such as data access, file system access, network calls, Logger service, Email service etc.

Infrastructure

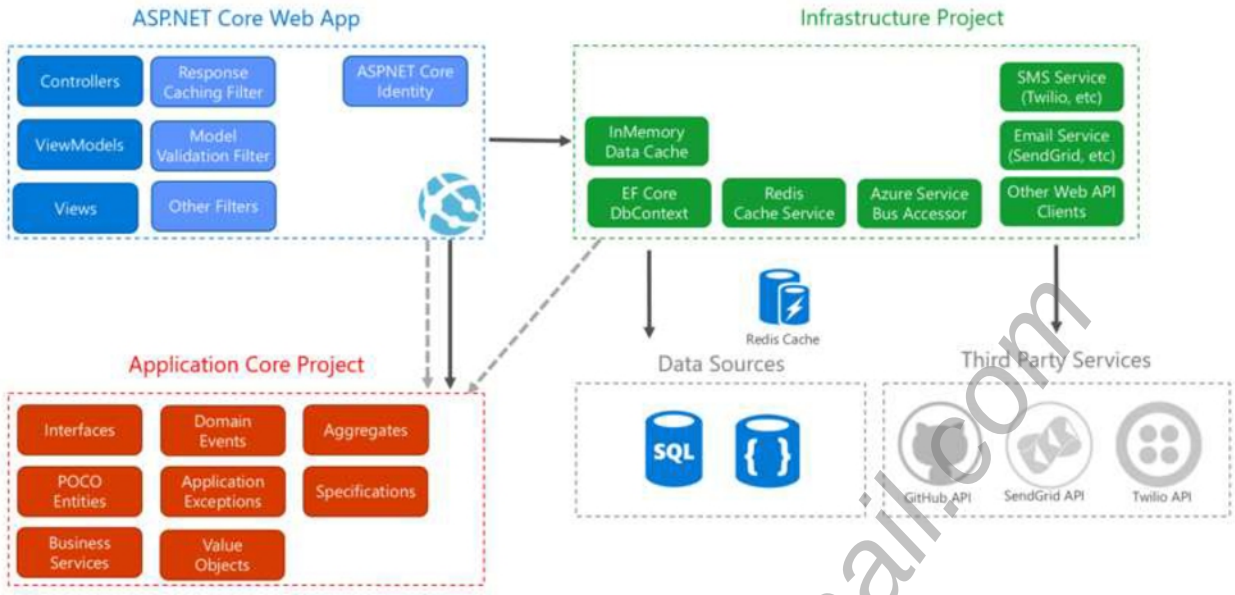
- The Infrastructure project typically includes data access implementations.
- In a typical ASP.NET Core web application, these implementations include the Entity Framework (EF) DbContext, any EF Core Migration objects that have been defined, and data access implementation classes (Repositories which is implemented via Repository Design Pattern), infrastructure specific services like Logger, Emailer etc.
- In addition to data access implementations, the Infrastructure project should contain implementations of services that must interact with infrastructure concerns. These services should implement interfaces defined in the Application Core.

UI Layer

- The user interface layer in an ASP.NET Core MVC application is the entry point for the application.
- This project should reference the Application Core project, and its types should interact with infrastructure strictly through interfaces defined in Application Core.
- No direct instantiation of or static calls to the Infrastructure layer types should be allowed in the UI layer.
- Since the UI layer doesn't have any direct dependency on types defined in the Infrastructure project, it's likewise very easy to swap out implementations, either to facilitate testing or in response to changing application requirements.
- UI Layer includes
  - Controllers
  - Views
  - ViewModels
  - Startup etc….
- The Startup class is responsible for configuring the application, and for wiring up implementation types to interfaces, allowing dependency injection to work properly at run time.

ASP.NET Core Application Architecture

- Below figure shows a more detailed view of an ASP.NET Core application's architecture when built following **Clean Architecture Recommendations.** i.e. UI, infrastructure and application core



- Gone are those days when the development of website consumed a lot of time and experiences as everything needed to be done from scratch by yourself. Now a day, everything is available to a web developer and all that is needed to build a website is understanding of several technologies, frameworks, design patters etc. which primarily focus on rapid application development with fewer effort, more user friendly and interactive.
- ASP.NET Core is a significant redesign of the framework ASP.NET. And with benefits like tighter security, improved performance, less coding, more user friendly and so on, many enterprises have already adopted this new technology to build their applications.
- The ideology behind ASP.NET core is to lay out web logic, infrastructures and the core components independently form each other in order to provide a more development friendly environment.
- For this ASP.NET core defines these layers as the core components of the platform which relieves the developer from redefining it in order to facilitate application rapid development, make solution more modular and reusable.
- In ASP.NET core application, business logic and UI logic are encapsulated in ASP.NET Core web application layer, while the database access layer, cache service and WEB API's services are encapsulated in infrastructure layer and similarly common utilities, interfaces, and other business services are encapsulated as micro-services in application core layer.
- ASP.NET Core creates necessary predefined N layers' architecture for us developers automatically, which saves our time and effort to worry less about the complexity of necessary N Layers architectures and focus on business logic. We are more focused on managing the complexity of the project in terms of code reusability and modular component architectures to make our product more robust and modular and scalable. In addition, ASP.NET Core's built-in use of and support for dependency injection makes this architecture the most appropriate way to structure non-trivial monolithic applications.