**Web API Core Theory:**

**Reference URLs:**

[An Introduction to .NET Core (c-sharpcorner.com)](https://www.c-sharpcorner.com/article/an-introduction-to-net-core/)

[.NET Core - Top 10 Advantages for App Development | Fortech](https://www.fortech.ro/top-advantages-net-core/)

<https://www.c-sharpcorner.com/article/asp-net-core-5-0-web-api/>

<https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting?view=aspnetcore-3.1>

<https://www.c-sharpcorner.com/article/routing-in-restful-apis-using-net-core/>

<https://www.c-sharpcorner.com/article/create-api-with-asp-net-core-day-three-working-with-http-status-codes-in-asp/>

<https://www.c-sharpcorner.com/article/frequently-used-status-code-and-how-to-return-them-from-asp-net-core-web-api/>

<https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting?view=aspnetcore-3.1>

<https://medium.com/cheranga/calling-web-apis-using-typed-httpclients-net-core-20d3d5ce980>

<https://www.c-sharpcorner.com/article/enabling-cors-in-asp-net-core-api-application/>

https://ochzhen.com/blog/created-createdataction-createdatroute-methods-explained-aspnet-core#:~:text=The%20main%20difference%20is%20how,and%20CreatedAtRoute%20gives%20more%20safety.

**For Swagger:**

<https://www.c-sharpcorner.com/article/asp-net-core-3-1-web-api-and-swagger/#:~:text=For%20adding%20Swagger%20or%20OpenAPI,Swagger%20to%20the%20DI%20container>.

<https://www.c-sharpcorner.com/article/how-to-implement-swagger-in-asp-net-core-web-api/>

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-6.0>

**For CORS:**

<https://docs.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-6.0>

<https://www.c-sharpcorner.com/article/enabling-cors-in-asp-net-core-api-application/>

**Reference URLs for Demo:**

<https://www.c-sharpcorner.com/article/crud-operation-in-asp-net-core-5-web-api/>

<https://www.codeproject.com/Articles/1259484/CRUD-Operation-in-ASP-NET-Core-Web-API-with-Entity>

<https://docs.microsoft.com/en-us/ef/core/modeling/relationships?tabs=fluent-api%2Cfluent-api-simple-key%2Csimple-key>

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-javascript?view=aspnetcore-3.1>

<https://www.c-sharpcorner.com/article/routing-in-restful-apis-using-net-core/#:~:text=NET%20Core%20APIs%2C%20it%20comes,template%20applies%20to%20each%20action.&text=When%20you%20create%20a%20WEB%20API%20with%20>.

**Exception Handling and Custom Middleware:** <https://codingsonata.com/exception-handling-and-logging-in-asp-net-core-web-api/>

**.NET Core:**

Since the initial version of .NET Framework (the first beta version of .NET 1.0 was released in the fall of 2000), Microsoft has released many more upgrades over the years. They did so with the intent to make the framework cross-platform, open-source, and modular. As such, the software giant has now come up with .NET Core Framework.

Designed for adaptability, Microsoft's .NET Core is an open-source, cross-platform, modular platform that enables you to build and deploy applications that are targeted at multiple operating systems. It should be noted that .NET Core is now available on the Windows, OS X, and Linux platforms. It is supported in several flavors of Linux, including Red Hat Enterprise Linux and Ubuntu. This article presents a discussion on .NET Core, its features, and why it is useful.

**What Is .NET Core?**

.NET Core is a open source and cross-platform .NET stack from Microsoft that is optimized for open-source development and agile delivery on NuGet. .NET Core is designed by Microsoft to support many different platforms and architectures. It should be noted that WPF and WinForms aren't supported on .NET Core. The MSDN states ".NET Core is a cross-platform, open-source, and modular .NET platform for creating modern web apps, microservices, libraries, and console applications."

Why .NET Core?

Because the performance is high ,it can serve upto 6.97 million request per second.

**Main Characteristics of .NET Core**

The major characteristics of .NET Core include the following:

**Cross-platform support**

* It can work on multiple platforms, such as Windows, Mac, and Linux.

**Flexible deployment**

You can deploy a .NET Core application side by side with your application seamlessly. It is a general-purpose development platform that consists of several components. These include the managed compilers, the runtime, and the base class libraries. It also includes many application models, such as the ASP.NET Core.

**Modular**

.NET Core is composed of a set of modular components. This enables you to take advantage of the package you want to use rather than including the entire .NET Core framework. This boosts performance as you end up creating applications that contain just what you need.

**Open-source**

* .NET Core is open source using MIT and Apache 2 licenses and is available in GitHub.

**.NET Core Stack: The Architectural Components**

As we have already discussed, .NET Core is designed in a modular manner. Hence, it is componentized. The components of .NET Core include:

**A .NET runtime**

This is used to provide type safety, garbage collection mechanism, native interop services, loading of assemblies, etc.

**A collection of Framework Libraries**

These provide the primitive data types, the utilities, etc.

**A collection of SDK tools and compilers**

These provide developer experience when working with the .NET Core SDK.

**Comparing and Contrasting .NET Framework and .NET Core**

Now let's take a quick tour of how .NET Framework and .NET Core compare and contrast to each other. Well, both are managed frameworks, and they have their own runtime environments and garbage collection mechanisms. Here’s the list of the differences between these two frameworks.

**Open-source**

While only a subset of .NET Framework is open source, the entire .NET Core framework is open source.

**Subsystems**

.NET Core has adopted a simpler programming model. Some of the features of .NET Framework—like Code Access Security—are not supported by .NET Core.

**Multi-platform support**

Unlike .NET Framework,

* .NET Core provides support for multiple platforms, i.e., Windows, Mac OS, and also Linux.

**Fewer Application Programming Interfaces (APIs) support**

As of this writing, .NET Core doesn’t support all the APIs that are available in .NET Framework.

**Advantages of .NET Core:**

**.NET Core is Cross-Platform**

**.NET Core is Opensource**

**.NET Core is Cost Effective**

**.NET Core Enables Flexibility**

**.NET Core has a Large Community**

**1. .NET Core is Cross-Platform**

When you start creating an application, you go on the road with a few things already settled.

One of them is the Operating System (OS) that your app targets. The decision is driven by the context, the market, the end users or, in some cases, even by your development team.

Because the market and your budget are not linear, constantly evolving and changing, you need to be able to adapt your business fast. If at a given time you need to be able to support more users at the same time, you might find yourself in a position where you should add more resources to servers – which means a bigger investment in infrastructure from your side. You might be able to avoid this by changing the hosting from Windows to Linux.

Before .Net Core, in April 2014, if you started a project in the .NET framework, you already knew that you’ll be stuck with Windows, and all the licensing and infrastructure costs involved.  Deciding to suddenly switch to another platform, would be possible only by starting the app from scratch. This would require a new platform, a new development team, with different skills and of course, will mean a new business investment. In many cases, you won’t able to reuse the code already written, and porting the app will come with its own challenges.

**.NET is a Portable and Runnable Framework**

The .Net framework has been re-invented, and what was once limited to Windows is now officially portable and runnable on just about any platform you need. You have a wide variety of hosting models, you can target macOS, Linux, Windows or even run it with Docker.  You can switch very easily to another platform if your business context requires it. For example, if we are talking about servers to host web apps, you can host them on Linux with Nginx or Apache, as windows services, in IIS (the Microsoft proprietary web server), in process, or out-of-process, with IIS as a reverse proxy, with Kestrel, HTTP.sys, or with Docker.

With .Net Core 2.2, you’ll even be able to create your own server programmatically, with all the power and responsibility that comes with it in terms of security and reliability. It will have only those features you need, giving you the most flexibility you can get on the market by removing any provider facing dependencies (even if it’s only a hosting web server).

According to the latest [StackOverflow survey](https://insights.stackoverflow.com/survey/2018/#technology-most-popular-technologies-all-respondents2),  .Net Core is one of the most used frameworks, being the second server-side development framework after Node.js. And if you ask me, the backend JavaScript hype will soon find its limitation, ceasing the place to more mature technologies, or to niche new ones (like Go).

The fact that .Net Core is a cross-platform framework allowed [VQ Communications](http://customers.microsoft.com/en-us/story/vqcommunications) to have the same code base on two platforms, thus addressing a potential scalability issue. They say this gives the developers more freedom on how they develop the product because it simply runs on the platform they need it to run on.

Having the same codebase for multiple operating systems reduces costs by eliminating the hassle of having to maintain, develop, track, fix and test the same bugs in at least two places.

**2. .NET Core is Open Source**

The Benefits of Open Source Frameworks

Open-source projects don’t suffer from bureaucracy or command-and-control development philosophies, benefiting from fast release cycles, abundant support, and modern tooling. This is due to the distribution of users and the lock-free. This world is as Agile as it gets, incorporating many best practices, real-world uses cases and scenarios, and vivid and modern development workflows.

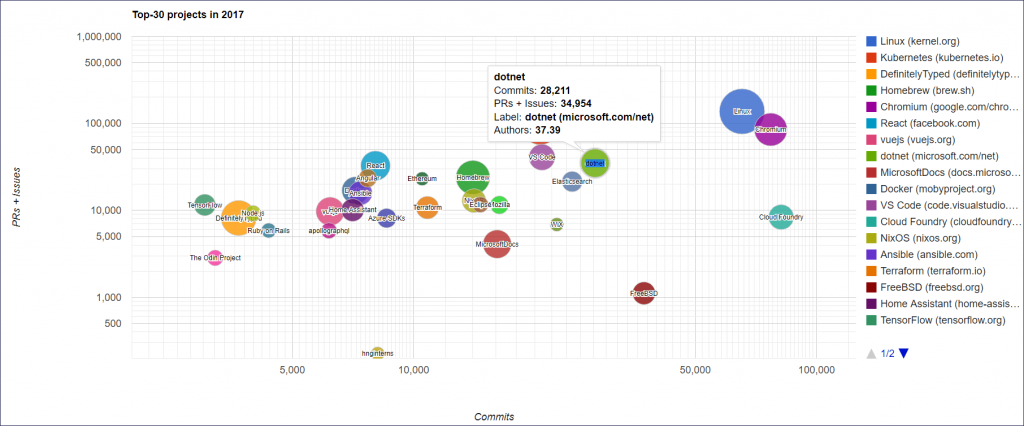
Most of the mainstream issues were already solved, and the developers can stop reinventing the wheel and focus on business cases, shifting to more high-value work and a high customer focus and business domain expertise.

Developing apps by using an open source platform may add longevity as a feature because you’re not locked into a certain vendor, and you can always find someone from the contributors’ pool to keep you running.  Moreover, if the market changes or evolves, or the business requirements change drastically, you might find yourself in a position where the only option to keep you afloat is to start investing in another solution provider, infrastructure, or consultancy hours.

By choosing an open-source framework, you can increase your project’s chances of success by directing the budget to other services like hosting, change management, training or even marketing. Overall, it can decrease the total cost of ownership for the project.

Additionally, by choosing to develop your application with an open source platform, your business will be able to evolve as the market evolves, without needing bigger and bigger investments in those areas that don’t bring an immediate return.

**.NET as an Open Source Framework**

.Net is the highest velocity application framework in open source as stated by [Cloud Native Computing Foundation.](https://www.cncf.io/blog/2017/06/05/30-highest-velocity-open-source-projects/) Due to the fact that many developers use it, the framework is highly optimized as they find and fix bugs, add features and contribute with ideas and implementations.[](https://www.fortech.ro/wp-content/uploads/2018/10/DotNetHighVelocityFramework.png)

In 2017, the dotnet project had over 28 000 commits from the community, being in the top 30 highest velocity open source projects.

The dotnet project is part of the [.Net Foundation](https://www.dotnetfoundation.org/), a non-profit organization that fosters and supports 25.000 contributors, 1700 companies and has more than 55 active projects.

This foundation gathers passionate developers and encourages communities around the world to meet and share knowledge, offering project guidance and mentorship, facilitating open source in the Microsoft ecosystem.

By being so close to the developer’s world, listening to their voices, taking into consideration their proposals, and basically supporting them. Overall open-source means people collaborating for a common goal.

Major players like Samsung, JetBrains and RedHat joined Microsoft in this initiative deciding to guide and shape the future of the .NET framework.

For example, [Samsung announced the collaboration](https://news.samsung.com/global/samsung-joins-the-microsoft-net-community-enabling-c-developers-to-build-applications-for-samsung-tizen-devices) with Microsoft on open source projects for [Tizen](https://developer.samsung.com/tv/tizen-net-tv).  With this, they reached out to a new pool of capable developers with strong technical knowledge and expertise, offering them the opportunity to work on cool and futuristic applications for Smart TVs, smartphones, IoT or wearables.

Since joining the foundation, Samsung is continuously contributing to ARM support for .NET Core, [Xamarin.Forms](https://github.com/xamarin/Xamarin.Forms) and Visual Studio Tools for Tizen.

**3. Mature Framework and Widely Used Programming Languages**

The .Net framework has been around for 18 years. This means that millions of apps, in different domains were developed with it. It proved its flaws and strengths.

According to similartech.com[, 1.17% of worldwide websites](https://www.similartech.com/categories/framework) are made with .Net, being outclassed only by PHP due to the large usage of WordPress.

In 2017, StackOverflow states that C# [is the 3rd most used programming language](https://insights.stackoverflow.com/survey/2017#technology-most-popular-languages-by-occupation) of the year meaning that C# has a lot of support and a large community of users.

*“Developers can reuse their C# skills to build native Android, iOS and Windows applications that deliver the right information to the right person at the right time.”*Alaska Airlines

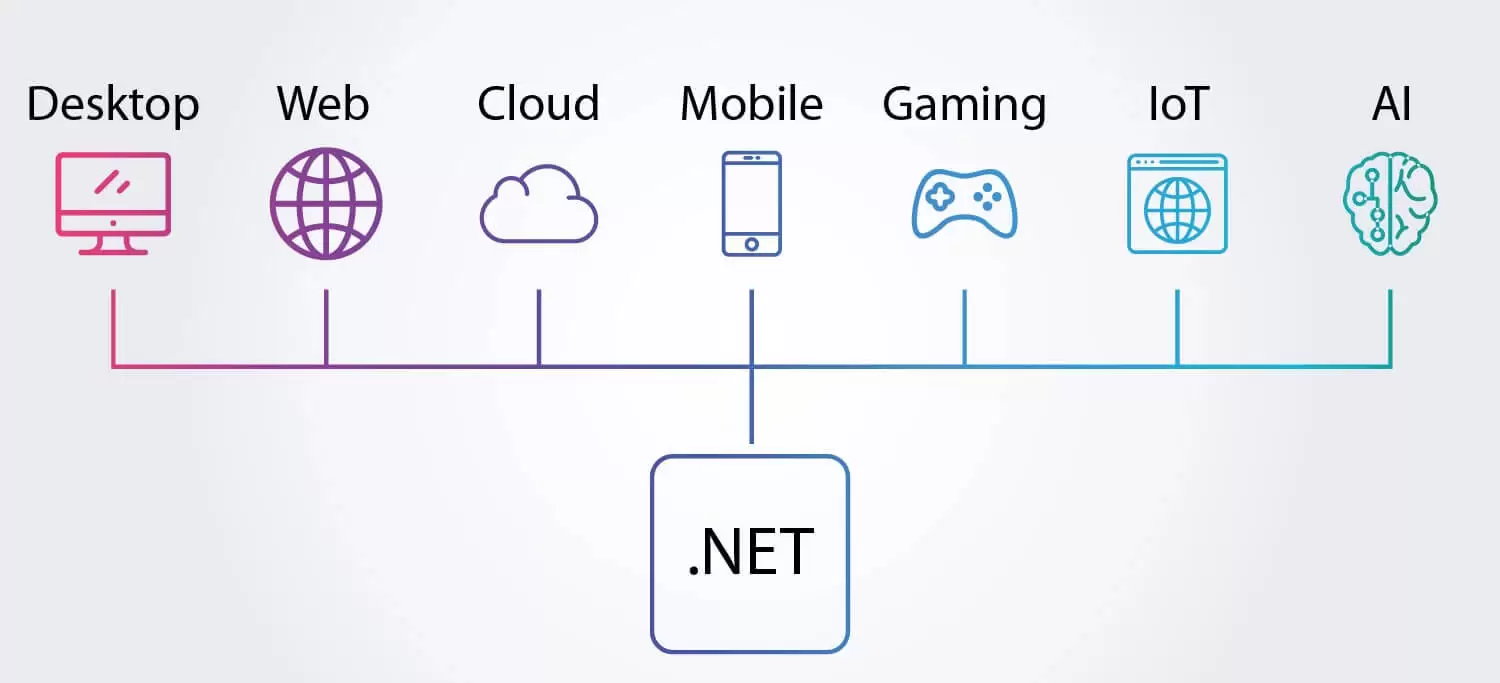
Coding in a language that is stable and always evolving, is a major plus for a developer. Not only it makes the day to day life easier and more enjoyable, but it increases productivity and sometimes prevents unwanted bugs.

C# is a multi-paradigm language, strongly-typed, modern and last but not least, object-oriented with a huge support for asynchronicity.  Its history started in 2002, with the 1.0 version.

Designed to incorporate many of the best features other languages provide, but without the issues those languages had, C# was the first language made for .NET framework.  The syntax is simpler, you don’t need to worry about memory management, and as a developer, if you code in C#, you are half-way to knowing JavaScript syntax.

From a business perspective, a C# developer can easily find his way around JavaScript. It becomes even easier if we add [TypeScript](https://www.typescriptlang.org/) into the picture (an open-source programming language developed and maintained by Microsoft) because it simply offers a familiar coding experience. And let’s be honest, most of the front-end developers use TypeScript, even though they won’t admit it.

**4. .NET Core Supports a Wide Range of Application Types:**



No matter what the business domain of an application is, at a certain point it might be valuable to develop a whole suite of other apps revolving around it, that can reach different markets and end users. With the .Net framework, you have the ability to develop apps in a multitude of domains, such as gaming, mobile, IoT, AI –  you name it, it can be done with .Net.

A few days ago, [Microsoft announced a collaboration with SWIFT](https://blogs.microsoft.com/blog/2018/10/21/how-microsoft-is-putting-data-and-ai-at-the-center-of-financial-services-industry-transformation/?utm_source=li&utm_medium=organic&utm_campaign=cmg_as&utm_term=SIBOS), the world leader in secure financial services. This means putting AI and data in the center of Fintech is changing perspectives from the ground up, by creating an ecosystem that enables faster and more efficient and secure operations for banks and corporations.

Recently more and more businesses saw a huge potential in AI and Machine Learning, Big Data. They started to invest in integrating such components in their applications. Healthcare, banking, education, business, marketing and pretty much everything we use in our daily life could have such components, making our life easier.

When it comes to using free libraries, you can choose from millions of public and private packages on [NuGet](https://www.nuget.org/) and [MyGet](https://www.myget.org/). You can also integrate them in other apps by using the IDE or the CLI.

**5. Increased Security with .NET Core**

Security nowadays it’s getting more and more highlight. Having an application exposed over the web is a major thing and protecting it should be a major concern for any serious business. You wouldn’t leave your child alone at night, outside, in another city, wouldn’t you?

A few years back, when we were talking about security in a web application, HTTPS redirection seemed to be the key, while protection against SQL Injection and CSRF were pretty much all the important things worth mentioning, at least for most developers.

Now, with the different and modern architectures, with REST APIs and microservices, looking at security in the way we used to, it’s not enough anymore. We hear about JWT Tokens, Claim-based authorization, Policy-based authorization, Resource-based authorization, payload protection, ephemeral keys, IP safelisting and many more terms that are hard to understand, and even harder to implement.

If all these aren’t scary, what do you do about security-sensitive data that a web app often needs – like connection strings, credentials, keys – but shouldn’t be stored it in code, or source control, even if encrypted?

Fortunately, .Net offers a wide range of easy to use mechanisms for authentication, authorization, data protection, and attack prevention.

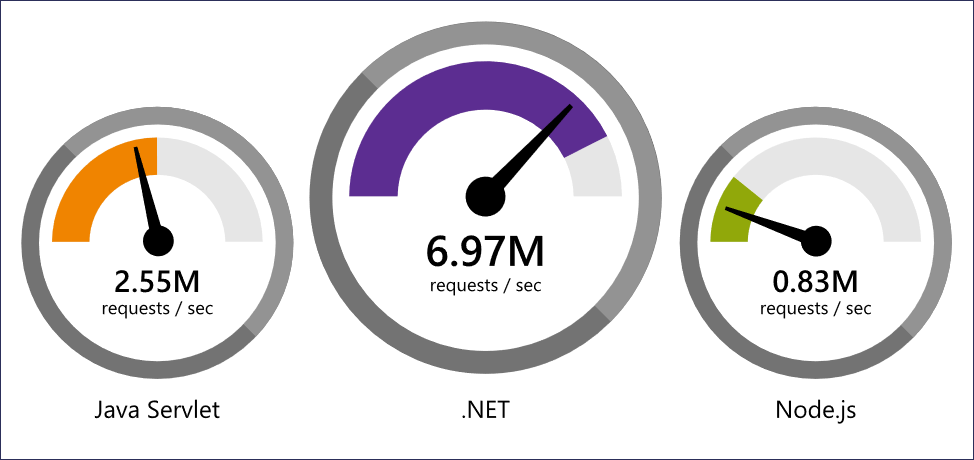
For example, DataProtection API and Secret Manager, along with the Azure Key Vault Provider can be used to protect data, generate keys, rotate keys or generate ephemeral keys. Moreover, you can build on top of these APIs, by extending them to cover the business scenarios you need.

In terms of authentication, ASP.NET Core Identity allows you to integrate your app with all major external providers.

Facebook, Google, WS-Federation, Windows or two-factor authentication might be valid requirements in real-life scenarios and without the right framework, it can be very hard to implement. Thus, having everything ready to be extended is a big win.

**6. .NET Core Enables Top App Performance**

.Net Core was designed to have a pay-for-what-you-use model, where performance is a key feature. With each released [version they add a few percents on performance](https://blogs.msdn.microsoft.com/dotnet/2017/06/07/performance-improvements-in-net-core/). For example, TechEmpower benchmark compares different application frameworks and one of the results looks like this.

[](https://www.fortech.ro/wp-content/uploads/2018/10/NETPerformanceTecEmpower.png)

Data sourced from official tests available at [TechEmpower Round 16.](https://www.techempower.com/benchmarks/#section=data-r16&hw=ph&test=plaintext)

[StackOverflow](http://customers.microsoft.com/en-us/story/stacked-for-speed), the leading knowledge website for programmers, serves 5.3M page views/day with just 9 servers.

Raygun, a company from New Zealand, developed a cloud-based error and crash reporting platform that monitors web and mobile applications.  The core of their app was written in the .Net framework, but they had APIs that received data from customers written in Node.js on Linux, to support elasticity. Over time they noticed the Node.js framework was becoming slower and heavyweight with each version and decided to scale up (to add more resources to the servers), but they realized the costs to support this will rise significantly.

As a consequence, they decided to port their API’s to .NET Core to reuse the core product know-how and ended up increasing their [throughput by 2000%](http://customers.microsoft.com/en-us/story/raygun) using the same-size servers as before.

*“Using the same-size server, we were able to go from 1,000 requests per second per node with Node.js to 20,000 requests per second with .NET Core.”*John-Daniel Trask, Chief Executive Officer and Co-Founder, Raygun

Paired with the right database and architecture, this framework is the perfect solution for any kind of business that might face the need to scale eventually.

**7. .NET Core Enables Flexibility**

Flexibility for a business means the ability to evolve, change visions and goals and adapt easily to different external scenarios.

In terms of code and choosing of the right framework, we are talking about databases and infrastructure mainly.

**Databases** – Being able to switch to another kind of database and reuse the code already written is something impossible to do in some frameworks because the framework comes as a package with the language and preferred tools, the database being a part of it. Changing it can be dictated by the performance needs, licensing costs, or scalability, hosting options or even the business domain.

**Infrastructure** – a new and modern infrastructure can add throughput, minimize downtimes and sustain a reliable business.

*“.NET Core gives us the freedom to take advantage of new infrastructure technologies that run on Linux such as Kubernetes and Docker.”* Jeremiah Gowdy, Principal Software Architect, GoDaddy

**8. .NET Core is Cost Effective**

When choosing the appropriate technology for your application, one of the first things you might want to look at is the IDEs and tools that are either mandatory for that technology or simply make the developers more productive. A lot of them are free, but the for sure, the best ones often come with licensing costs per developer, per year and the total price can sum up to thousands of dollars. Regardless if you are a startup trying to find your way into the market, or a big company already there, that money can be better spent somewhere else, with a higher return, or not spent at all.

A few years ago, Microsoft developed Visual Studio Code, an IDE that can be used for commercial apps. This was so good, small and extensible and updated, that became the preferred IDE even for front-end developers who were using IDEs that need a paid license.

In terms of infrastructure, with .Net Core and the multitude of hosting options, you have the freedom to switch to Linux and to pick a provider of your choosing – that offers you a great SLA or is cheaper. Furthermore, you can choose to host your application in the cloud, by relying on the services of any cloud provider (such as Azure).

In terms of databases or storage, you have unlimited options. You are not forced to use a big, paid solution, so you can easily go to something that is Open-Source – like PostgreSQL.

Even if costs aren’t an issue, not being forced to invest in something that isn’t really necessary gives your business flexibility, and maybe more capacity to invest in the development of your team or even R&D.

**9. .NET Core has a Large Community**

Like it or not, when a framework is used by many developers and businesses, it means that it must have something good to attract them. It also proves its stability and a high level of trust from the community, even more, if it’s not something that is super new, mainstream, that everyone wants to try.

Under the .Net Foundation umbrella, passionate people that work in different domains, with different expertise and know-how, contribute and share ideas, shaping thus the way applications come to life.

**10. .NET Core was Created by Microsoft**

Having such a big corporation behind a framework doesn’t mean that you will need to pay for everything you use from them, tools or languages or IDEs.

It means that you have a guarantee that all the good ideas that come from the developer’s communities will see the light of the day as soon as possible. They will sometimes be bundled with the coolest paradigms out there, and as a result, you’ll get the most modern, flexible and well-maintained framework you can get for an application.

**Conclusion**

When you choose the right technology for your business’s application, you shouldn’t pick a framework that is popular because the context made it that way – there was no other competing option out there at that point, or the cost of ownership was smaller because it was open source.

You should analyze and consider every reason that could add value to your business, like flexibility and cost reduction, performance, infrastructure and all the benefits that come from them.

You will want something that is fast and cross-platform, that is stable and mature while having open-source advantages and can easily be integrated with other libraries, databases, and modern tooling.

By choosing .NET Core, you can benefit from all the advantages above and develop a large variety of applications for different domains, while reusing your existing resources and the developers’ technical expertise and passion.

**Project templates supported by web api : JSON,XML**

**REST: Representational State Transfer**

**HTTP Verbs:**

**GET :** You can retrieve the data over HTTP. Method with **Select** query

**POST :** You can insert the data over HTTP. Method with **Insert** query

**PUT :** Update the data over HTTP. Method with **Update** query

**DELETE :** Remove the data over HTTP. Method with **Delete** query

**Introduction to RESTful Services:**

Represents services as resources which can be accessed via HTTP.

Service created based on REST guidelines (Google Search: RESTful Commandments).

Service should be stateless : Service should not maintain any state within the service. It can maintain the state outside E.g. – In database.

It should be Client Server architecture.

There should be the complete representation of the state of the resource in the client request and service response. There should not be any assumption (Indicator for CRUD operation).

Service should tell the clients whether details are to be cached or not (expiry).

**Introduction to ASP.NET Web API:**

**Agenda:**

1) What is API?

2) What is Web API?

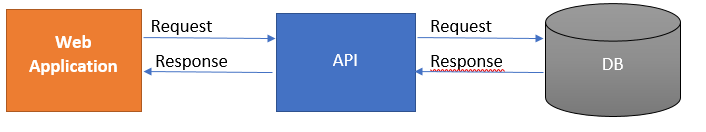
3) Why Web API is required?

4) What is ASP.NET Core Web API?

**What is Web API?**

The first question comes to mind is, "What is API”?

API stands for Application Programming Interface. It is an intermediate software agent that allows two or more applications to interact with each other.



Now the next question is

**"What is a web API?"**

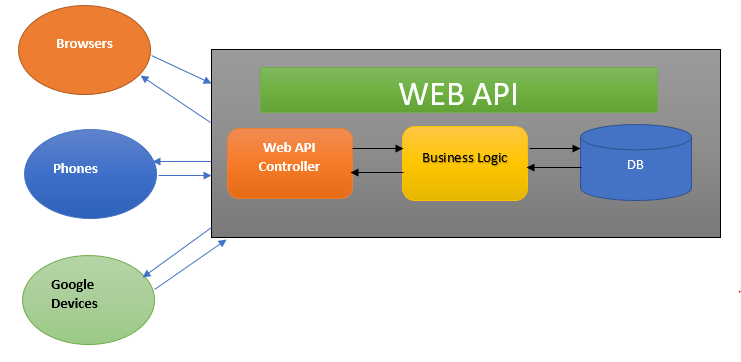
In simple words, we can say that a web API is an application programming interface for a web application or web server. It uses HTTP protocol to communicate between clients and websites to have data access.

**Why is Web API required?**

The user wants to access the application from different devices like mobile, browser, Google devices, etc. In this case, Web API can be useful.

Different devices request to Web API and Web API will respond in JSON format. Most of the devices are able to understand JSON output.

Let’s see the below web Api Architecture diagram,



**This diagram explains the architecture of Web API.**

1. A client called api/controller – In the above diagram Browsers, Phones, and Google Devices are called Web API Controllers.
2. api/Controller interact with business layer and get Data from DB.
3. The output will be returned in JSON format.

**What is ASP.NET Core Web API?**

ASP.NET Core Web API is a RESTful service created using ASP.NET Core. ASP.NET Core Web API is a cross-platform web API.

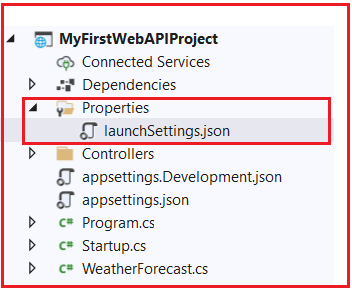
**Understanding the**

**ASP.NET Core Web API Project Structure:**

**Reference URL:** [ASP.NET Core Web API Files and Folders - Dot Net Tutorials](https://dotnettutorials.net/lesson/asp-net-core-web-api-files-and-folders/#:~:text=The%20ASP.NET%20Core%20Web,shown%20in%20the%20below%20image.)

##### **Properties:**

The **Properties Folder** in ASP.NET Core Web Application by default contains one JSON file called as **launchsettings.json file** as shown in the below image.



The launchsettings.json file contains some settings that are going to be used by .NET Core Framework when we run the application either from Visual Studio or by using .NET Core CLI. Another point that you need to keep in mind, the launchSettings.json file is only used within the local development machine. So, this file is not required when we publishing our ASP.NET Core Web API application into the production server.

##### **appsettings.json file:**

The next file that we are going to discuss is the appsettings.json file. This is the same as web.config or app.config of our traditional .NET Application. The appsettings.json file is the application configuration file in ASP.NET Core Web Application used to store the configuration settings such as database connections strings, any application scope global variables, etc.

##### **Startup.cs class file:**

The Startup class is like the Global.asax file of our traditional .NET application. As the name suggests,

it is executed when the application starts.

The Startup class includes two public methods: ConfigureServices and Configure.

##### **ConfigureServices() method:**

The ConfigureServices method of the Startup class is the place where we can register our dependent classes with the built-in IoC container. Once we register the dependent classes, they can be used anywhere within the application. The ConfigureServices method includes the IServiceCollection parameter to register services to the IoC container.

##### **Configure() method:**

The Configure method of the Startup class is the place where we configure the application request pipeline using the IApplicationBuilder instance that is provided by the built-in IoC container. ASP.NET Core introduced the middleware components to define a request pipeline, which will be executed on every request.

**Working with ASP.NET Core Web API:**

Main() method calls a custom CreateHoldBuilder() method.

There is a Host generic class on which public static method CreateHoldBuilder() is called.

CreateHostBuilder() method returns a HostBuilder and on HostBuilder Build() is called to create a Host and that host is Run(). Based on how it is configured, Build() method creates a host and Run() method starts Hosting. WebHost is Run() means it will start listening on the endpoints it is configured for. From where it is supposed to listen, that instruction is given in launchSettings.json.

In the CreateHostBuilder(), you are supposed to configure customized hosting parameter.

If you don’t configure, then there is a certain default configuration done on the Host. That’s why it is called as a builder.

**Default Configurations:**

Logging is enabled. Logging to the Debug Window and Console Window is enabled.

By default, logging destination is Debug Window and Console.

If you want to write a log to the file, you need to customize.

The Kestral server is internally started.

The timeout is set to 5 seconds. If the Host doesn’t start in 5 seconds, it will be timed out. You will get an error.

You can change this startup timeout.

Other things that you can set are content route and web route.

Default HostBuilder tells .NET Core that .json is its Configuration File.

This is part of Default Host Configuration.

When WebAPI Bootstraps, the Main() method is called. When Main() method is called,

Main-🡪CreateHostBuilder()🡪 The control is handed over to the Startup class.

The instance of Startup() class is created after the WebHost is created.

The constructor gets called. The injection is done.

Then two functions are called. ConfigureServices() and Configure().

**ConfigureServices():** Register the services that you need in your application. Whatever external services (E.g. – Swagger, CORS, Versioning etc – Built-in Services) your application may require are registered over here. User Defined Services can be registered in ConfigureServices().

**Configure()** method configures the middleware.

**Routing in ASP.NET Core Web API:d**

When it comes to listing the best practices for REST APIs, the mechanism,

Routing always makes its place on the top of the stack.

For novice APIs developers, technical consultants, and all other IT professionals associated with REST APIs, especially with a Microsoft technologies stack, this article will explain the importance and capabilities of Routing focusing Attribute Routing in REST APIs with Microsoft’s .NET Core.

Routing is a functionally based tag or Uri template used by APIs to match the desired action or methods expected to be executed. There are **two types** or rather two different types of Routing being used during development. Namely, ‘**Convention-based Routing**’ and ‘**Attribute Routing**’.

In ‘Convention-based Routing’, route templates are defined by developers as per requirement, basically a set of strings of type text decorated with parameters. Once the request is received, it tries to match requested URI with this defined route templates. The only merit of using this routing type is, templates are defined at a single location in application solution structures, leveraging the template rules religiously across the controllers and actions.

Then, why is Attribute routing important? Yes, it is not only important but strongly recommended for API development by developers and architects across the communities. Though convention-based routing has its own Pros, while building a good API, there are few considerations, where this type of routing is not advisable. There are common URI patterns in REST APIs, which are tough to support by convention-based routing. Consider, a set of Response data or resources, are often clubbed with their hierarchical data or child resources. For eg. Departments have Employees, Songs have singers, Movies have actors and so on. URIs expected in such scenarios are,

/movies/1/actors

In the case of multiple controllers and huge resources this type of URI is difficult though achievable using convention-based routing, but at the cost of scaling and performance. This hits the key consideration area of designing scalable APIs. Here is where another routing type, Attribute Routing, plays a role.

**Attribute Routing**

**What is Attribute Routing?**

Technically, Attribute routing is all about attaching a route, as an attribute, to a specific controller or action method. Decorating Controller and its method with [Route] attribute to define routes is called Attribute Routing. In simpler terms,

* using [Route] attribute with controllers and method is Attribute Routing.

[Route ("api/customers/{id}/orders")]

It started from Web API 2 and now is the most recommended and adapted Routing type in RESTful APIs design and development.

**Why use Attribute Routing?**

As the name indicates, attribute routing uses attributes to define routes. Attribute routing gives you precise control over the URIs than convention-based routing in your APIs. Above described scenario of Hierarchical resources can be easily achieved by Attribute Routing, with making no compromise with the scalability of APIs.

Also, versioning APIs, overloading URI segments and multiple parameter type patterns can be achieved through attribute routing with ease.

**Working with Attribute Routing**

Any route attribute on the controller makes all actions in the controller attribute routing. Defining route attribute to the action or the controller takes precedence over conventional routing. Let’s be more precise to .NET Core APIs, it comes by default with Attribute routing. Attribute routing requires detailed input to specify a route. However, it allows more control of which route template applies to each action.

**Content Negotiation in ASP.NET Core Web API:**

ASP.NET Core MVC supports formatting response data,

using specified formats or in response to a client's request.

## Format-specific Action Results

Some action result types are specific to a particular format, such as [JsonResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.jsonresult) and [ContentResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.contentresult). Actions can return results that always use a specified format, ignoring a client's request for a different format. For example, returning JsonResult returns JSON-formatted data and returning ContentResult returns plain-text-formatted string data.

An action isn't required to return any specific type. ASP.NET Core supports any object return value. Results from actions that return objects that aren't [IActionResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.iactionresult) types are serialized using the appropriate [IOutputFormatter](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.ioutputformatter) implementation. For more information, see [Controller action return types in ASP.NET Core web API](https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-6.0).

By default, the built-in helper method [ControllerBase.Ok](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.ok) returns JSON-formatted data:

[HttpGet]

public IActionResult Get() =>

Ok(\_todoItemStore.GetList());

The sample code returns a list of todo items. Using the F12 browser developer tools or [Postman](https://www.getpostman.com/tools) with the previous code displays:

* The response header containing **content-type:** application/json; charset=utf-8.
* The request headers. For example, the Accept header. The Accept header is ignored by the preceding code.

To return plain text formatted data, use [ContentResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.contentresult) and the [Content](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.content) helper:

[HttpGet("Version")]

public ContentResult GetVersion() =>

Content("v1.0.0");

In the preceding code, the Content-Type returned is text/plain.

For actions with multiple return types, return IActionResult. For example, when returning different HTTP status codes based on the result of the operation.

## Content negotiation

Content negotiation occurs when the client specifies an [Accept header](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html). The default format used by ASP.NET Core is [JSON](https://json.org/). Content negotiation is:

* Implemented by [ObjectResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.objectresult).
* Built into the status code-specific action results returned from the helper methods. The action results helper methods are based on ObjectResult.

When a model type is returned, the return type is ObjectResult.

The following action method uses the Ok and NotFound helper methods:

[HttpGet("{id:long}")]

public IActionResult GetById(long id)

{

var todo = \_todoItemStore.GetById(id);

if (todo is null)

{

return NotFound();

}

return Ok(todo);

}

By default, ASP.NET Core supports the following media types:

* application/json
* text/json
* text/plain

Tools such as [Fiddler](https://www.telerik.com/fiddler) or [Postman](https://www.getpostman.com/tools) can set the Accept request header to specify the return format. When the Accept header contains a type the server supports, that type is returned. The next section shows how to add additional formatters.

Controller actions can return POCOs (Plain Old CLR Objects). When a POCO is returned, the runtime automatically creates an ObjectResult that wraps the object. The client gets the formatted serialized object. If the object being returned is null, a 204 No Content response is returned.

The following example returns an object type:

[HttpGet("{id:long}")]

public TodoItem? GetById(long id) =>

\_todoItemStore.GetById(id);

In the preceding code, a request for a valid todo item returns a 200 OK response. A request for an invalid todo item returns a 204 No Content response.

### The Accept header

Content negotiation takes place when an Accept header appears in the request. When a request contains an accept header, ASP.NET Core:

* Enumerates the media types in the accept header in preference order.
* Tries to find a formatter that can produce a response in one of the formats specified.

If no formatter is found that can satisfy the client's request, ASP.NET Core:

* Returns 406 Not Acceptable if [MvcOptions.ReturnHttpNotAcceptable](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions.returnhttpnotacceptable#microsoft-aspnetcore-mvc-mvcoptions-returnhttpnotacceptable) is set to true, or -
* Tries to find the first formatter that can produce a response.

If no formatter is configured for the requested format, the first formatter that can format the object is used. If no Accept header appears in the request:

* The first formatter that can handle the object is used to serialize the response.
* There isn't any negotiation taking place. The server is determining what format to return.

If the Accept header contains \*/\*, the Header is ignored unless RespectBrowserAcceptHeader is set to true on [MvcOptions](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions).

### Browsers and content negotiation

Unlike typical API clients, web browsers supply Accept headers. Web browsers specify many formats, including wildcards. By default, when the framework detects that the request is coming from a browser:

* The Accept header is ignored.
* The content is returned in JSON, unless otherwise configured.

This approach provides a more consistent experience across browsers when consuming APIs.

To configure an app to respect browser accept headers, set the [RespectBrowserAcceptHeader](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions.respectbrowseracceptheader#microsoft-aspnetcore-mvc-mvcoptions-respectbrowseracceptheader) property to true:

C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>

{

options.RespectBrowserAcceptHeader = true;

});

### Configure formatters

Apps that need to support extra formats can add the appropriate NuGet packages and configure support. There are separate formatters for input and output. Input formatters are used by [Model Binding](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-6.0). Output formatters are used to format responses. For information on creating a custom formatter, see [Custom Formatters](https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/custom-formatters?view=aspnetcore-6.0).

### Add XML format support

To configure XML formatters implemented using [XmlSerializer](https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer), call [AddXmlSerializerFormatters](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.mvcxmlmvcbuilderextensions.addxmlserializerformatters):

C#

var builder = WebApplication.CreateBuilder(args);

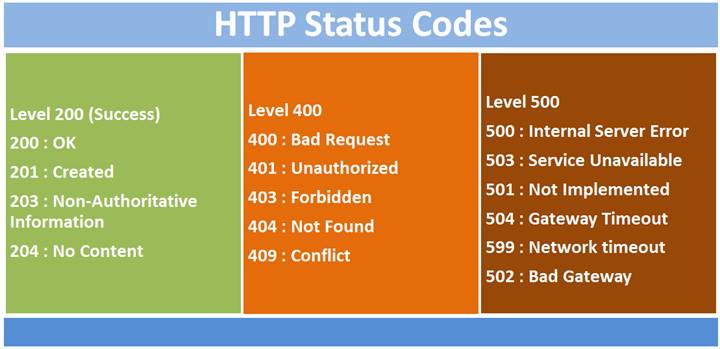
builder.Services.AddControllers()

.AddXmlSerializerFormatters();

When using the preceding code, controller methods return the appropriate format based on the request's Accept header.

**Working with HttpStatusCodes:**

The HTTP code Status Codes are important because they tell consumers about what exactly happened to their request; A wrong HTTP code can confuse the consumer. A consumer should know via the response if his/her request has been well taken care of or not, and if the response is not as expected, the Status Code should tell the consumer where the problem is - if it is at consumer level or at API level?

Suppose there is a situation where consumer gets response as status code 200 but at the service level, there is some problem or issue. In that case, consumer will get a false assumption of everything being fine, whereas that won’t be a case. So if there is something wrong at service or there occurs some error on the server, the status code, so that the consumer knows that there actually is something wrong with the request that it sent. In general, there are a lot of access codes. One can find the complete list [here](http://www.restapitutorial.com/httpstatuscodes.html), but not all are so important 500 should be sent to consumer except the few. Few status codes are very frequently used with the normal CRUD operations that a service performs, so service does not necessarily have to support all of them. Let’s have a glance over a few of the important status codes.  
  
****

**200--- success**

**400----client**

**500---server**

When we talk about the levels of status codes, there are 5 levels. Level 100 status codes are more of informal nature. Level 200 status codes are specifically for request being sent well. We get 200 codes for success of a GET request, 201 if a new resource has been successfully created. 204 status codes is also for success but in return it does not returns anything, just like if consumer has performed delete operation and in return doesn’t really expect something back. Level 300 http status codes are basically used for redirection, for example to tell a consumer that the requested resource like page, image has been moved to another location. Level 400 status codes are meant to state errors or client error for e.g. status code 400 means Bad Request, 401 is Unauthorized that is invalid authentication credentials or details have been provided by the consumer, 403 means that authentication is a success, but the user is not authorized. 404 are very common and we often encounter which mean that the requested resource is not available. Level 500 are for server errors. Internal Server Error exception is very common, that contains the code 500. This error means that there is some unexpected error on the server and client cannot do anything about it. We’ll cover how we can use these HTTP status codes in our application.

Request and response are the backbones of any RESTful Web API. Every status code has a specific meaning and during the development, we must make sure that we are returning a valid response with a valid status code.

**What is the Status Code?**

Status code is a numeric value, and it is the main component of HTTP Response. The status code is issued by the server based on the operation, input data, and some other parameters.

The status code gives some useful information about the behavior of the response.

**Status code categories**

All status codes are divided into the following 5 categories,

**1xx – Informational**

**2xx – Successful**

**3xx – Redirection**

**4xx – Client Error**

**5xx – Server Error**

**Commonly Used Status Codes:**

**200 status code**

This is the most common status code that is returned from Web API.

This 200 status code belongs to the Successful category. It means everything about the operation is successful.

**Example**

Get all employees data

Get single employee data

Asp.Net Core has **Ok() method** to **return 200 status code.**

**201 Status code**

201 status code indicates that the new resource has been created successfully and the server will return the link to get that newly created resource.

Asp.Net Core has **Created() method** to return **201 status code.**

**400 status code**

400 status code indicated the bad request. It means there is something wrong in the request data.

In asp.net core we can return **400 status** using the **BadRequest()** method.

**404 status code**

If we are looking for a resource that does not exist, then the server returns a 404 status code.

In asp.net core we can return **404 status** using the **NotFound() method.**

**Middle ware:**

It is a piece of code that perform some action.

**How can we identify the middleware?**

app .use

**Dependency Injection:**

* Simplifies the unit testing

It is a mechanism which dependency automatically created in object.

3 types

1. Add singleton----single instance will serve all the request throughout the lifetime of application.

Ex: login (ATM Machine)

1. Add scoped-----new instance will be created per session

Ex: maintain state (Amazon)

1. Add transient-----new instance will be created per request

Ex: no need to maintain state

**CreatedAtRoute vs CreatedAtAction**

Now we know a lot about these two methods and, therefore, can discuss them together. Here are some ***key points*** to consider:

* Both of them provide different ways of achieving the same thing - returning 201 Created response with a **Location** header
* Both of them check correctness and existence of the target action, thus, ensuring validness of the returned URI
* **CreatedAtAction** finds target action by method and controller names **CreatedAtRoute** finds target action by a route name
* **CreatedAtAction** requires action name, default is controller method name but can be assigned with **ActionName** attribute **CreatedAtRoute** requires a name of the target route, it can be assigned to particular route by us or declared in Startup
* **CreatedAtRoute** covers functionality of **CreatedAtAction** when using overload that doesn’t require **routeName** parameter

As always, which one to use depends on the use case. In general, **CreatedAtRoute** gives more options and includes functionality of **CreatedAtAction**. However, in some cases **CreatedAtAction** is more convenient, for example, when handling actions inside of the same controller.

* ***The main difference*** is how you specify the URI of the created resource which will be included in the Location header.
* **Created** gives you ***more control*** over URI creation, whereas **CreatedAtAction** and **CreatedAtRoute** gives ***more safety***.
* If your server is behind ***load balancer*** or ***reverse proxy***, you might want to use **Created** method

## Created Explained

This method is very straightforward and has only two overloaded versions. Detailed signatures of both methods can be found [here](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.created?view=aspnetcore-5.0).

Since **Created** method is very simple, it needs only these two pieces of information:

* **uri** - simply the URI that should be returned in the **Location** header
* **value** - content to return in a response body

Basically, the main difference between them is how you pass the value of the location header - as **string** or **Uri**.

**Important points** about passing location header using these methods:

* We are responsible for ensuring URI correctness and existence on the server at all times, for example, when we update paths or arguments.
* Using **string** means that value will be returned as is, without any validation.
* **Uri** [class](https://docs.microsoft.com/en-us/dotnet/api/system.uri?view=net-5.0) helps ensure that the URI format is correct. However, it doesn’t check that this path exists on the server.
* Value can be either absolute or relative.

## CreatedAtAction Explained

This method provides more support in generating URI for the Location header.

As the name suggests, this method allows us to set Location URI of the newly created resource by **specifying the name of an action** where we can retrieve our resource.

**Using Strongly Typed HttpClients:**

In these days, it’s almost impossible to implement a decent system without implementing or consuming a web API.

Building APIs has always been a first-class citizen and it has been evolving ever since in any programming platform with any major platform update or how we deploy our software.

In Microsoft related technologies, we have evolved from implementing an ASMX web service to implementing and deploying an API using Azure functions (yes, WCF, MVC, Web API approaches are, still widely used).

In modern (well maybe not so modern) times consumption of a web service has never been easy with the introduction of response types such as JSON (application/json). In this post I am going to discuss about how to consume a REST based API where the responses are in JSON format. For that I’ll be using .NETs HttpClient .

Alright, so what’s the big deal?

Well, as developers we would like to write beautiful, functioning code. So, when we saw HttpClient was implementing IDisposable interface, we used it within an using statements because that’s the sensible (A.K.A — best practice) thing to do. But… boy we were wrong! 😆

Well that’s one object which you shouldn’t be messing with by wrapping inside an using statement.

By default the behavior of an HttpClient instance is it’s thread safe and can be reused. So if you have many HttpClient instances created; each instance will have its own connection pool (which basically means, there will be many sockets opened in a given time). The problem is, though you dispose the HttpClient instance, the underlying sockets will not be released that soon. So, if you get a burst or if your API needs to serve many requests at the same time, it will fail to do at a certain point in time because the sockets will be exhausted.

**Behold..here comes .NET Core 2.1**

Well starting from .NET Core 2.1 Microsoft saw this becoming a real problem and introduced HttpClientFactory . As per the name, it’s a factory which we can use to create HttpClient instances without worrying about disposables, DNS changes and all that drama.

And, to be honest they did it in style! 👍 👍 👍

You can use the HttpClient in many ways. In this article I am going to demonstrate my favorite way. That’s by using a “Typed Client” approach. Also known as the “Service Agent Pattern”.

**CORS (Cross Origin Resource Sharing):**

**What is CORS?**

Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction is called the same-origin policy. The same-origin policy prevents a malicious site from reading sensitive data from another site. Sometimes, you might want to allow other sites to make cross-origin requests to your app.

Cross-Origin Resource Sharing (CORS) manages the cross-origin requests. Unlike same-origin policy, CORS allows making a request from one origin to another. CORS allows the servers to specify who can access the resource on the server from outside.

The origin is made up of three parts - the protocol, host, and the port number.

**Same Origin:**

<https://www.something.com:8080/demo.html> is sending request to

<https://www.something.com:8080/test.html>

**Cross Origin:**

<https://www.something.com:8080/demo.html> is sending request to

<https://www.anything.com:8080/demo.html>

<https://www.something.com:8080/demo.html> is sending request to

<https://www.something.com:9090/test.html>

<https://www.something.com:8080/demo.html> is sending request to

<http://www.something.com:8080/test.html>

**Enable CORS in ASP.NET Core API Application**

**Enabling CORS Globally**

Open your ASP.NET Core API application .

Go to Startup.cs file and add the below code in Configure method, which will inject CORS into a container.

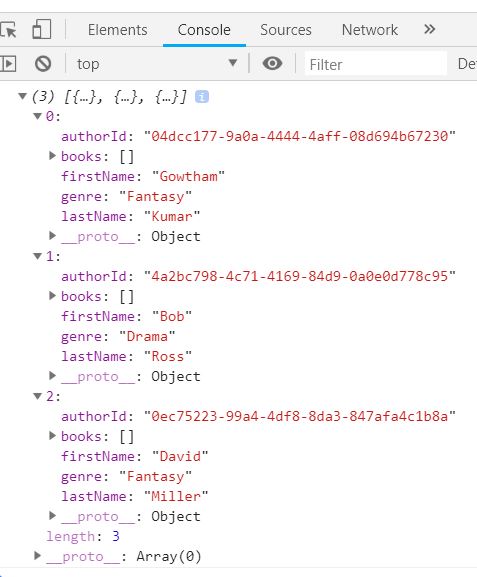
1. app.UseCors(options => options.AllowAnyOrigin());

Add the below code in ConfigureServices method,

1. services.AddCors(c =>
2. {
3. c.AddPolicy("AllowOrigin", options => options.AllowAnyOrigin());
4. });

The above code tells that the API’s can be accessed from any origin globally.

Run the application,



From the above figure you can notice we got a response from the API successfully and the response it printed in browser console was as expected.

**Enabling for origin**

Go to Startup.cs file and add the below code in Configure method,

1. app.UseCors(options=>options.WithOrigins("https://localhost:44342"));

Add the below code in ConfigureServices method

1. services.AddCors(c =>
2. {
3. c.AddPolicy("AllowOrigin", options => options.WithOrigins("https://localhost:44342"));
4. });

Go to controller and decorate the action with Enable CORS attribute, as given below,

1. // GET: api/Libraries/GetAllAuthor
2. [HttpGet]
3. [Route("GetAllAuthor")]
4. [EnableCors("AllowOrigin")]
5. **public** IActionResult GetAllAuthor()
6. {
7. IEnumerable<Author> authors = \_libraryRepository.GetAllAuthor();
8. **return** Ok(authors);
9. }

Now this API can be accessed only from the origin https://localhost:44342.

We can also define EnableCors at the controller level so that all the actions under this controller can be accessed from the origin https://localhost:44342

1. [Route("api/Libraries")]
2. [ApiController]
3. [EnableCors("AllowOrigin")]
4. **public** **class** LibrariesController : ControllerBase
5. {
6. **private** **readonly** ILibraryRepository<Author> \_libraryRepository;


10. **public** LibrariesController(ILibraryRepository<Author> libraryRepository)
11. {
12. \_libraryRepository = libraryRepository;
13. }
15. // GET: api/Libraries/GetAllAuthor
16. [HttpGet]
17. [Route("GetAllAuthor")]
19. **public** IActionResult GetAllAuthor()
20. {
21. IEnumerable<Author> authors = \_libraryRepository.GetAllAuthor();
22. **return** Ok(authors);
23. }
24. }
25. }

**Swagger:**

**Reference URLs:**

[ASP.NET Core web API documentation with Swagger / OpenAPI | Microsoft Docs](https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-6.0)

[ASP.NET Core 3.1 Web API and Swagger (c-sharpcorner.com)](https://www.c-sharpcorner.com/article/asp-net-core-3-1-web-api-and-swagger/#:~:text=For%20adding%20Swagger%20or%20OpenAPI,Swagger%20to%20the%20DI%20container)

Swagger (OpenAPI) is a language-agnostic specification for describing REST APIs. It allows both computers and humans to understand the capabilities of a REST API without direct access to the source code. Its main goals are to:

* Minimize the amount of work needed to connect decoupled services.
* Reduce the amount of time needed to accurately document a service.

The two main OpenAPI implementations for .NET are [Swashbuckle](https://github.com/domaindrivendev/Swashbuckle.AspNetCore) and [NSwag](https://github.com/RicoSuter/NSwag), see:

* [Getting Started with Swashbuckle](https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-6.0)
* [Getting Started with NSwag](https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-nswag?view=aspnetcore-6.0)

## OpenApi vs. Swagger

The Swagger project was donated to the OpenAPI Initiative in 2015 and has since been referred to as OpenAPI. Both names are used interchangeably. However, "OpenAPI" refers to the specification. "Swagger" refers to the family of open-source and commercial products from SmartBear that work with the OpenAPI Specification. Subsequent open-source products, such as [OpenAPIGenerator](https://github.com/OpenAPITools/openapi-generator), also fall under the Swagger family name, despite not being released by SmartBear.

In short:

* OpenAPI is a specification.
* Swagger is tooling that uses the OpenAPI specification. For example, OpenAPIGenerator and SwaggerUI.

## OpenAPI specification (openapi.json)

The OpenAPI specification is a document that describes the capabilities of your API. The document is based on the [XML](https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-6.0#xml-comments) and attribute annotations within the controllers and models. It's the core part of the OpenAPI flow and is used to drive tooling such as SwaggerUI. By default, it's named openapi.json. Here's an example of an OpenAPI specification, reduced for brevity:

JSON

{

"openapi": "3.0.1",

"info": {

"title": "API V1",

"version": "v1"

},

"paths": {

"/api/Todo": {

"get": {

"tags": [

"Todo"

],

"operationId": "ApiTodoGet",

"responses": {

"200": {

"description": "Success",

"content": {

"text/plain": {

"schema": {

"type": "array",

"items": {

"$ref": "#/components/schemas/ToDoItem"

}

}

},

"application/json": {

"schema": {

"type": "array",

"items": {

"$ref": "#/components/schemas/ToDoItem"

}

}

},

"text/json": {

"schema": {

"type": "array",

"items": {

"$ref": "#/components/schemas/ToDoItem"

}

}

}

}

}

}

},

"post": {

…

}

},

"/api/Todo/{id}": {

"get": {

…

},

"put": {

…

},

"delete": {

…

}

}

},

"components": {

"schemas": {

"ToDoItem": {

"type": "object",

"properties": {

"id": {

"type": "integer",

"format": "int32"

},

"name": {

"type": "string",

"nullable": true

},

"isCompleted": {

"type": "boolean"

}

},

"additionalProperties": false

}

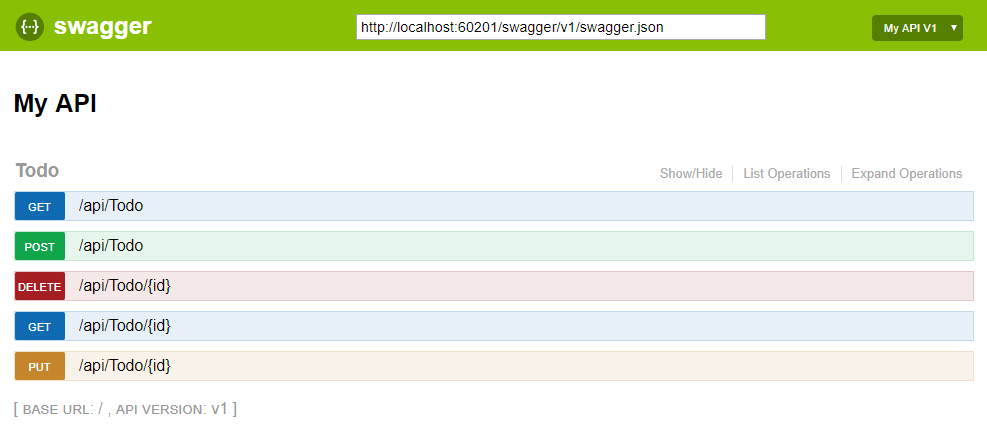
}

}

}

## Swagger UI

[Swagger UI](https://swagger.io/swagger-ui/) offers a web-based UI that provides information about the service, using the generated OpenAPI specification. Both Swashbuckle and NSwag include an embedded version of Swagger UI, so that it can be hosted in your ASP.NET Core app using a middleware registration call. The web UI looks like this:



Each public action method in your controllers can be tested from the UI. Select a method name to expand the section. Add any necessary parameters, and select **Try it out!**.

For adding Swagger or OpenAPI to an ASP.NET Web API service, you can use Swashbuckle. The NuGet package Swashbuckle.AspNetCore is the library for ASP.NET Core. After you add the NuGet package, you need to add Swagger to the DI container. AddSwaggerGen is an extension method to add swagger services to the collection. To configure Swagger, you invoke the method SwaggerDoc. Passing an Info object, you can define the title, description, contact information, and more in code file Startup.cs.

**Implement JWT In ASP.NET Core 3.1**

**Reference URLs:**

<https://www.codemag.com/Article/2105051/Implementing-JWT-Authentication-in-ASP.NET-Core-5#:~:text=JSON%20Web%20Tokens%20(commonly%20known,consumers%20in%20a%20secure%20manner>.

<https://www.c-sharpcorner.com/article/implement-jwt-in-asp-net-core-3-1/>

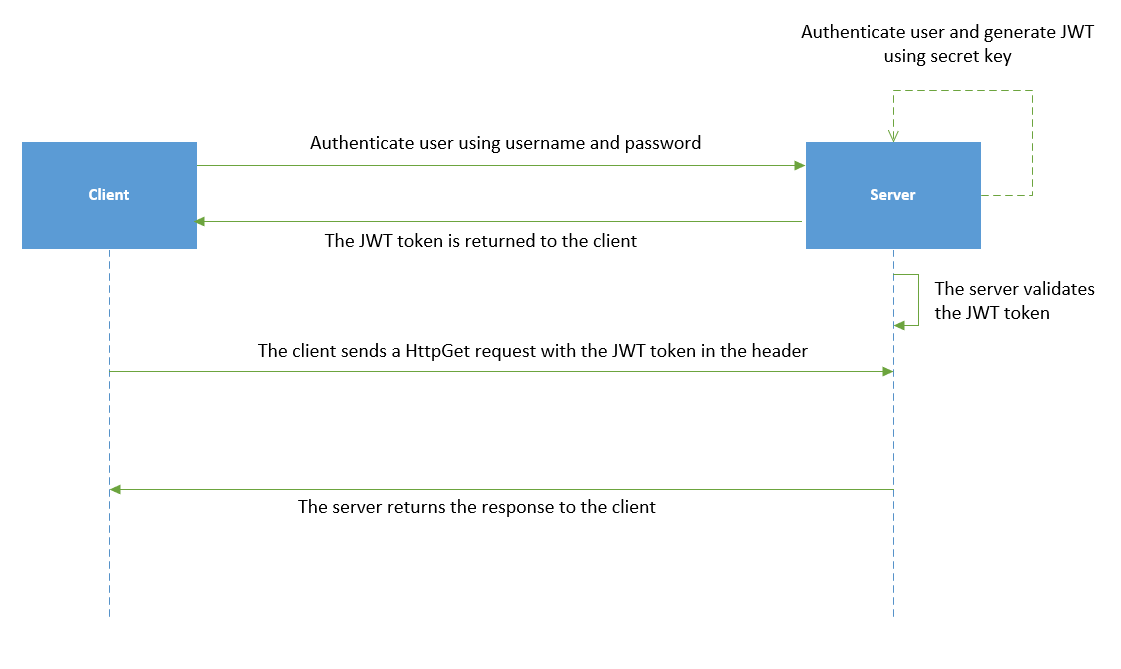
With the surge in APIs and their consumption globally, API security is extremely important these days. JWT authentication is a standard way for protecting APIs.

JSON Web Tokens (commonly known as JWT) is an open standard to pass data between client and server, and enables you to transmit data back and forth between the server and the consumers in a secure manner.

## What Are JSON Web Tokens (JWT)?

JSON Web Token is an open standard (RFC 7519) that defines a safe, compact, and self-contained, secured way for transmission of information between a sender and a receiver through a URL, a POST parameter, or inside the HTTP Header. It should be noted that the information to be transmitted securely between two parties is represented in JSON format and it is cryptographically signed to verify its authenticity. JWT is typically used for implementing authentication and authorization in Web applications. Because JWT is a standard, all JWTs are tokens but the reverse is not true. You can work with JSON Web Tokens in .NET, Python, Node.js, Java, PHP, Ruby, Go, JavaScript, etc.

Following figure illustrates how a typical JWT authentication works.



In this article we will cover the following,

* Creating the method to generate the JWT token
* Creating the middleware needed to validate the token
* Decorating the API controller
* Testing our API with Fiddler

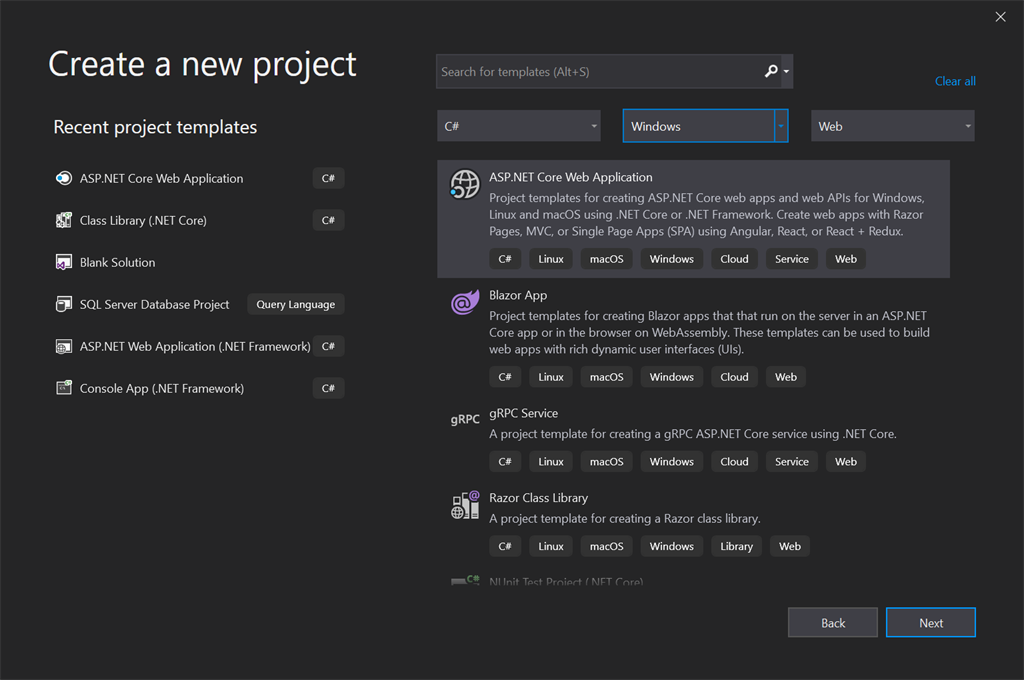
First things first, let's start with a brand new project. I am using VS 2019 Community Edition.

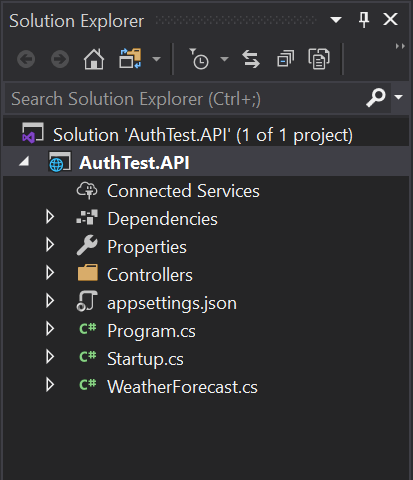
Create a new ASP.NET Core Web Application.  Choose the API with no authentication template.

I am calling my project AuthTest.API

IMPORTANT. I highly suggest you name your project, folders and classes the same as the article below, or you will find yourself having to track down and clean up the namespaces.

If you want the source code you can get it from [github](https://github.com/fscopel/token-based-authentication.git).





Good! Now let's do some coding. I am adding two folders to the project: Services and Middleware just for organization purposes.

In the Services folder add a class called JwtServices.cs

We also need some Nuget packages. Right-click Dependencies -> Manage Nuget Packages...  on the Browse tab search and install both of these packages:

* System.IdentityModel.Tokens.Jwt
* Microsoft.IdentityModel.Tokens

Let's fill in the JwtService class.

1. **using** System;
2. **using** System.Text;
3. **using** System.Security.Claims;
4. **using** Microsoft.IdentityModel.Tokens;
5. **using** System.IdentityModel.Tokens.Jwt;
6. **using** Microsoft.Extensions.Configuration;
8. **namespace** AuthTest.API.Services
9. {
10. **public** **class** JwtService
11. {
12. **private** **readonly** **string** \_secret;
13. **private** **readonly** **string** \_expDate;
15. **public** JwtService(IConfiguration config)
16. {
17. \_secret = config.GetSection("JwtConfig").GetSection("secret").Value;
18. \_expDate = config.GetSection("JwtConfig").GetSection("expirationInMinutes").Value;
19. }
21. **public** **string** GenerateSecurityToken(**string** email)
22. {
23. var tokenHandler = **new** JwtSecurityTokenHandler();
24. var key = Encoding.ASCII.GetBytes(\_secret);
25. var tokenDescriptor = **new** SecurityTokenDescriptor
26. {
27. Subject = **new** ClaimsIdentity(**new**[]
28. {
29. **new** Claim(ClaimTypes.Email, email)
30. }),
31. Expires = DateTime.UtcNow.AddMinutes(**double**.Parse(\_expDate)),
32. SigningCredentials = **new** SigningCredentials(**new** SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
33. };
35. var token = tokenHandler.CreateToken(tokenDescriptor);
37. **return** tokenHandler.WriteToken(token);
39. }
40. }
41. }

In the construtor I am pulling some data out of the appsettings.json, but we haven't added those settings yet. Don't worry - we will right after this.

The reason for it is that the JWT generator needs some kind of secret string, some kind of password if you will, and an expiration date to generate the token.

The secret can be anything you want, just like a random password. I just typed in some random letters and numbers, and I decided the expiration is 1440 minutes (24hrs).

That means, the users for my API will have to get a new token every 24 hrs. It can be anything you want. You could choose to only expire the token if the user logs out (not recommended) or you could renew the token every so often. I am not covering that here.

Pay special attention to the Subject property line 27 through 30.

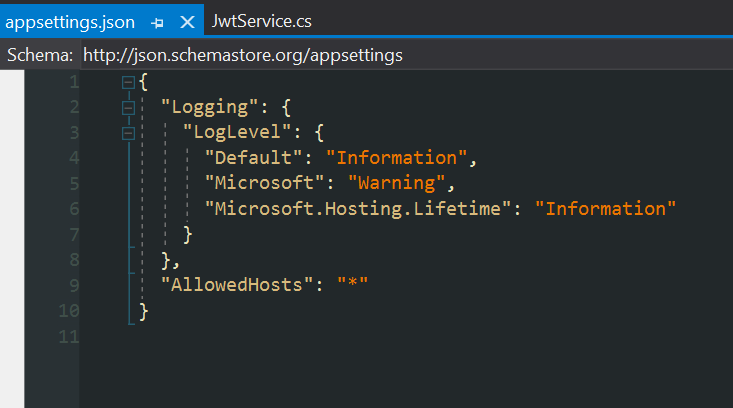
I am passing in an email to the function, GenerateSecurityToken(string email) and storing that email in the token. You could pass in some a user object GenerateSecurityToken(User user) for example and store a lot more information by adding new claims. This way you don't need to take trips to the DB to get that data, when the user makes a call into the system.

**Example**

Something like this...

1. **public** **string** GenerateSecurityToken(User user){
2. ...
3. Subject = **new** ClaimsIdentity(**new**[]
4. {
5. **new** Claim(ClaimTypes.Email, user.Email),
6. **new** Claim(ClaimTypes.Name, user.Name),
7. **new** Claim(ClaimTypes.Role, user.Role),
8. **new** Claim(ClaimTypes.DateOfBirth, user.DOB),
9. })
10. ...
11. }

We went on a bit of a tangent, let's get back to the appsettings.config. In the appsettings.json we need to add a new configuration.  This is what my appsettings.json looks like right now.



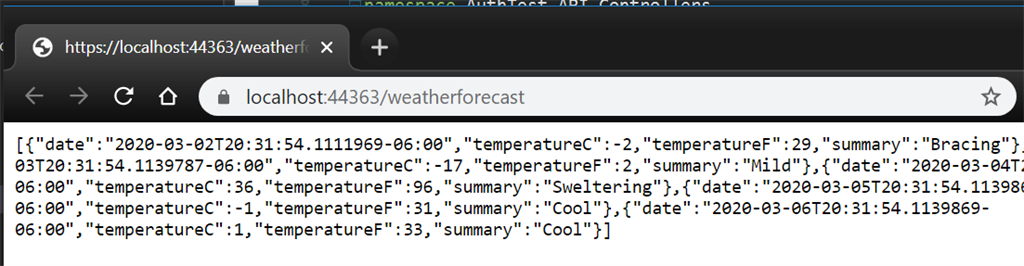
This is what it looks like after adding the JwtConfig section.

The JWT generation,

1. {
2. "Logging": {
3. "LogLevel": {
4. "Default": "Information",
5. "Microsoft": "Warning",
6. "Microsoft.Hosting.Lifetime": "Information"
7. }
8. },
9. "AllowedHosts": "\*",
10. "JwtConfig": {
11. "secret": "PDv7DrqznYL6nv7DrqzjnQYO9JxIsWdcjnQYL6nu0f",
12. "expirationInMinutes": 1440
13. }
14. }

We now have the secret and the expiration data points needed by the JwtService class.

Go ahead and run your app right now.  If Microsoft hasn't changed the template by the time you are following this article, you should probably get some fake weather json data on your browser.

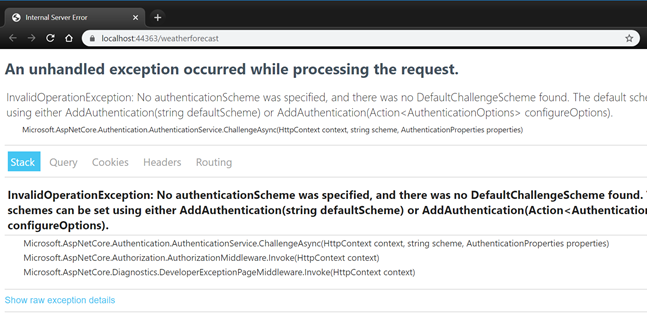


This means the app is working and currenlty not requiring any kind of authentication to serve up data. Let's go ahead and mess that up! :)

Go head add the [Authorize] attribute, you will need to bring in the Microsoft.AspNetCore.Authorization and try running the project again.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** Microsoft.AspNetCore.Authorization;
5. **using** Microsoft.AspNetCore.Mvc;
6. **using** Microsoft.Extensions.Logging;
8. **namespace** AuthTest.API.Controllers
9. {
10. [ApiController]
11. [Authorize]
12. [Route("[controller]")]
13. **public** **class** WeatherForecastController : ControllerBase
14. {
15. **private** **static** **readonly** **string**[] Summaries = **new**[]
16. {
17. "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
18. };
20. **private** **readonly** ILogger<WeatherForecastController> \_logger;
22. **public** WeatherForecastController(ILogger<WeatherForecastController> logger)
23. {
24. \_logger = logger;
25. }
27. [HttpGet]
28. **public** IEnumerable<WeatherForecast> Get()
29. {
30. var rng = **new** Random();
31. **return** Enumerable.Range(1, 5).Select(index => **new** WeatherForecast
32. {
33. Date = DateTime.Now.AddDays(index),
34. TemperatureC = rng.Next(-20, 55),
35. Summary = Summaries[rng.Next(Summaries.Length)]
36. })
37. .ToArray();
38. }
39. }
40. }

When I ran the project, I got the error below. The error means that ASP.NET Core sees the [Authorize] attribute but doesn’t know how to handle that, because we haven't configured a middleware to do so.  Let's go ahead a take care of that.



Back to the project go ahead and create a new class inside the Middleware folder, let's call this one AuthenticationMiddleware

Before we make any changes to this new class we need to bring one more Nuget package:

* Microsoft.AspNetCore.Authentication.JwtBearer

Browse and install the above package, and update the AuthenticationMiddleware class with the code below

1. **using** System.Text;
2. **using** Microsoft.IdentityModel.Tokens;
3. **using** Microsoft.Extensions.Configuration;
4. **using** Microsoft.Extensions.DependencyInjection;
5. **using** Microsoft.AspNetCore.Authentication.JwtBearer;
7. **namespace** AuthTest.API.Middleware
8. {
9. **public** **static** **class** AuthenticationExtension
10. {
11. **public** **static** IServiceCollection AddTokenAuthentication(**this** IServiceCollection services, IConfiguration config)
12. {
13. var secret = config.GetSection("JwtConfig").GetSection("secret").Value;
15. var key = Encoding.ASCII.GetBytes(secret);
16. services.AddAuthentication(x =>
17. {
18. x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
19. x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
20. })
21. .AddJwtBearer(x =>
22. {
23. x.TokenValidationParameters = **new** TokenValidationParameters
24. {
25. IssuerSigningKey = **new** SymmetricSecurityKey(key),
26. ValidateIssuer = **true**,
27. ValidateAudience = **true**,
28. ValidIssuer = "localhost",
29. ValidAudience = "localhost"
30. };
31. });
33. **return** services;
34. }
35. }
36. }

Now that we have the middleware built we need to hook it up to our services.

Open the Startup.cs class, find the ConfigurationServices, and the Configure functions and update with the code below.

You will have to import the reference to the namespace where the AuthenticationExtension is located

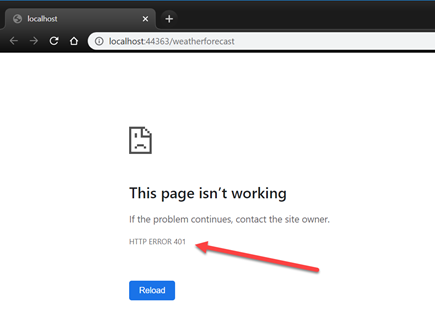
1. **using** AuthTest.API.Middleware;
2. **using** Microsoft.AspNetCore.Builder;
3. **using** Microsoft.AspNetCore.Hosting;
4. **using** Microsoft.Extensions.Configuration;
5. **using** Microsoft.Extensions.DependencyInjection;
6. **using** Microsoft.Extensions.Hosting;
8. **namespace** AuthTest.API
9. {
10. **public** **class** Startup
11. {
12. **public** Startup(IConfiguration configuration)
13. {
14. Configuration = configuration;
15. }
17. **public** IConfiguration Configuration { **get**; }
19. // This method gets called by the runtime. Use this method to add services to the container.
20. **public** **void** ConfigureServices(IServiceCollection services)
21. {
22. services.AddControllers();
23. services.AddTokenAuthentication(Configuration);
24. }
26. // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
27. **public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)
28. {
29. **if** (env.IsDevelopment())
30. {
31. app.UseDeveloperExceptionPage();
32. }
34. app.UseHttpsRedirection();
36. app.UseRouting();
37. app.UseAuthentication();
38. app.UseAuthorization();
40. app.UseEndpoints(endpoints =>
41. {
42. endpoints.MapControllers();
43. });
44. }
45. }
46. }

Cool! Let's go ahead a run a quick test! Go ahead and just start your app again.

Did you get a "This page isn't working" - If you did, good job!  That's exactlyy what we are lookin for. No more exceptions!

If you look closely you will see the server is displaying a 401 error - which means... drum roll... Unauthorized: https://httpstatuses.com/401

Our app now understands what we are looking for and since it didnt see a token in the request it returned an 401 - Unathorized error.



Stop the app and let's go ahead and create a new controller. Name this one TokenController

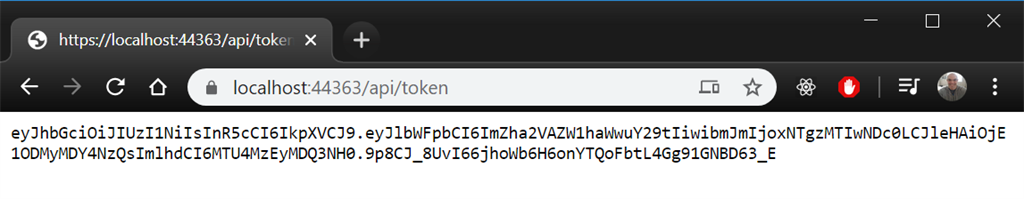
Right-click the Controllers folder and choose Add -> Controller...

Pick the API Controller - Empty template and click Add

1. **using** AuthTest.API.Services;
2. **using** Microsoft.AspNetCore.Mvc;
3. **using** Microsoft.Extensions.Configuration;
5. **namespace** AuthTest.API.Controllers
6. {
8. [Route("api/[controller]")]
9. [ApiController]
10. **public** **class** TokenController : ControllerBase
11. {
12. **private** IConfiguration \_config;
14. **public** TokenController(IConfiguration config)
15. {
16. \_config = config;
17. }
19. [HttpGet]
20. **public** **string** GetRandomToken()
21. {
22. var jwt = **new** JwtService(\_config);
23. var token = jwt.GenerateSecurityToken("fake@email.com");
24. **return** token;
25. }
26. }
27. }

Restart the app, and navigate to https://localhost:44363/api/token, your port number may vary.

Hopefully you got a token like me,



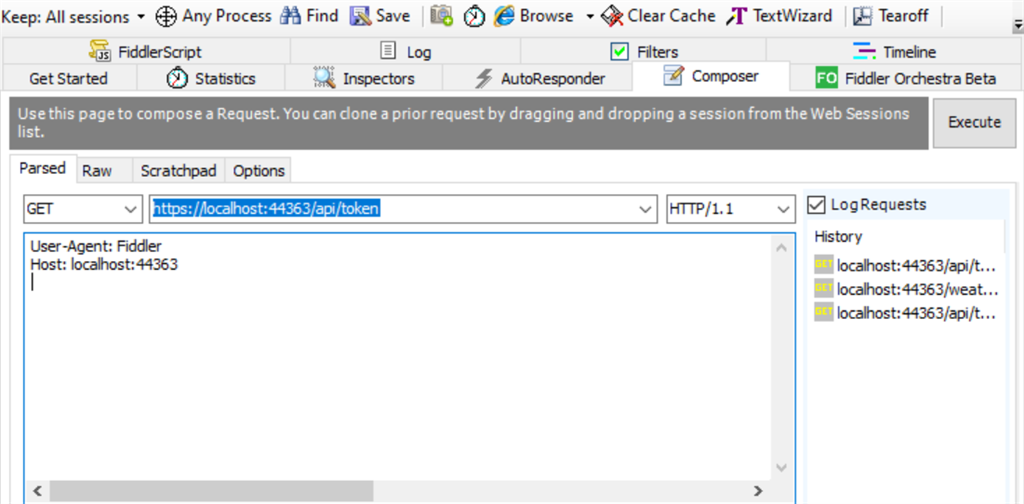
Now with Postman or Fiddler whichever tool you prefer, let's try to call into the WeatherForecastController and see if we can get through.

With the app running let's go ahead and make a call into the token endpoint to get a fresh token and then let's use that token to call into the weather forecast service.

Make sure your app is running end do a GET on .../api/token

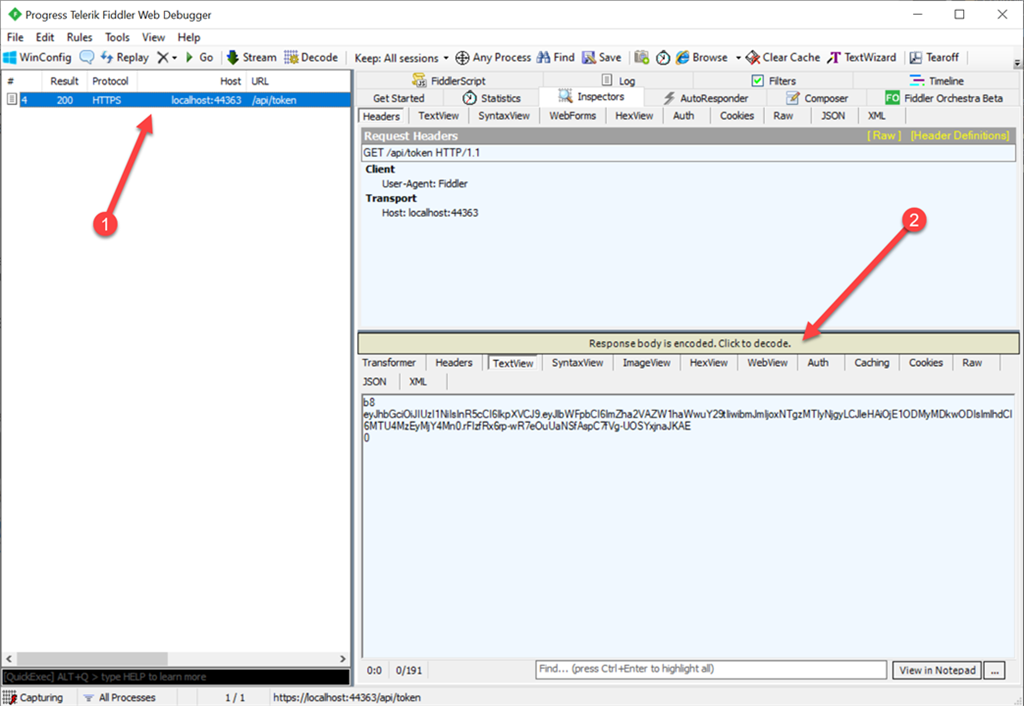
This is what fiddler looks like.

On the Composer tab choose GET from the dropdown and type in your URL, then click the Execute button.

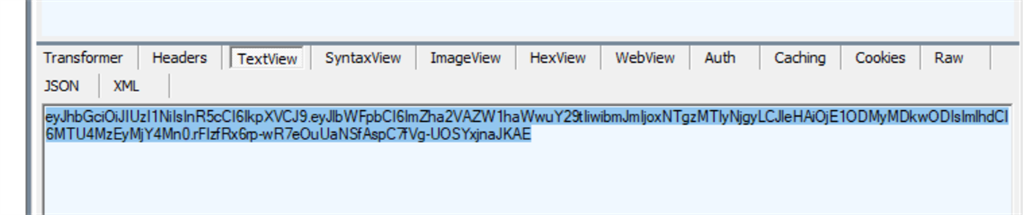


Double click the result on the left and then click on decode, to see your actual token.

Important: This is only happening because I am running my app in HTTPS. If I was running in HTTP, I would not need to decode the result.



After decoding the yellow text goes away and you can copy the token:



Let's compose another call into the WeatherForecast endpoint.

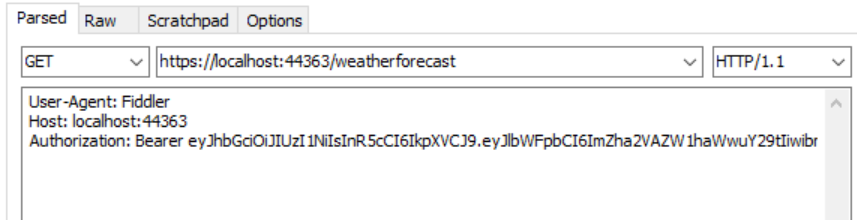
GET

header area:

User-Agent: Fiddler

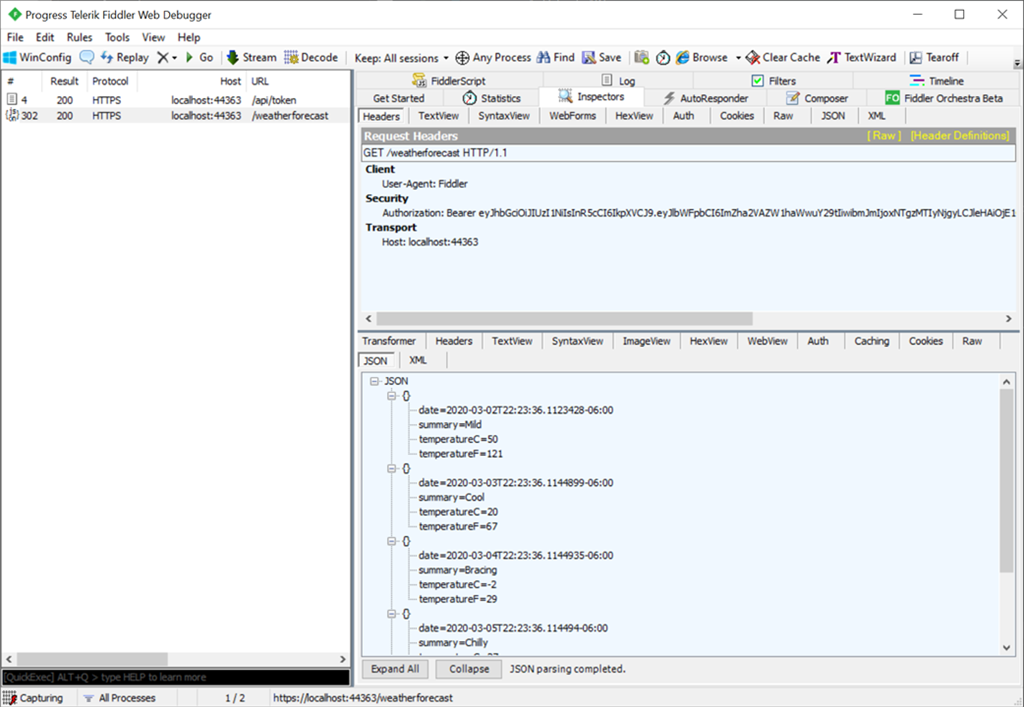
Host: localhost:44363

Authorization: Bearer YOUR TOKEN GOES HERE



Click the Execute button:

There it is - data!



To pass the token in Postman:

