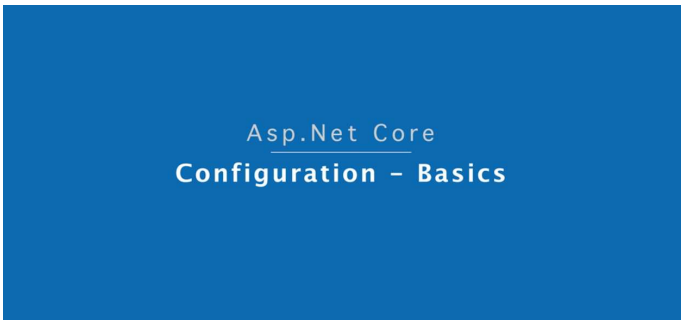
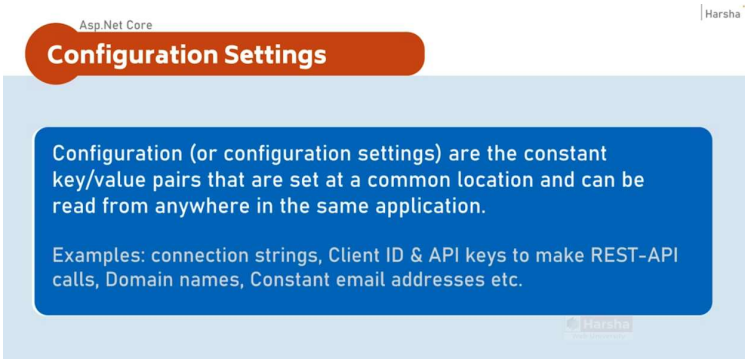
Section 14: Configuration [MVC and Web API]
Tuesday, January 21, 2025 5:11 PM





ASP.NET Core has a system called **Configuration Settings** to store constant key-value pairs that can be accessed from anywhere within the same application.

In real-world projects, you may need to store common artifacts that are accessible from different components, such as:
- Controllers
- Views
- Middleware
- Reusable services

For example, you might store:
- **Connection strings**
- **Client IDs** or **API keys** for REST API calls
- **Domain addresses**
- **Constant email addresses**, such as customer care contact information

These values are typically set once for the entire application and are reusable across various parts of the project.

Consider an example where you've registered a **Client ID** in the Google API, OpenWeatherMap, or any third-party API server. Using the provided **Client ID** and **Client Secret**, you can make REST API calls to that server. To avoid redundancy, you store the **Client ID** and **API key** in a central location within your application. This way, these values can be reused across different services within the project.

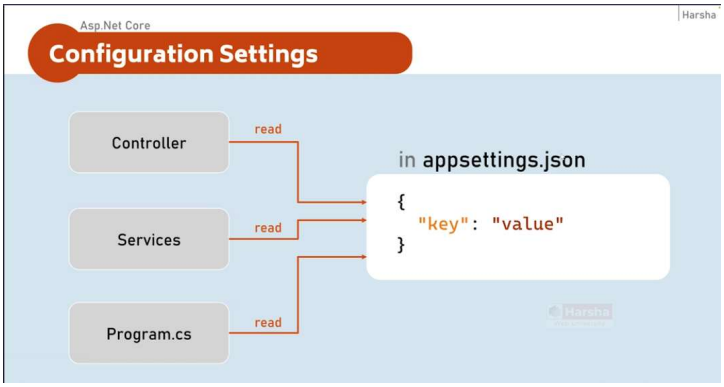
Whenever you need to make a REST API call to the same server, you access the **Client ID** and **API key** from the configuration settings. This ensures consistent and efficient usage of these values across the application.

This is the essence of the **Configuration Settings** system.

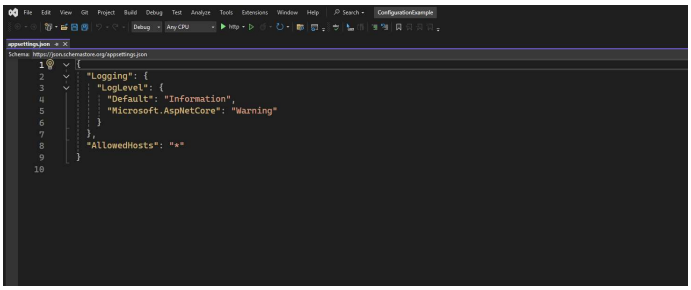
In this section, we will explore:
- **How to store configuration settings**
- **How to read configuration settings**

For example, by default, there is a file called appsettings.json in the project.
This file is automatically created every time you create a new ASP.NET Core project. It is used to store key-value pairs in JSON format, which can be accessed from anywhere in the entire application, such as:
- **Controllers**
- **Services**
- **Program.cs**
- Any other part of the application

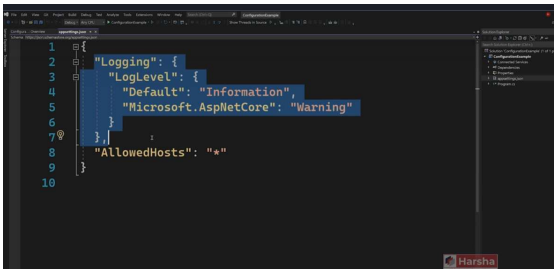
Let me demonstrate this practically.



appSetting.json is a simple file which contain some default settings.

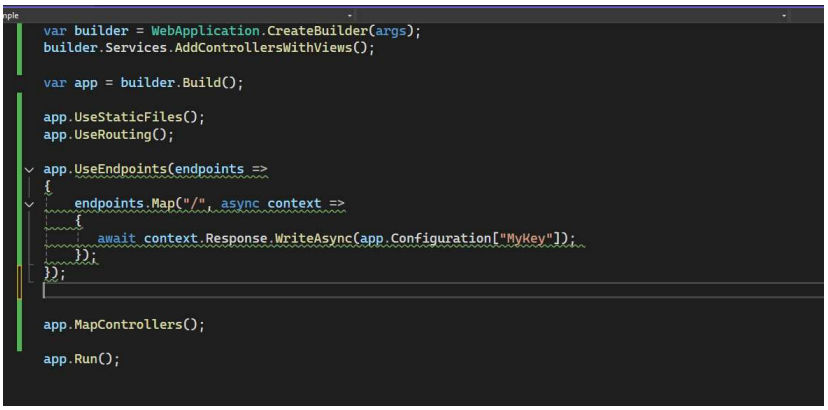


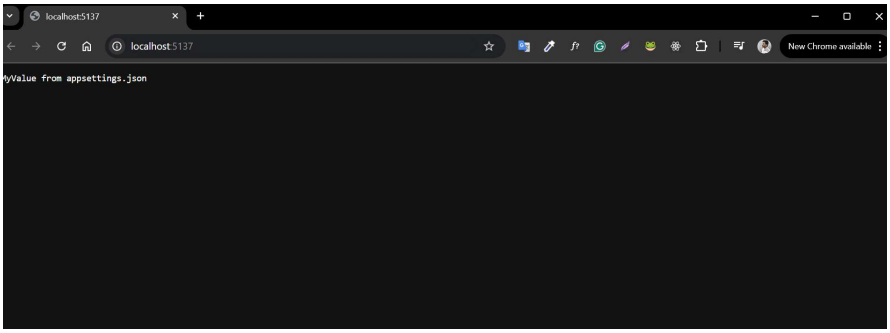
These configuration settings are for logging middleware. In case if you use built-in logger middleware then these configuration settings are useful. So currently ignore it.



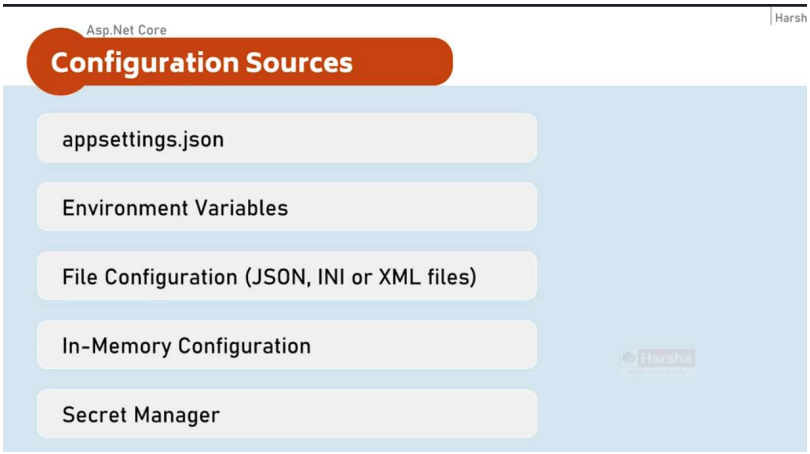
You can add any key with any value.

Now let's say I would like to read this key from 'Program.cs' file

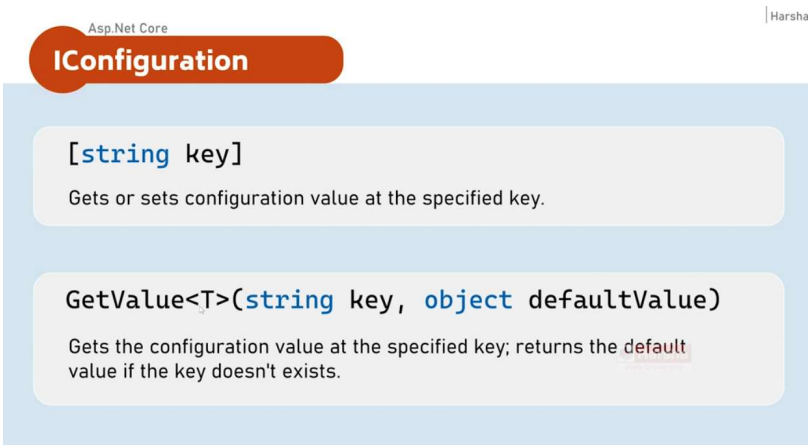




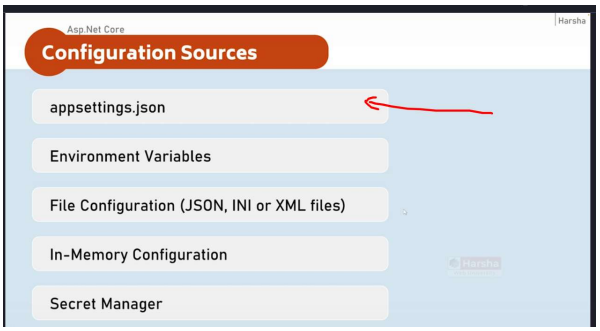
There are few more configuration sources.



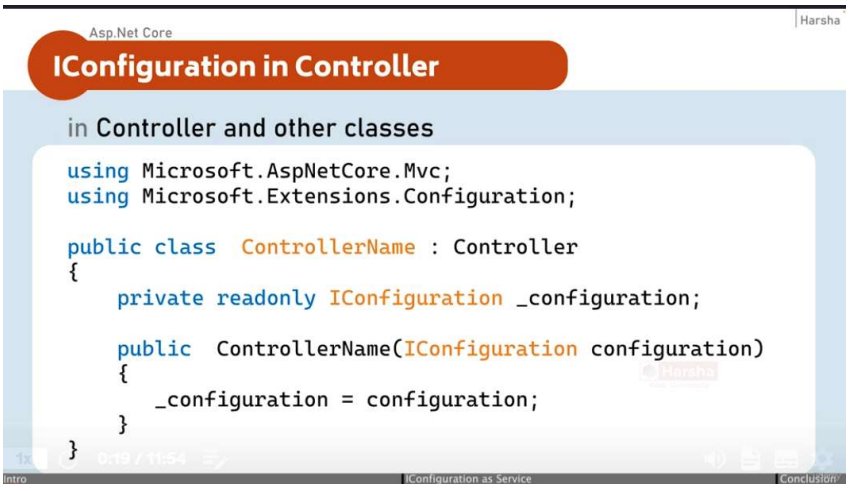
There are other way to read value from configuration.



Till now we have only explored the 'appsetting.json' file. We will explore other configuration sources later in this section.



Sometimes you would like to read the configuration settings programmatically in the controller or in your other services then you can inject **Iconfiguration** Interface as a service in your controller class.



For example, suppose you have registered with the **OpenWeather API** and want to make a request to fetch weather information from their website. This involves making a REST API request to weatherapi.com.

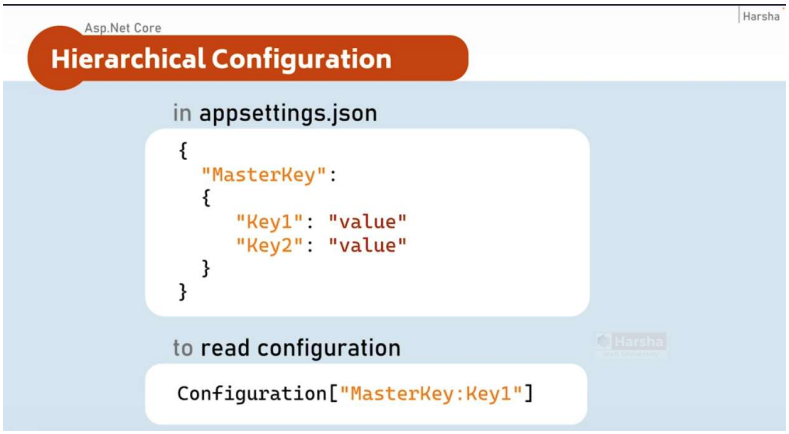
While making the request, you need the **Client ID** and **Client Secret**. If these details are already stored in the **configuration settings**, you would need to read them from there while making the request in your controller. Without these details, you cannot make a REST API call to the third-party server.
To achieve this, you will inject IConfiguration as a service into your controller or service.

In the actual method (for example, an action method in the controller), you can make the REST API request and access the configuration settings. Specifically, you use the _configuration field to retrieve the **Client ID** and **Client Secret** from the configuration settings.
To set this up:

1. Create a **private, read-only field** of type IConfiguration.

2. Inject IConfiguration via the constructor of your controller or service.
This allows you to access the configuration settings wherever needed in your application.

Like this, you can inject in any service to use it.



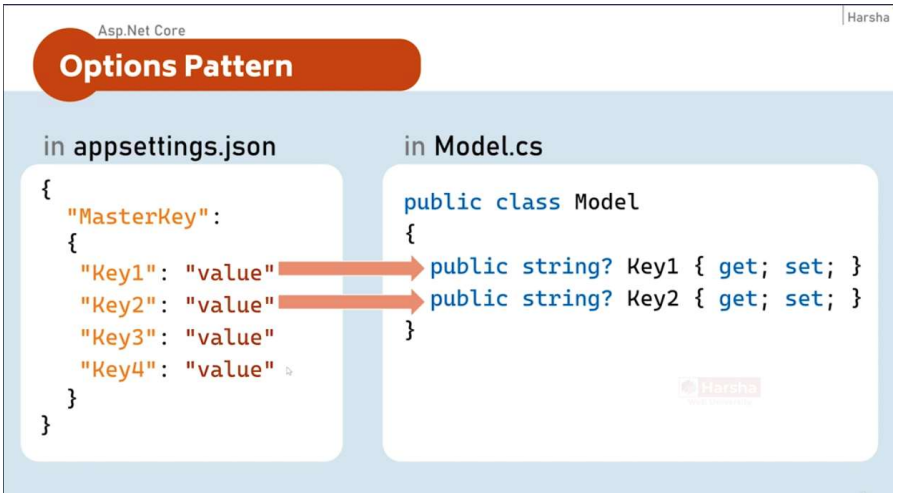
Sometimes, you may want to maintain **hierarchical configuration** in your application.
What does this mean? Instead of assigning a simple value to a key, you can create a **JSON object** under a key, which contains a set of key-value pairs.
For example, under the key WeatherApi, you can store the **Client ID** and **Client Secret** values as follows:

```
{
  "WeatherApi": {
    "ClientId": "your-client-id",
    "ClientSecret": "your-client-secret"
  }
}
```

In this case, both values (ClientId and ClientSecret) are nested under the key WeatherApi. This is called a **hierarchical configuration**.
To read these values in your application, you can use the **parent key** followed by the **child key**, separated by a colon (:). The colon acts as the default separator between keys.
For example, to read the ClientId, you would use:
WeatherApi:ClientId.

Let me demonstrate this practically.

We can also read with this.



The **Options Pattern** is used to specify the exact properties you want to read from the configuration.
For example, if your configuration source contains more than 10 properties (key-value pairs), but you only need one or two of them, you can define a model class with properties corresponding to the keys you want to use. This allows you to load only the necessary values into these properties, ignoring the rest.
To implement the **Options Pattern**:

1. **Create a model class**: Define properties in the class with names that exactly match the keys in your configuration.

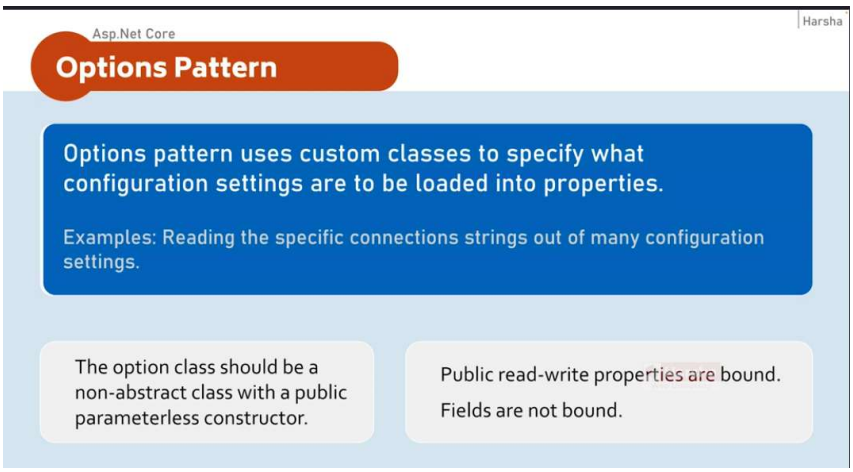
2. **Bind the configuration**: Use dependency injection to bind the configuration values to the properties of your model class.

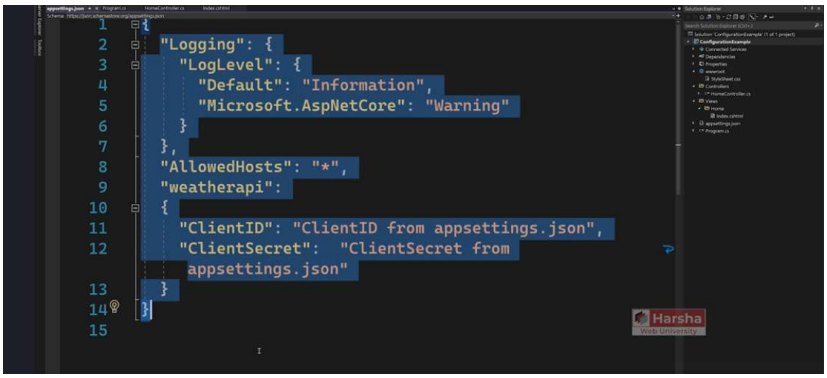
This approach ensures that only the values of the specified keys are loaded into the model, while the rest of the configuration remains unused.
Let me know if you'd like a practical example or further clarification!

For example, let's say in your configuration, you have connection strings, client ID, client API key, and many other details.
If you create a model class and specify the properties you want to read, only those exact properties will be read, and others will not.
This is called the **Options Pattern**.

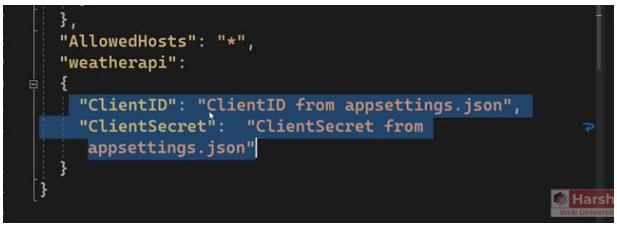
It is pretty useful, right? Because why should you read all the configuration details you don't need?
For implementing this Options Pattern, you need to create a normal C# class, called an **Options Class**, that contains the public properties.
The actual configuration values will be read from the configuration source into the properties of the class.

In this case, the options class should be a **non-abstract class** with a **parameterless constructor**.
All the properties should be **read/write properties** in public.
Fields are not bound, meaning even if you create one or more fields in your options class, those will be ignored.

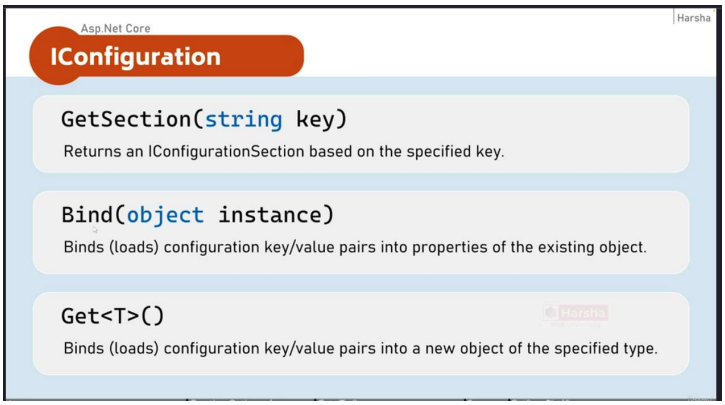




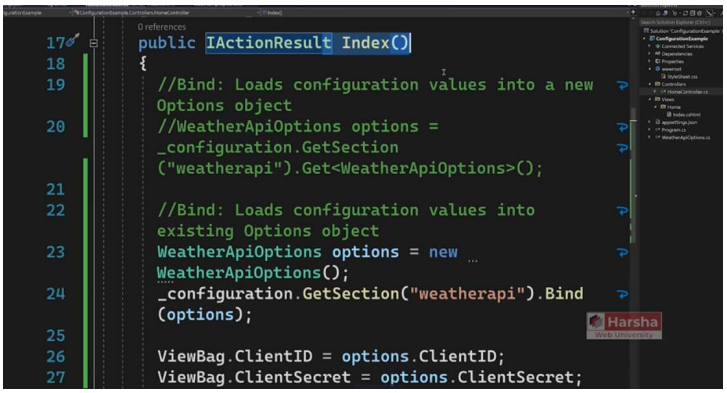
For example, in this appsettings.json file, you have a lot of configuration settings, but your point of interest is only these particular values: ClientId and ClientSecret.

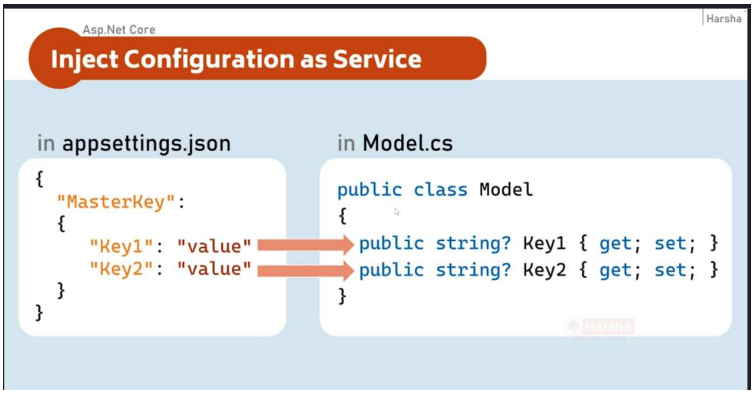


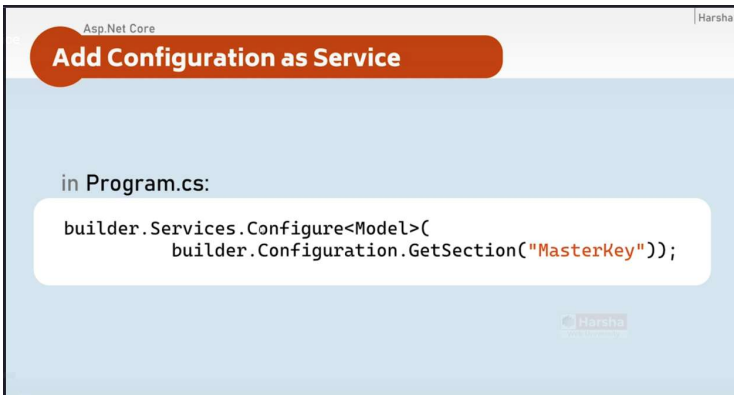
so in order to read them let's create an
options class with properties same name as the keys



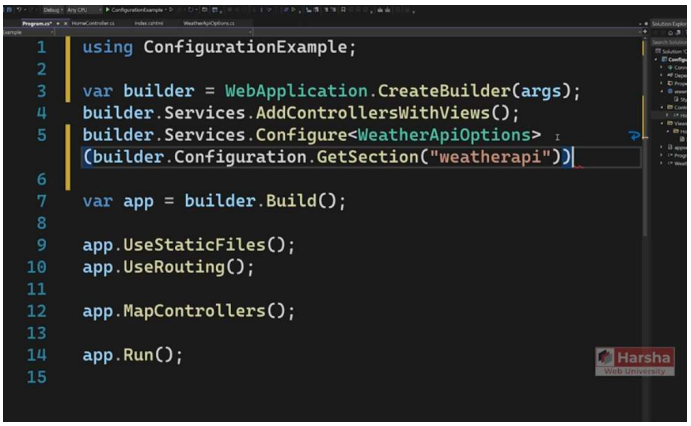
Rather than creating a new object of this WeatherApiOptions class, if we inject the same as a service in the constructor, that will be better, right?
Because you don't need to write an extra statement in the action method, the action method becomes simpler.
That is exactly what we are going to try in this lecture.

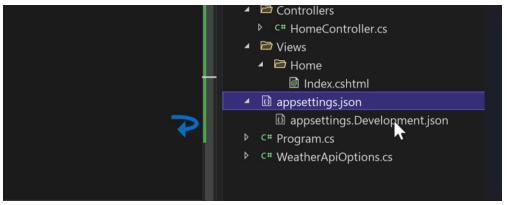






Hey services, I would like to add a configuration as a service. But what type of options? That is 'WeatherApiOptions'





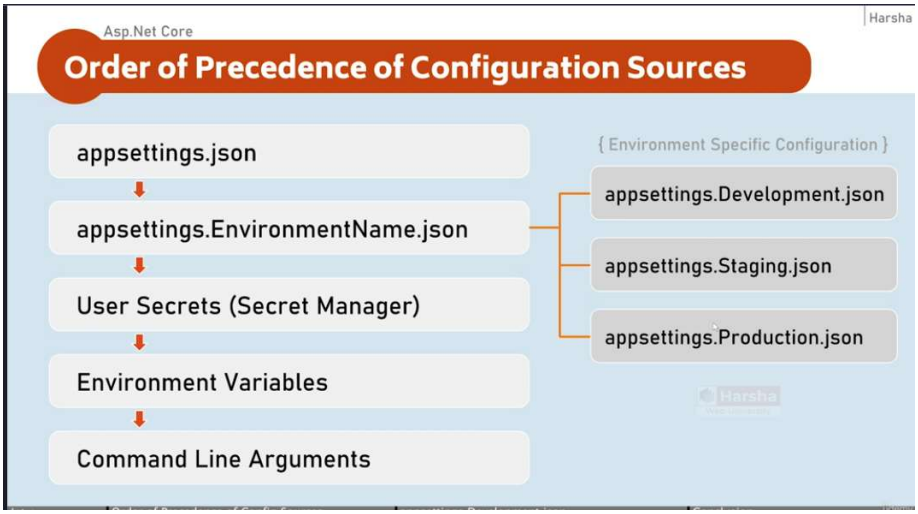
Did you notice in your Solution Explorer, apart from the appsettings.json file, if you expand this arrow mark, you can find one more file there: appsettings.Development.json.
When you create a new project, this file gets created automatically by default.
So, what's the purpose of that? What's the difference between appsettings.json and appsettings.Development.json?
Which takes more precedence? Which loads first, and which comes next?
Which can override the values of the other?
Let's try to understand.

This concept is called **Environment-Specific Configuration**.
In the last lecture, we discussed three types of environments and how to configure them: **Development**, **Staging**, and **Production**.

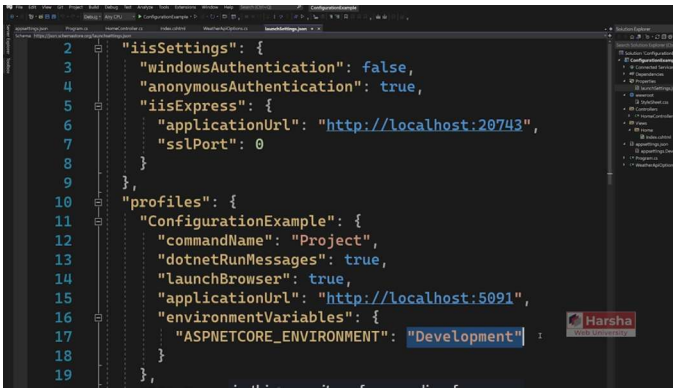
You can create different configuration files for each environment.

For example:
- This one is for the **Development** environment.
- This one is for the **Staging** environment.
- And this one is for the **Production** environment.

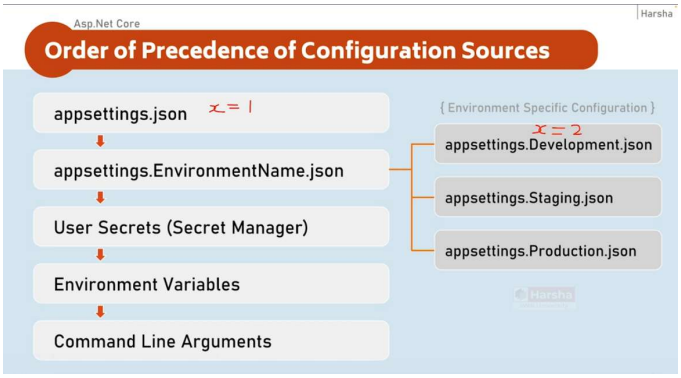
While reading the configuration from the sources, it first reads the value from the appsettings.json.
For example, if there is a key-value pair like x=1 in appsettings.json, it will then read the environment-specific file, depending on the current environment set in your **launch settings**.



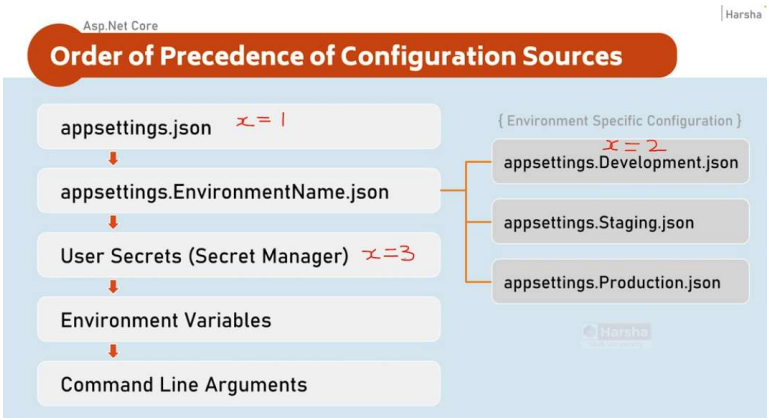
For example, Currently we are inside the development. So, it preferes reading form appSettings.devleopment.json



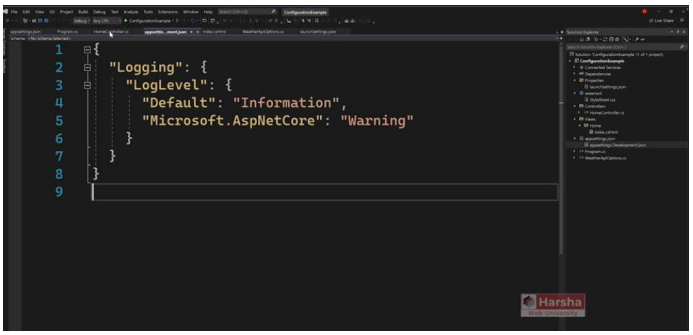
Of course, obviously, if you are in the **Production** environment, it loads the appsettings.Production.json based on the current environment.
For example, in appsettings.Development.json, if you have the same key, such as x=2, it overwrites the previous value.
In appsettings.json, the value of x is 1. But in the case of Development.json, x=2.
So, this takes more precedence because it is loaded later.



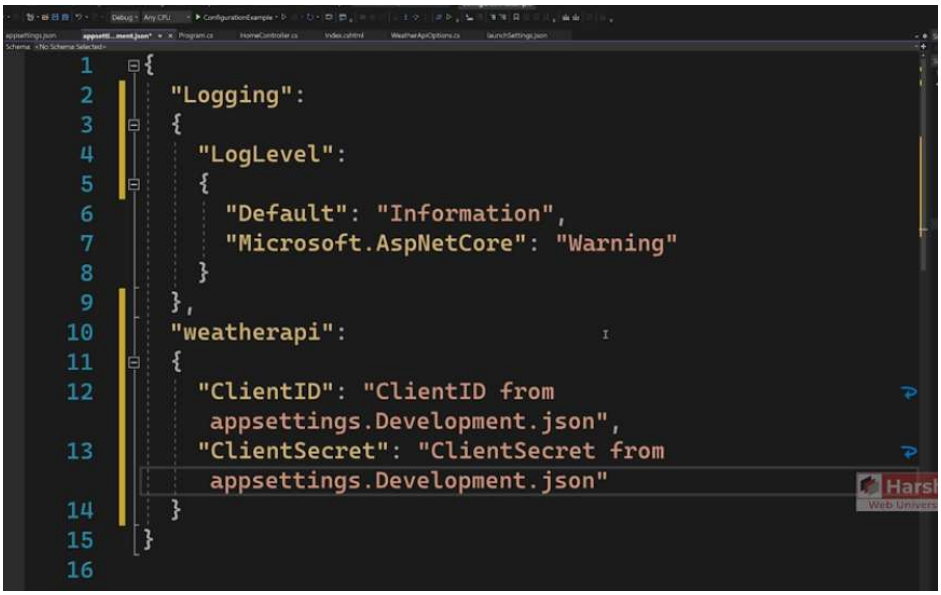
Then after that it reads the value from user secrets, it takes more precedence that means for example in the user secrets, you have mentioned x = 3, finally x=3 wins the race because it is loaded recently or last. Like that 'Environment Variables' and 'Command Line Argument' takes precedence, whichever is loaded last that value only be accessible in the actual code.



In this lecture, let us focus on this environment specific app settings. For example currently we are int the development environment. And now we have appSettings.development.json if we can open that it is also a normal json file just like app.settings.json and you can add any values.

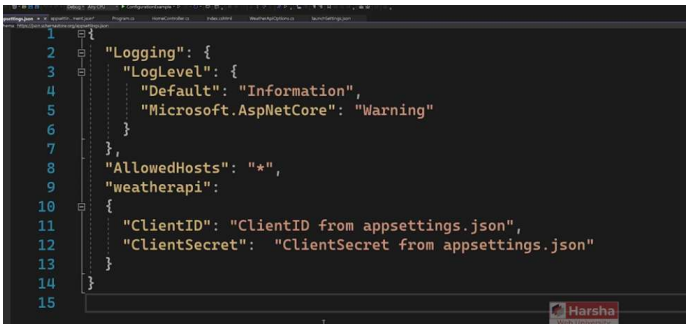


appSettings.development.json



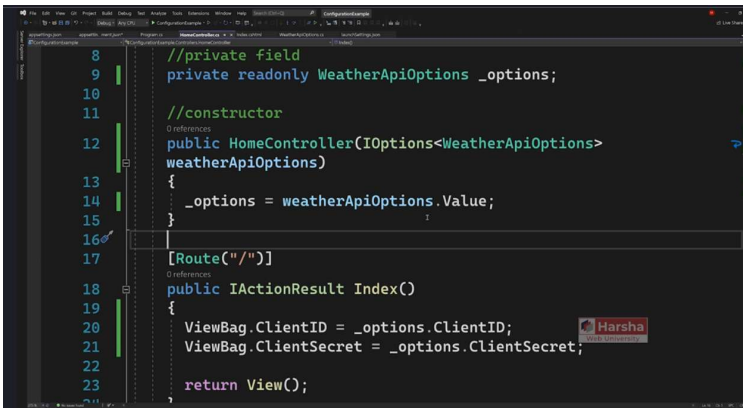
```
{
  "Logging":
  {
    "LogLevel":
    {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "weatherapi":
  {
    "ClientID": "ClientID from
     appsettings.Development.json",
    "ClientSecret": "ClientSecret from
     appsettings.Development.json"
  }
}
```

appSettings.json



```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "weatherapi":
  {
    "ClientID": "ClientID from appsettings.json",
    "ClientSecret":  "ClientSecret from appsettings.json"
  }
}
```

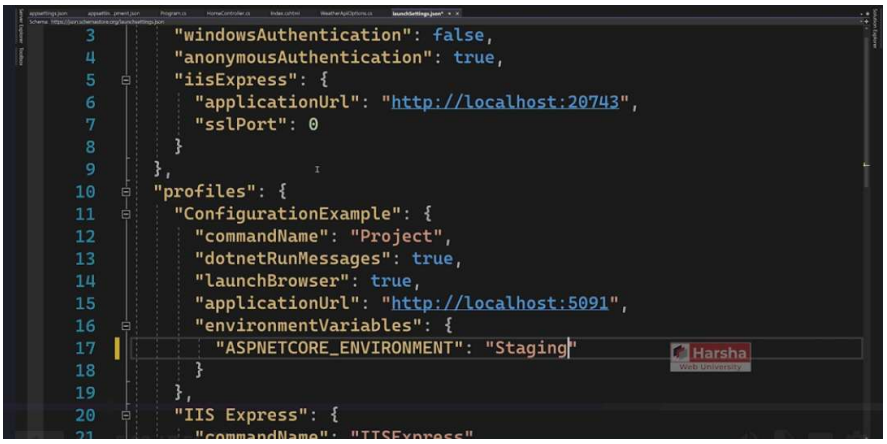
Let's run our app and see which value is going to show.



```
//private field
private readonly WeatherApiOptions _options;

//constructor
0 references
public HomeController(IOptions<WeatherApiOptions>
weatherApiOptions)
{
  _options = weatherApiOptions.Value;
}

[Route("/")]
0 references
public IActionResult Index()
{
  ViewBag.ClientID = _options.ClientID;
  ViewBag.ClientSecret = _options.ClientSecret;

  return View();
}
```

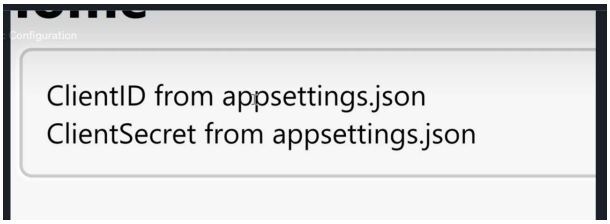
See we have got this! File reads from 'appSettings.Development.json'

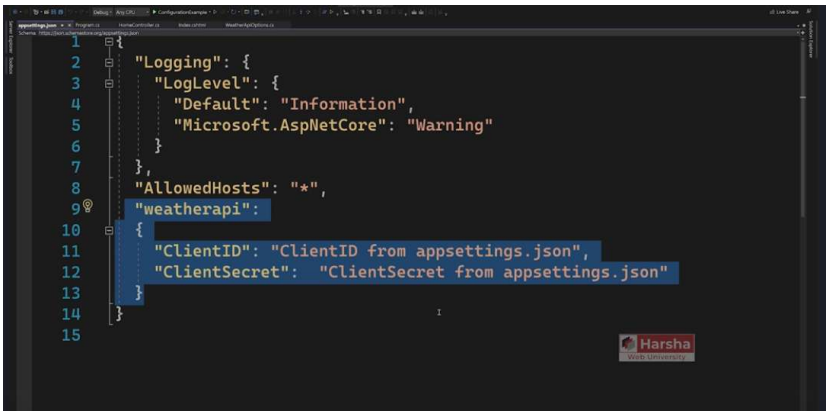
Now let's change the development environment from 'Development' to 'Staging'



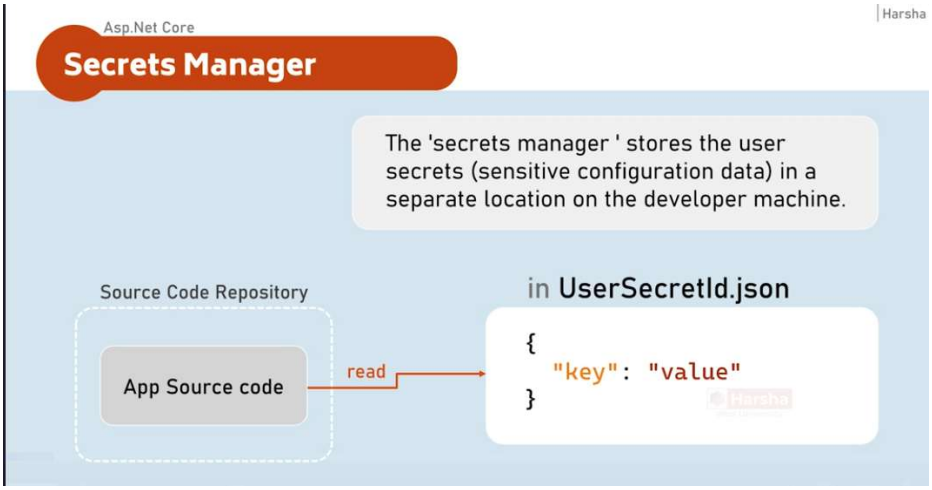
Now restart the application and run again.

Since we are in the Development environment so 'appSettings.Development.json' file was not loaded.





Think about the security of the configuration values stored in the configuration files.
For example, if you store important sensitive information in these configuration files, such as client ID, client secret, passwords, or any other sensitive data, and push the source code into a source control system like GitHub, the sensitive information will be exposed to other developers.
This is not secure. Sensitive information should be stored locally and outside the source code of the project.
The solution to this problem from Microsoft is the **Secret Manager**.
In .NET Core, you can use the Secret Manager to store sensitive information outside the source code of the project.
If you store sensitive information in configuration files as part of the source code, it will be uploaded to the source code repository (e.g., GitHub or other version control systems), which is unsafe.
As an alternative, you can use the Secret Manager.



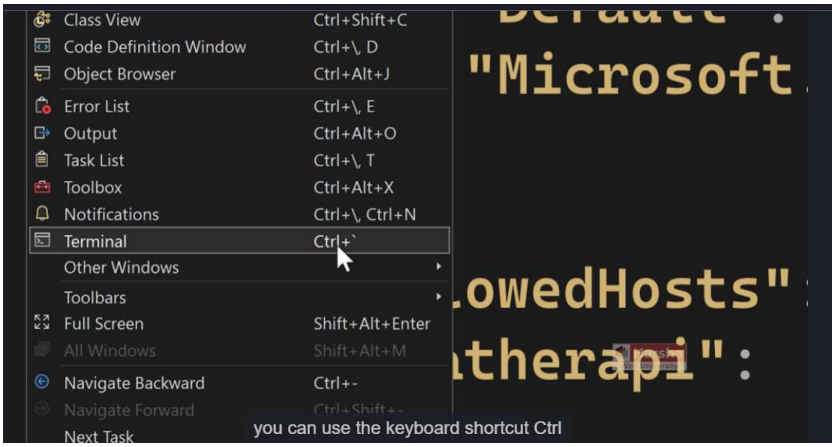
But how does the Secrets Manager help store sensitive information outside the source code?

You can enable the Secrets Manager with the dotnet command and set key-value pairs using the same.

These key-value pairs will be stored in a separate location on the developer machine, specifically in the **AppData** folder on Windows. This setup is unique to each developer's machine.
In this way, sensitive information is not stored in the configuration file as part of the source code. Instead, it is stored outside the source code, preventing it from being pushed to source control.

So it will not be pushed into source control.
In this way, sensitive information will remain on the same developer machine and will not be revealed to other developers.
To work with these commands, you can use either Windows PowerShell or the Developer PowerShell window in Visual Studio.
In this case, I am using Developer PowerShell.

In Visual Studio, go to the **View** menu and select **Terminal**.
You can use the keyboard shortcut **Ctrl + Backtick**.



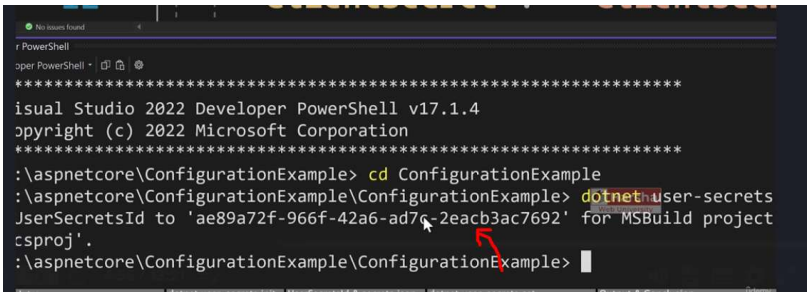
It opens up **Developer PowerShell** in the same Visual Studio window.
Once it opens, this **Developer PowerShell** by default locates the current working solution folder.

But we have to locate the project folder where your **.csproj** file is present.
In my case, the solution folder name and the project folder name are the same.
This one is the solution folder. Enter the command cd followed by the project folder name, which is **ConfigurationExample**.
Make sure you have located the project folder.

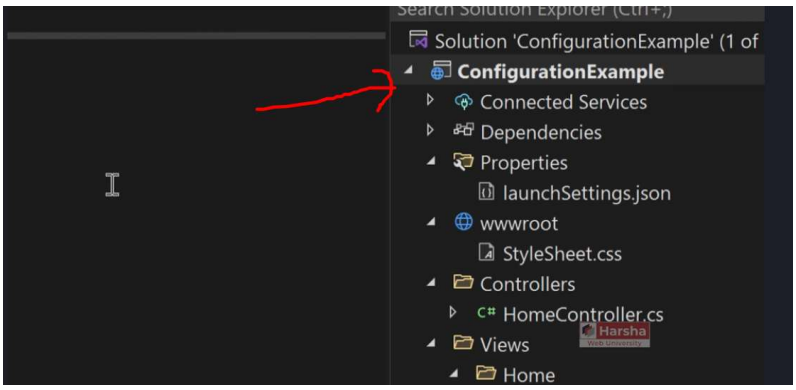
Now type this command,

It will initialize the user secrets. That means it internally does the two things, one is automatically generates a hexadecimal code as user secret ID and with that name a json file will be created means hexadecimal code.json that will be stored in the appData folder and that user secret ID will be stored as a part of the csproj version file in the current working asp.net core project.

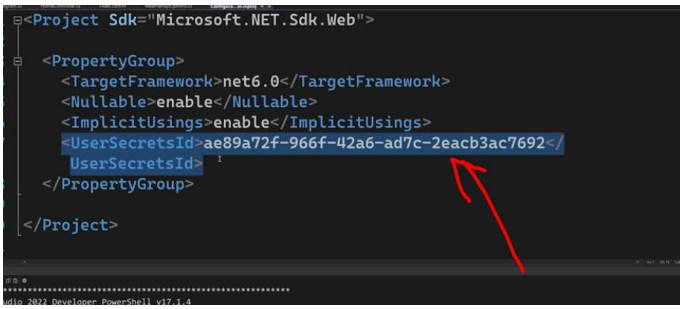
User secret Id is set to this particular location.



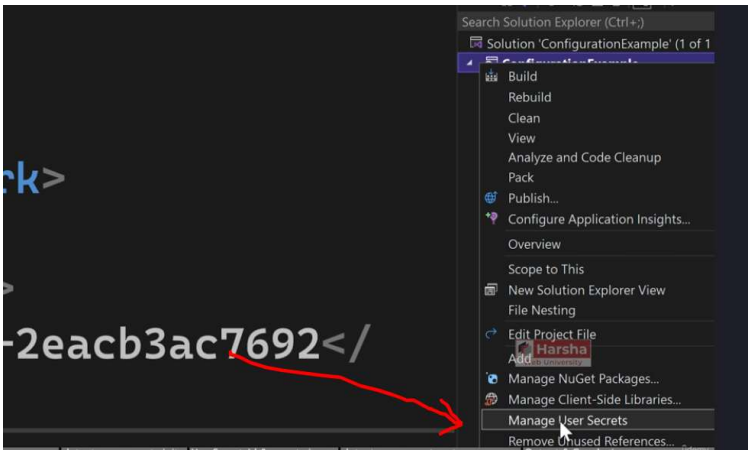
Double Click on this project name, it will open .csproj file.

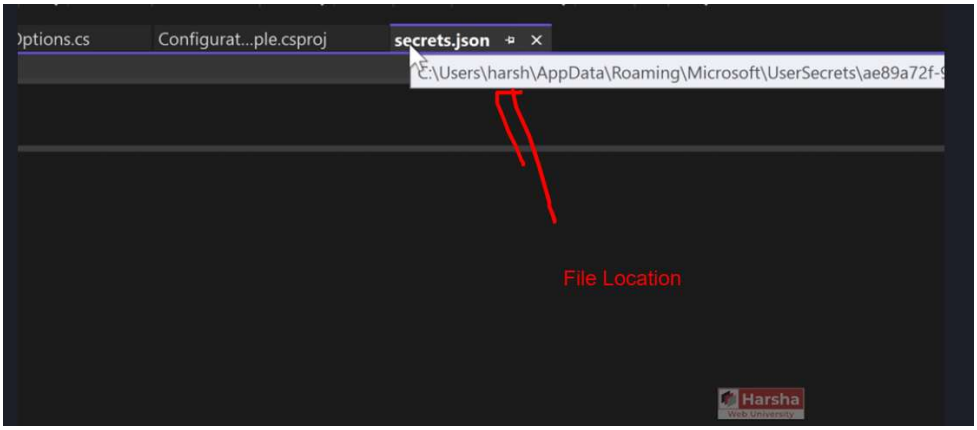


Over here, it adds user-secret id. Exactly with the same user secret Id, a json file will be created.



In case if you want to see that user-secret file, Right Click on the project and select 'Manage User Secrets'

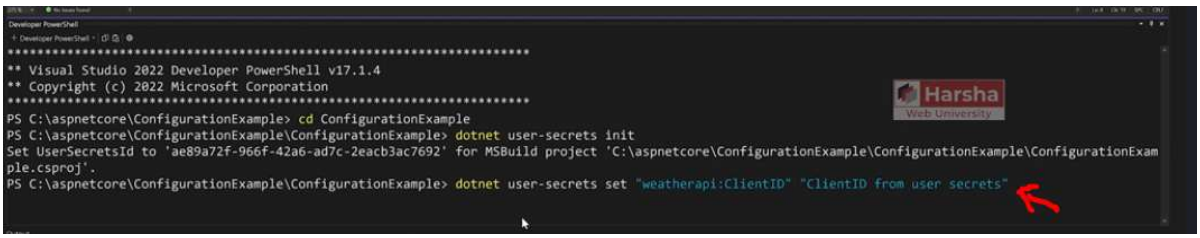


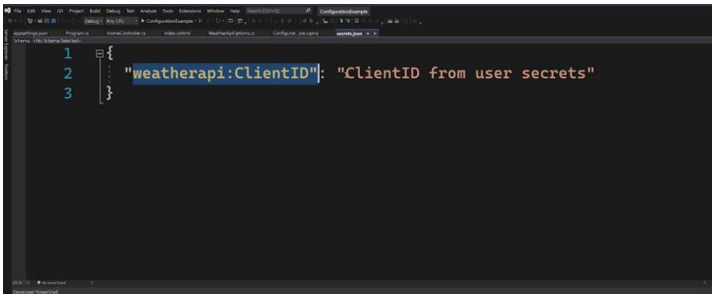


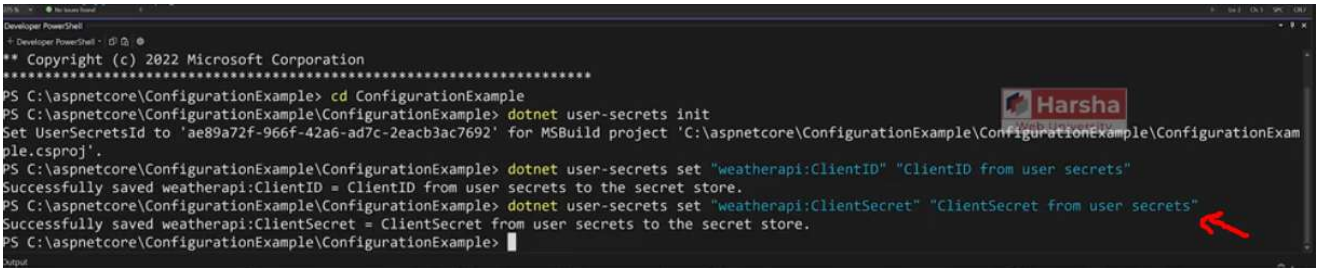
But you are not supposed to store any kind of configuration values in this JSON file directly.
You are not supposed to edit this file directly.
Just for educational purposes, we have opened this file.
Now, let's close it and come back.

We are in the developer power shell again. At the same prompt, run the second command.

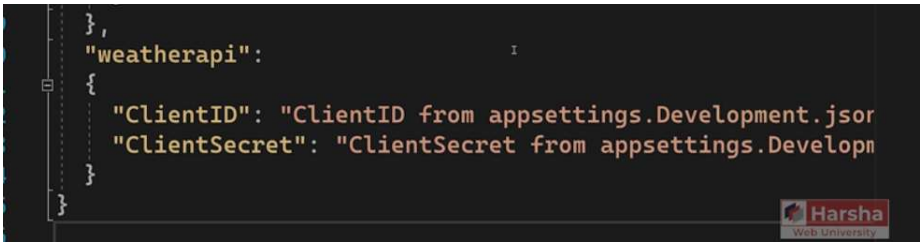
Once you press the enter key for this command, it stores the given key value pair in the user secrets you can see the message.



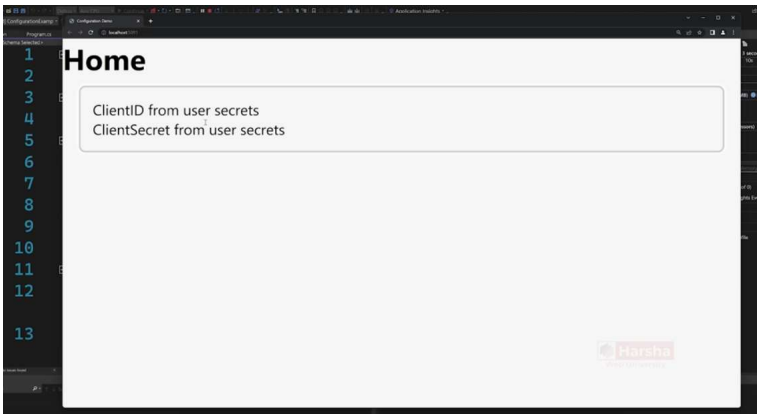




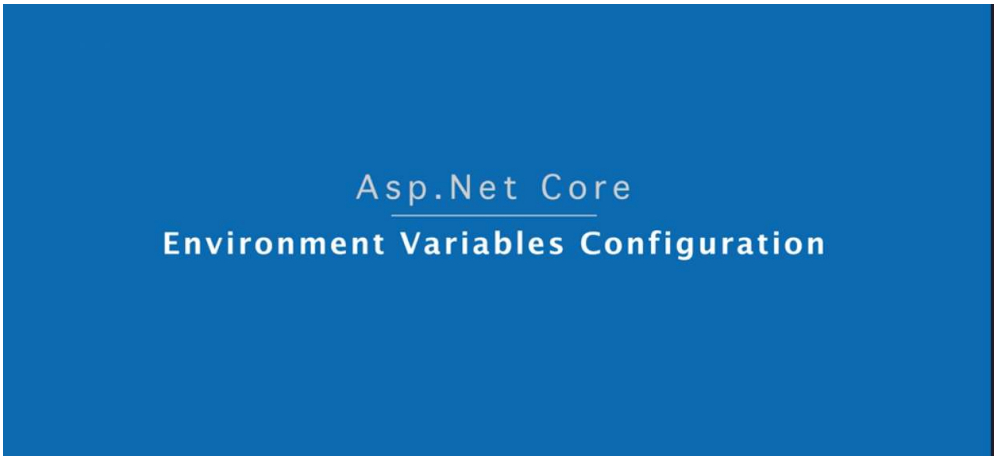
Now run our application, First it reads value from 'appSettings.Development.json' then it reads value from user-secrets.

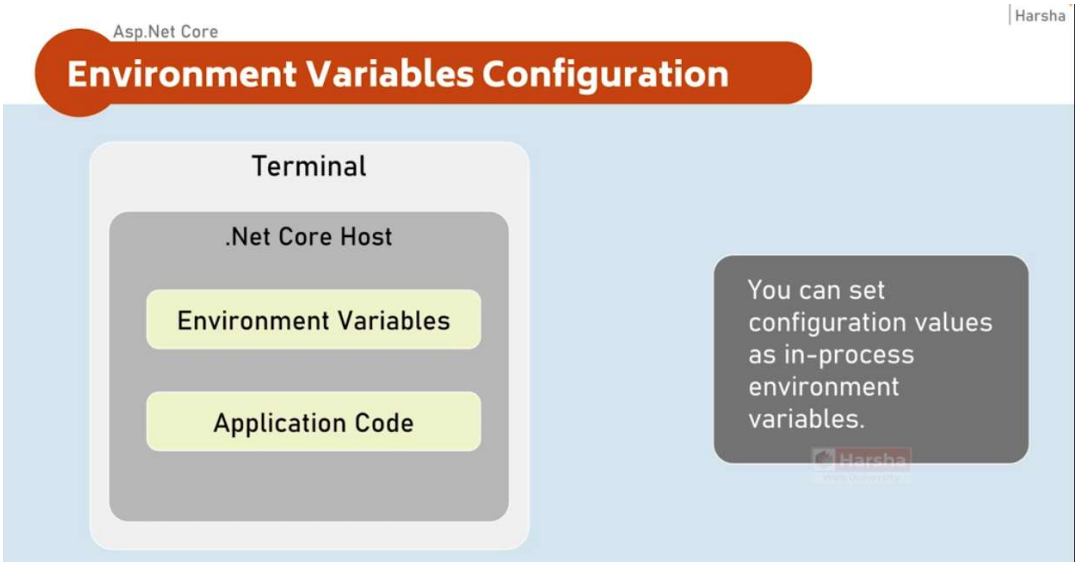


That's why it overrides all the previous value.



But if your project is being maintained on the cloud in Microsoft Azure, you can use the Azure Key Vault for better security.





So it is not safe to store sensitive values as configuration in the source code. If you store sensitive data in the configuration settings in the application source code, such as in the appsettings.json file, it can be published into the application repository. This makes it visible to all other developers and anyone who can access the repository, which is not safe.

As an alternative, in the development environment, you can use **User Secrets**, which are accessible only within the same development machine and are not visible or accessible by other developers. In this way, you separate your configuration settings from your application source code.

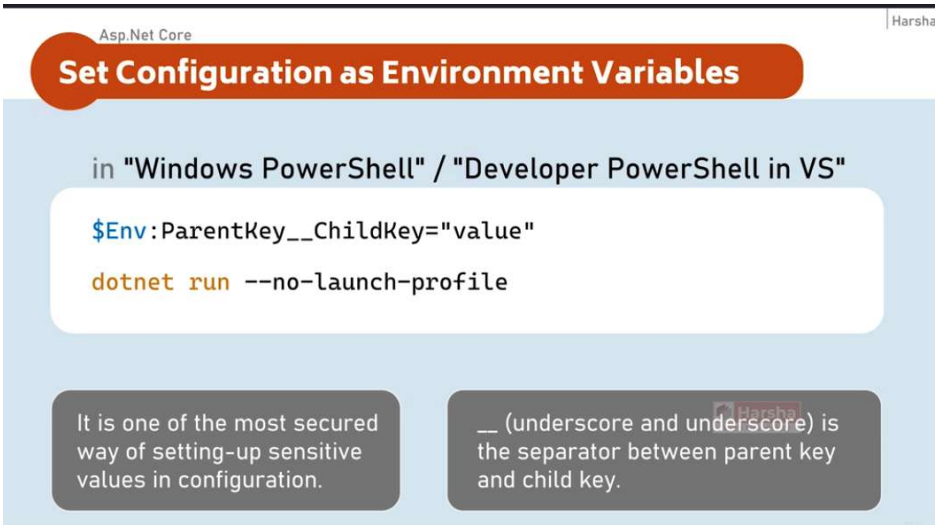
However, the concept of User Secrets works only in the development environment and should not be used in production. On a production server, you won't have Visual Studio or other development tools. You typically have access only to the terminal.

In such cases, you can store sensitive data as **environment variables** on the production server. Environment variables set in the terminal window are accessible only within the same terminal window and not from others on the same machine. This is one of the secure ways to store configuration settings as environment variables in a production environment.

For this case, you have to use your regular Windows PowerShell or the Developer PowerShell in Visual Studio. In either of these windows, you can enter this command: $env (for environment), followed by a colon, then mention your key and set it equal to a value like this:

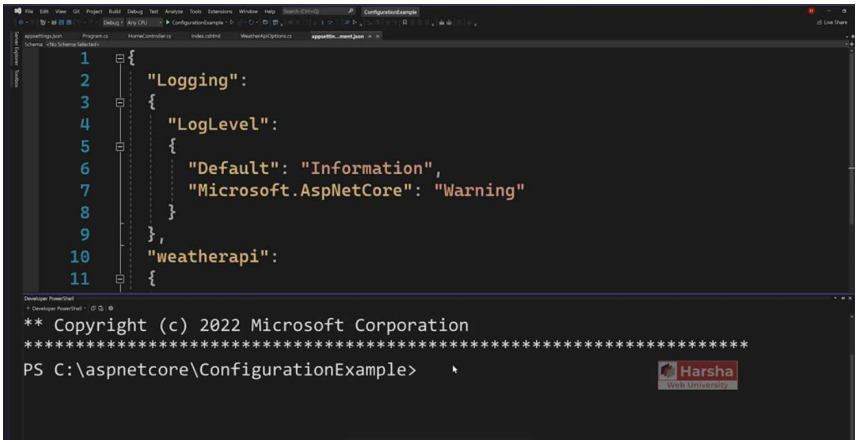
$env:KEY = "value"

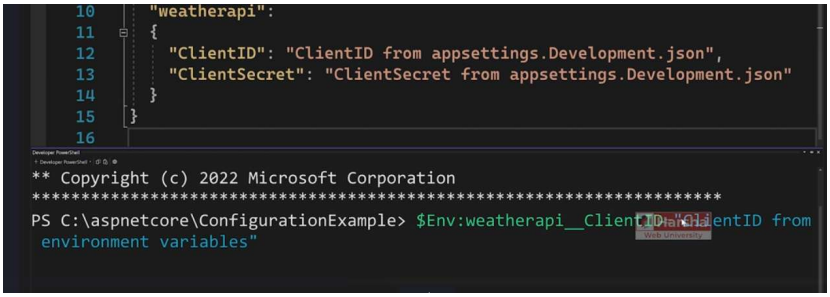
This key-value pair is called an **environment variable** and is accessible from the application running in the same terminal window. Here, __ (double underscores) can be used as the separator between the keys.



For Example, in our application the parent key is 'weather api' and the child key is 'ClientID' and 'ClientSecret' .

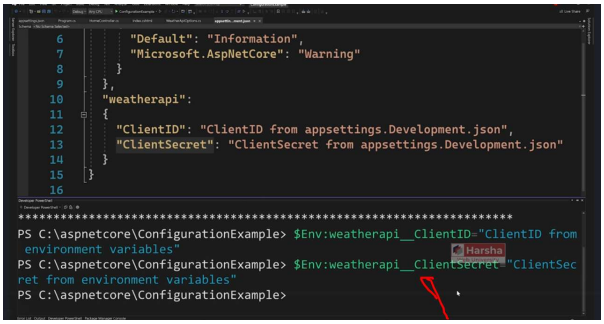
Open terminal at your project folder. Not Solution folder.



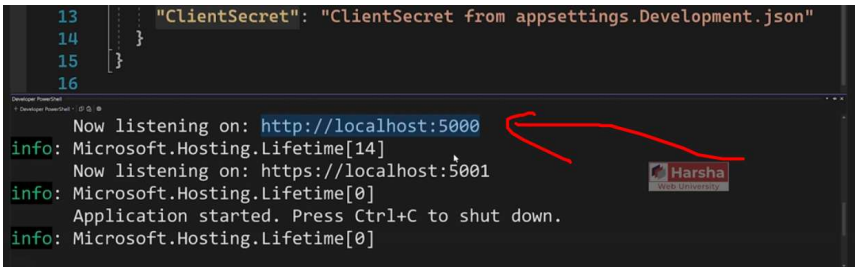


Within the same powershell window, you can access the same environment value through code. These environment variables are not accessible from other powershell windows even on the same machine. This is why these environment variables are called as **process level environments.**

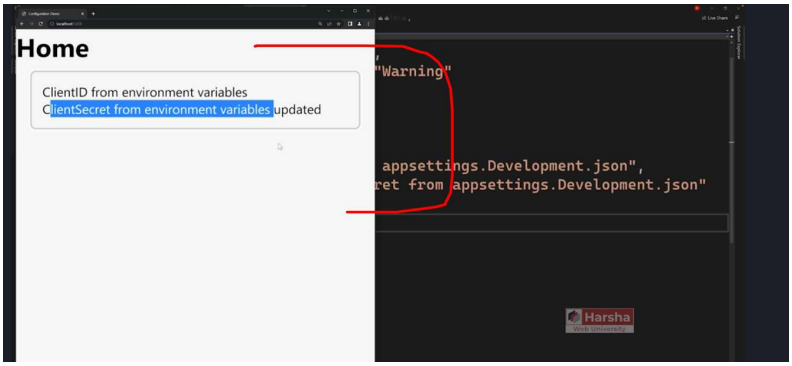
Let me try to set another one.



Now we can start our application but we don't want to launch our application from 'launchSettings.json' file.

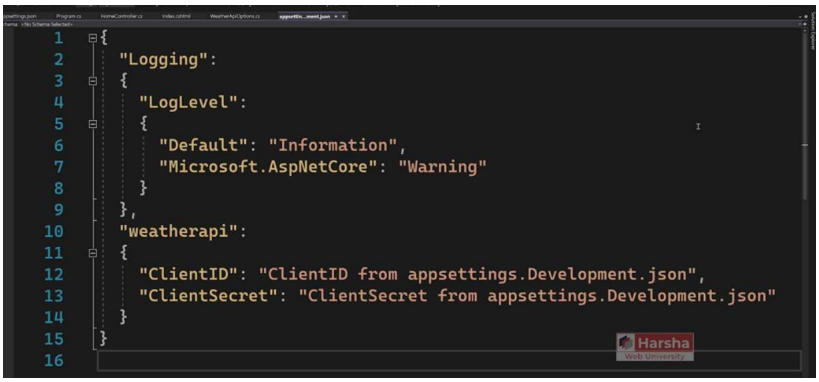


Open up the browser.



So when you want to run the application other than development, may be for production server or staging environment, you can set your configuration settings as environment variables.

appSettings.json file



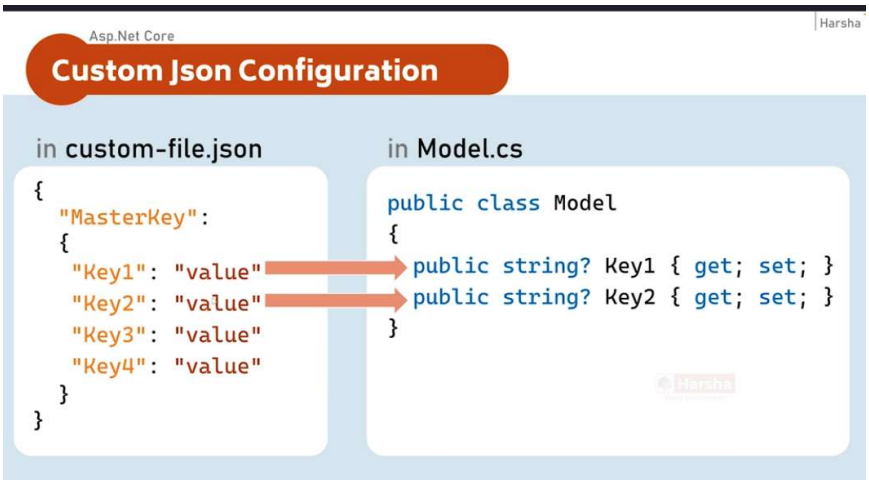
In this context, assume that you would like to store non-sensitive data as configuration within the configuration files as part of the source code. For example, you would like to store some non-sensitive data in the appsettings.json file.

However, if the number of settings increases, there can be too many configuration settings written in the same file. This makes it difficult to differentiate between the configuration settings.

As an alternative, you can group common or related configuration settings into a separate JSON file. This means you will create another JSON file, other than the appsettings.json file, and store the related configuration settings in that JSON file.

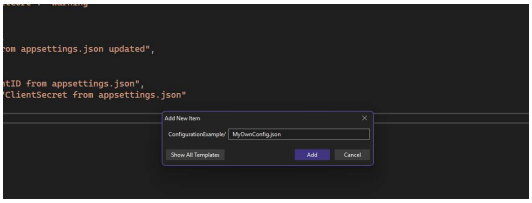
In this way, you can separate related configuration settings into a dedicated JSON file.

So you have a custom Json file with key value pairs as configuration and you can read those settings into your options class.

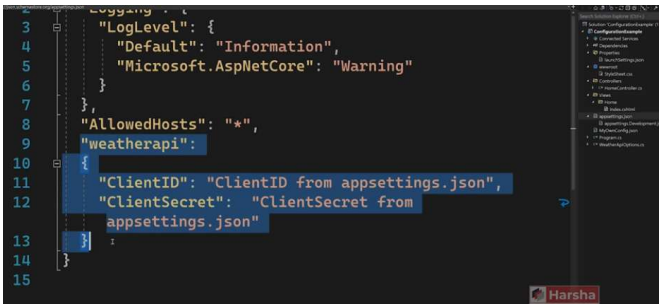


Assume that I don't want to store this weather api options within this configuration files. But I would like to store them in a separate json file as a custom json file.

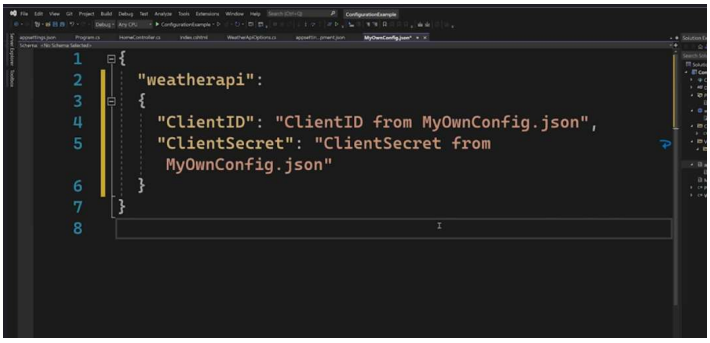
Click on your project and create your json file



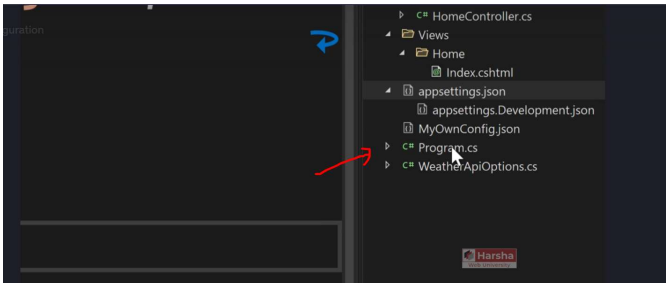
Cut this weather api from appSettings.josn



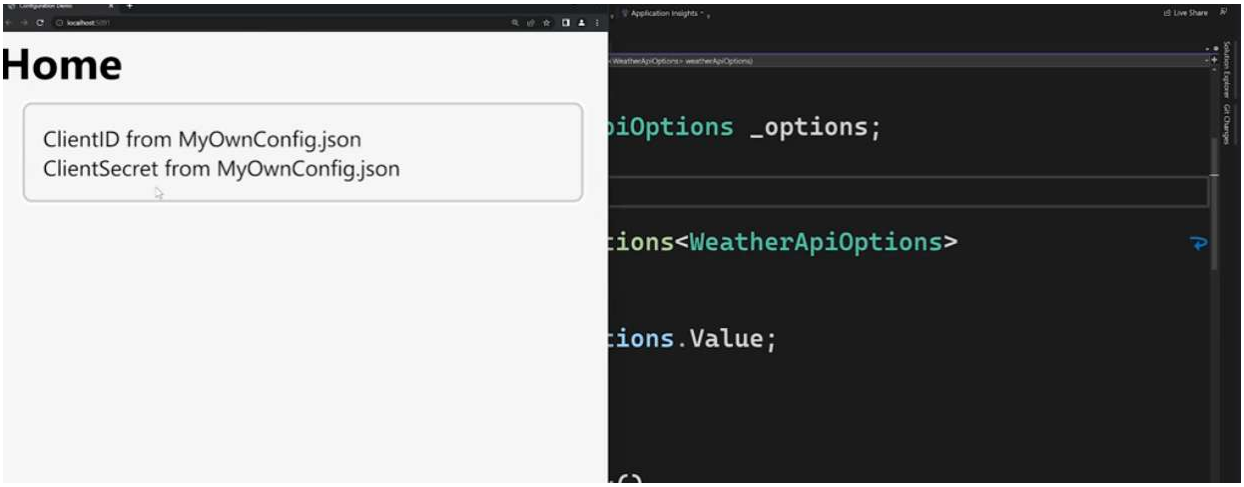
And paste in into 'myOwnConfig.json' file.

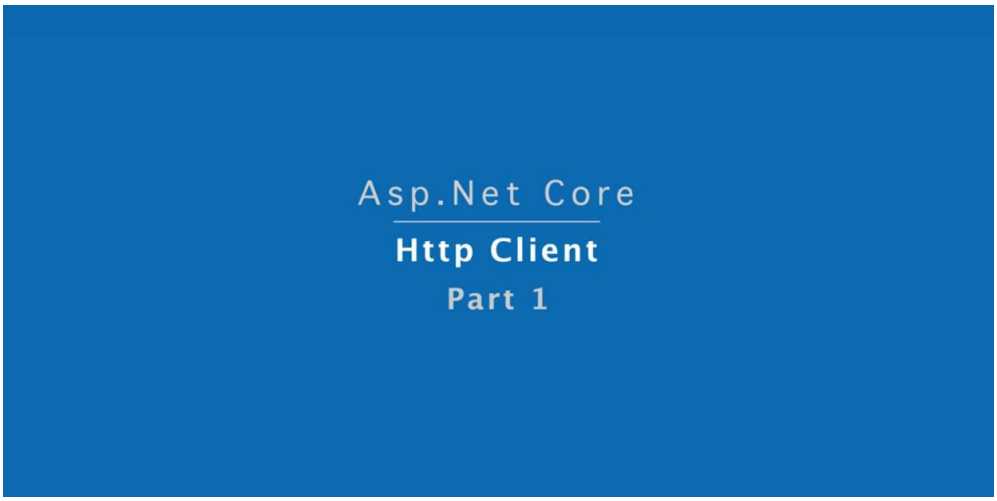


Now we have to add 'myOwnConfig.json' as configuration source through this program.cs file




```
using ConfigurationExample;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();

//We have added an options object as a service.
builder.Services.Configure<WeatherApiOptions>(builder.Configuration.GetSection("WeatherApi"));

//Load MyOwnConfig.json
builder.Host.ConfigureAppConfiguration((hostingContext, config) =>
{
    //Optional: true (means at runtime even if the json file is missing, it will not throw an exception)
    //reloadOnChange: true (means if the json file is changed, it will restart the application)
    config.AddJsonFile("MyOwnConfig.json", optional: true, reloadOnChange: true);
});

var app = builder.Build();

app.UseStaticFiles();
app.UseRouting();
app.MapControllers();

app.Run();
```



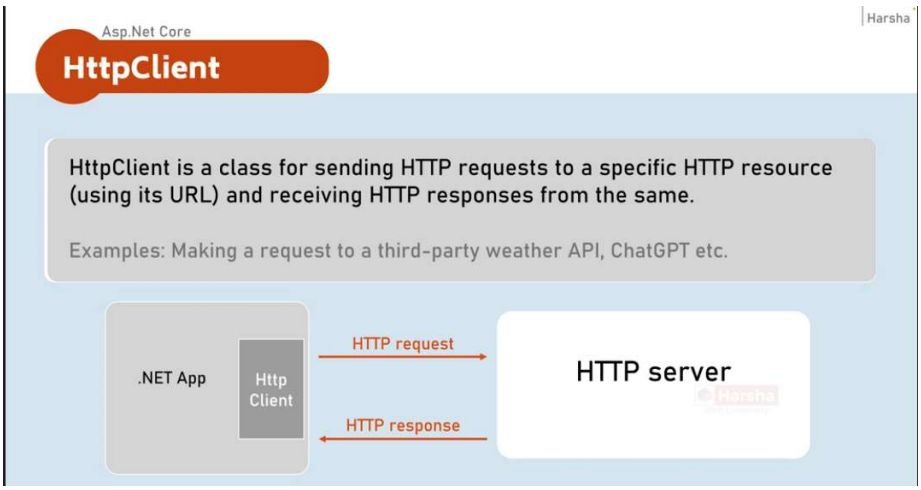


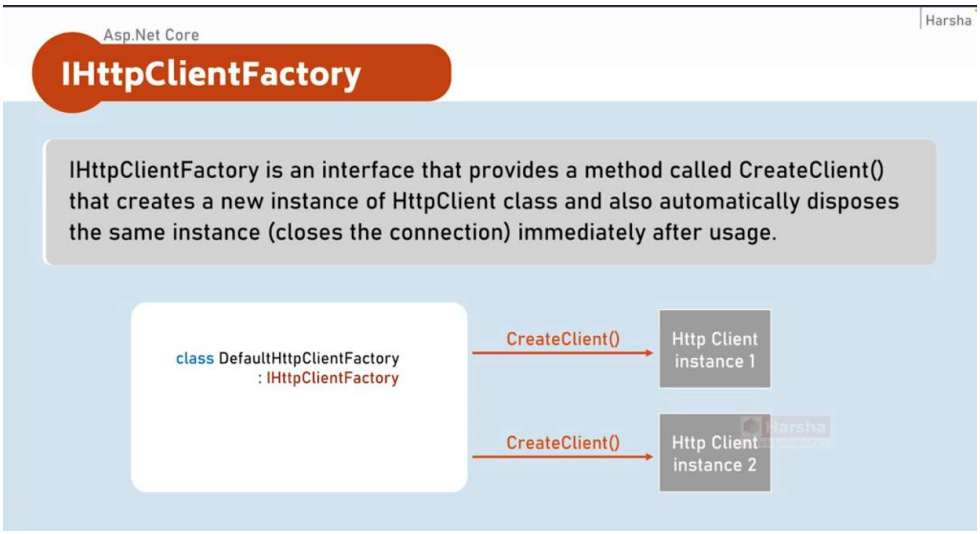
When you want to connect to external RESTful web services from your ASP.NET Core web application, you have a predefined class called HttpClient.

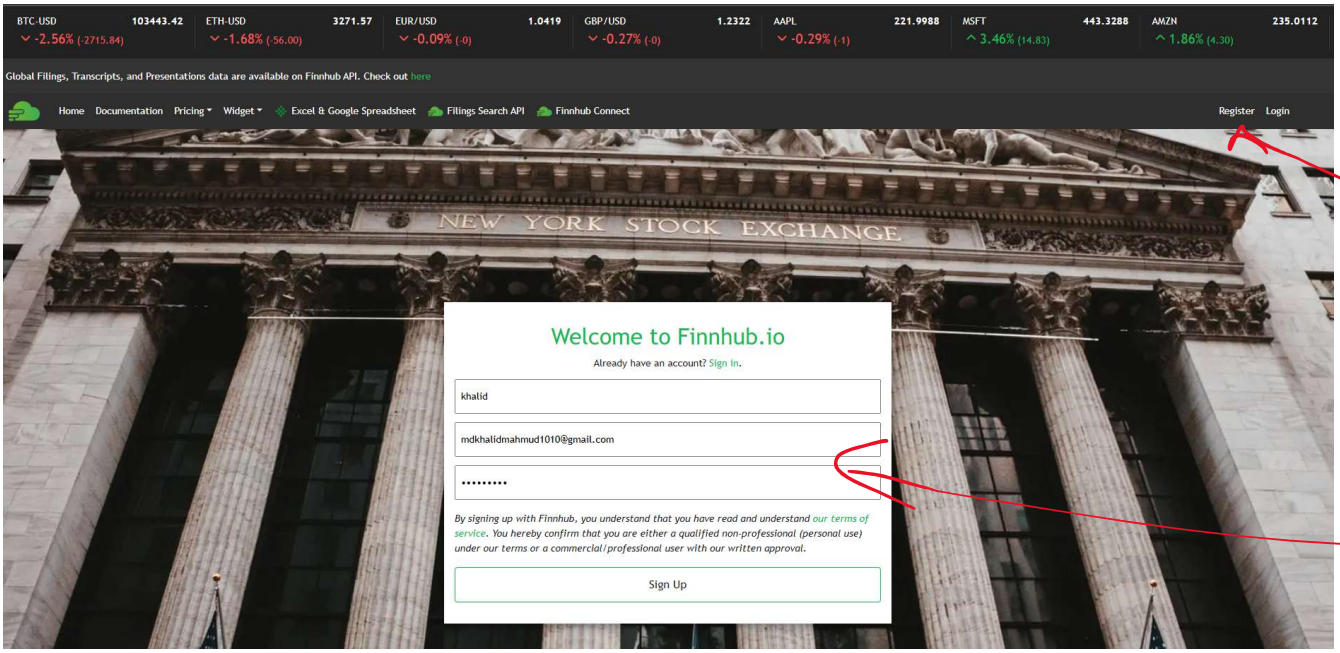
HttpClient is a predefined class in the System.Net.Http namespace, used to send HTTP requests to a server and receive responses from it. For example, when you need to connect to a third-party or external web service, you might want to make HTTP GET, POST, PUT, or DELETE requests and read the responses.

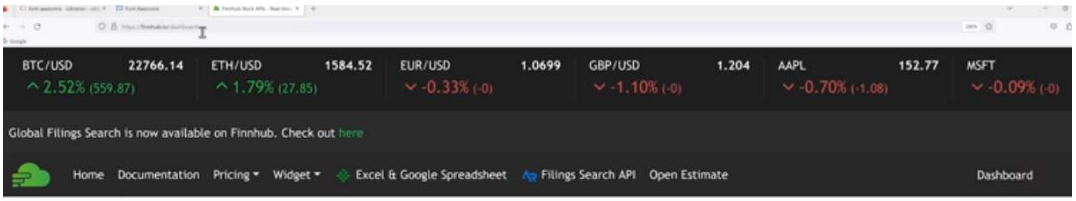
In scenarios where you need to connect to external web services to retrieve information not available in your local database, HttpClient is useful. For example, you might need to get the latest weather information, stock market data, news, tweets, or any other external information.

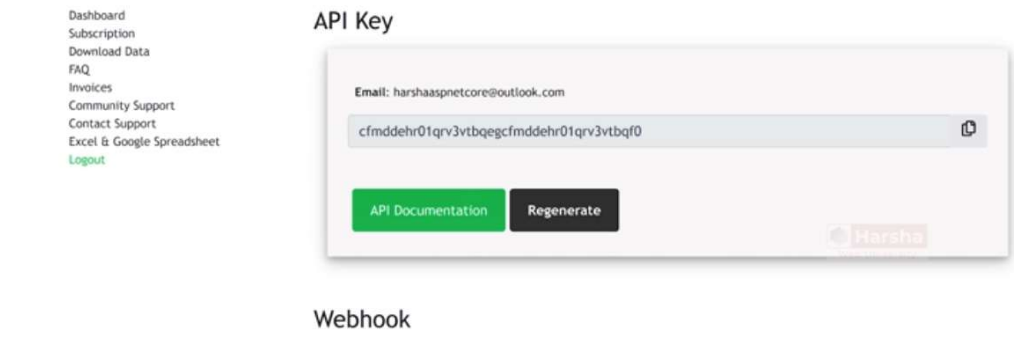
In such cases, you use the HttpClient class. Your ASP.NET Core application, particularly the server-side code, acts as a client. This means the browser sends a request to the server, and your .NET application running on the server acts as a client, creating an object of the HttpClient class to send an HTTP request to an external web service and receive the response.
Let me demonstrate this practically.




Harsha
Asp.Net Core
IHttpClientFactory
IHttpClientFactory is an interface that provides a method called CreateClient() that creates a new instance of HttpClient class and also automatically disposes the same instance (closes the connection) immediately after usage.
class DefaultHttpClientFactory
: IHttpClientFactory
CreateClient()
Http Client instance 1
CreateClient()
Http Client instance 2

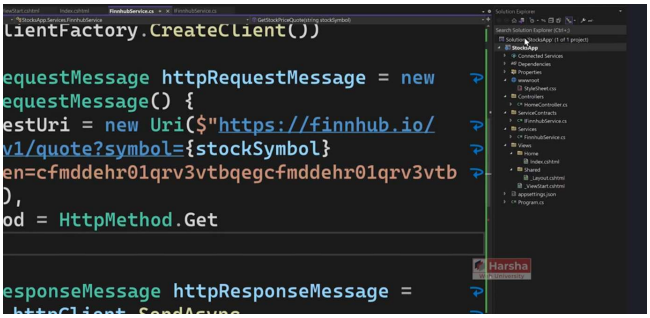

BTC-USD 103443.42 -2.56% (-2715.84)
ETH-USD 3271.57 -1.68% (-56.00)
EUR/USD 1.0419 -0.09% (-0)
GBP/USD 1.2322 -0.27% (-0)
AAPL 221.9988 -0.29% (-1)
MSFT 443.3288 3.46% (14.83)
AMZN 235.0112 1.86% (4.30)
Global Filings, Transcripts, and Presentations data are available on Finnhub API. Check out here
Home Documentation Pricing Widget Excel & Google Spreadsheet Filings Search API Finnhub Connect
Register Login
NEW YORK STOCK EXCHANGE
Welcome to Finnhub.io
Already have an account? Sign In.
khalid
mdkhalidmahmud1010@gmail.com
By signing up with Finnhub, you understand that you have read and understand our terms of service. You hereby confirm that you are either a qualified non-professional (personal use) under our terms or a commercial/professional user with our written approval.
Sign Up

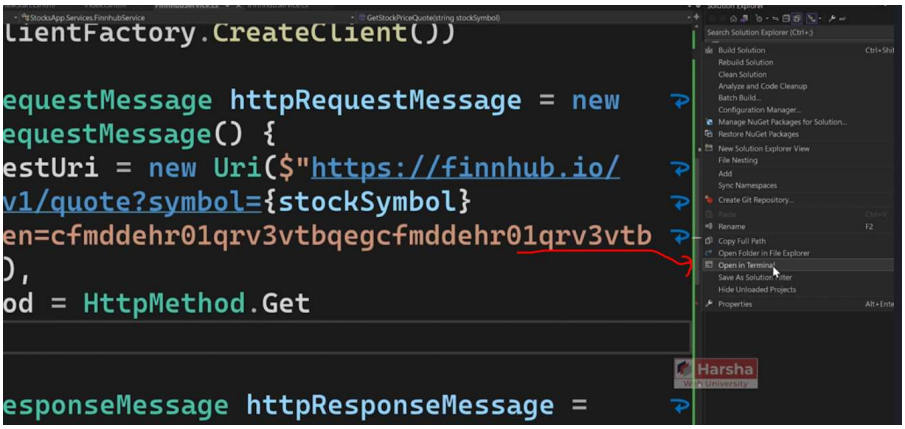

BTC/USD 22766.14 2.52% (559.87)
ETH/USD 1584.52 1.79% (27.85)
EUR/USD 1.0699 -0.33% (-0)
GBP/USD 1.204 -1.10% (-0)
AAPL 152.77 -0.70% (-1.08)
MSFT -0.09% (-0)
Global Filings Search is now available on Finnhub. Check out here
Home Documentation Pricing Widget Excel & Google Spreadsheet Filings Search API Open Estimate
Dashboard


Dashboard
Subscription
Download Data
FAQ
Invoices
Community Support
Contact Support
Excel & Google Spreadsheet
Logout
API Key
Email: harshaaspnetcore@outlook.com
cfmddehr01qrv3vtbqegcfmddehr01qrv3vtbqf0
API Documentation
Regenerate
Webhook

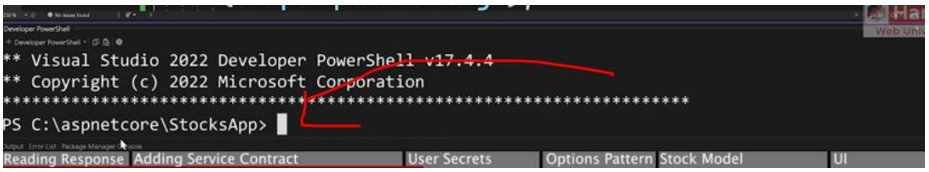
But it is not safe to store 'user-secrets to the code. User Secrets is individual for every developer machine. So, let's enable user-secrets in this application.

In order to enable 'user-secrets', right click on the solution

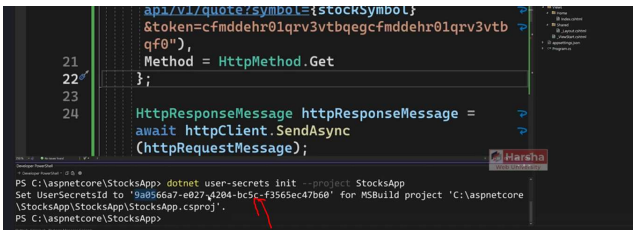




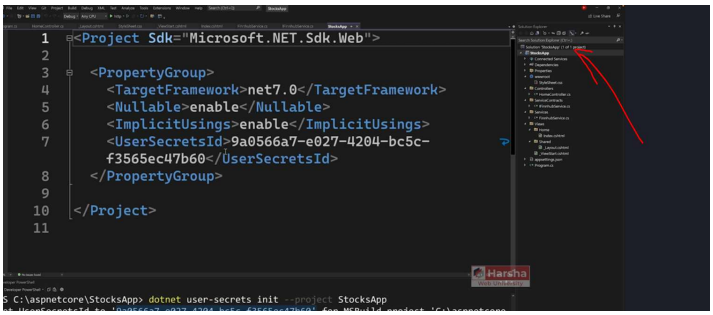
Make sure you are in the project folder.

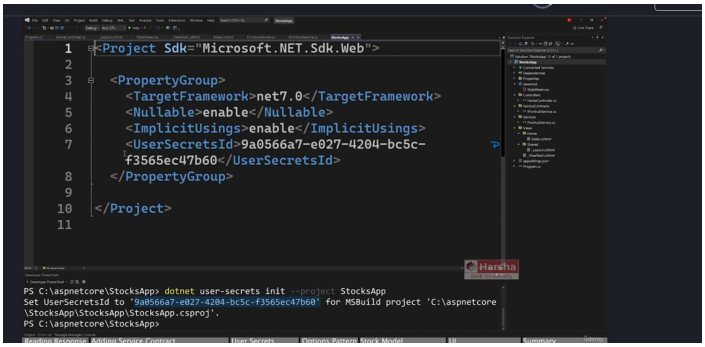


This user secrets id will be stored in the .csproj file.

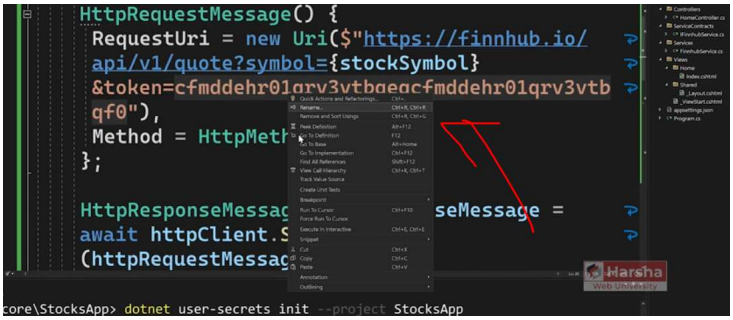


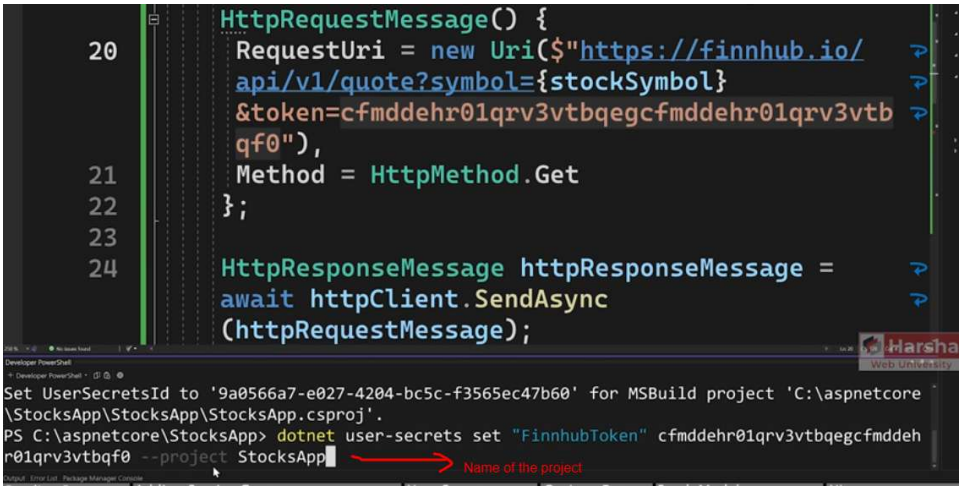
If you double click on the project, then you can see it

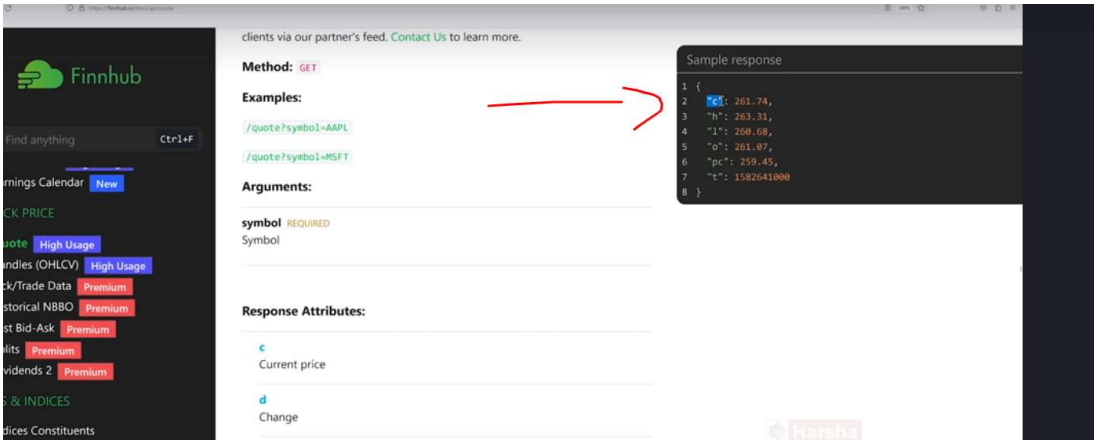


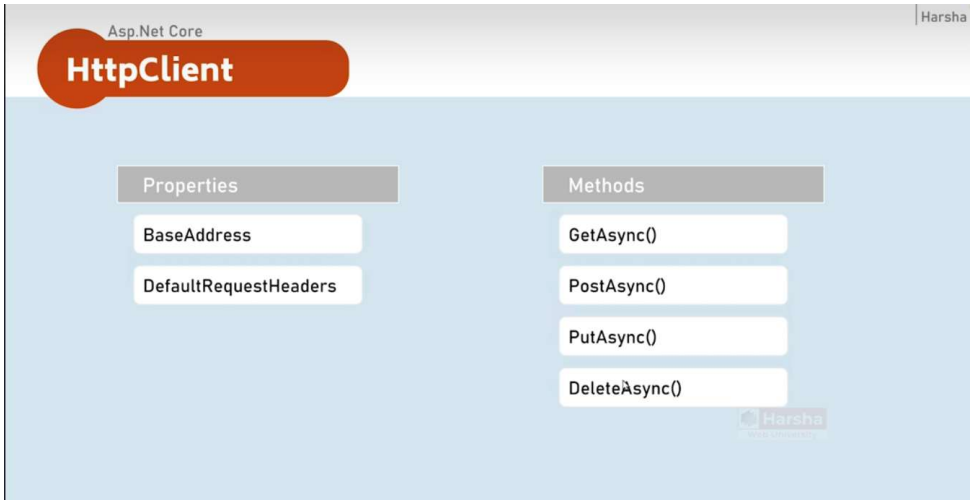


Copy this token which we have got from the 'finhub site'






Finnhub
Find anything
Ctrl+F
New
High Usage
High Usage
Premium
Premium
Premium
Premium
Premium
& INDICES
clients via our partner's feed. Contact Us to learn more.
Method: GET
Examples:
/quote?symbol=AAPL
/quote?symbol=MSFT
Arguments:
symbol REQUIRED
Symbol
Response Attributes:
c
Current price
d
Change
Sample response
{
"c": 261.74,
"h": 263.31,
"l": 260.68,
"o": 261.07,
"pc": 259.45,
"t": 1582641000
}
Harsha


Asp.Net Core
HttpClient
Harsha
Properties
BaseAddress
DefaultRequestHeaders
Methods
GetAsync()
PostAsync()
PutAsync()
DeleteAsync()
Harsha