tutorialr.com

.NET Notes MCP

Contents

Setup & Start ... 2
- Install .NET SDK ... 2
- Install Template ... 5
- Create Project ... 6
- Install Package ... 7
- Launch Project ... 8
- Create GitHub Account ... 9
- Enable GitHub Copilot ... 11
- Install Visual Studio Code ... 14
- Launch Visual Studio Code ... 16

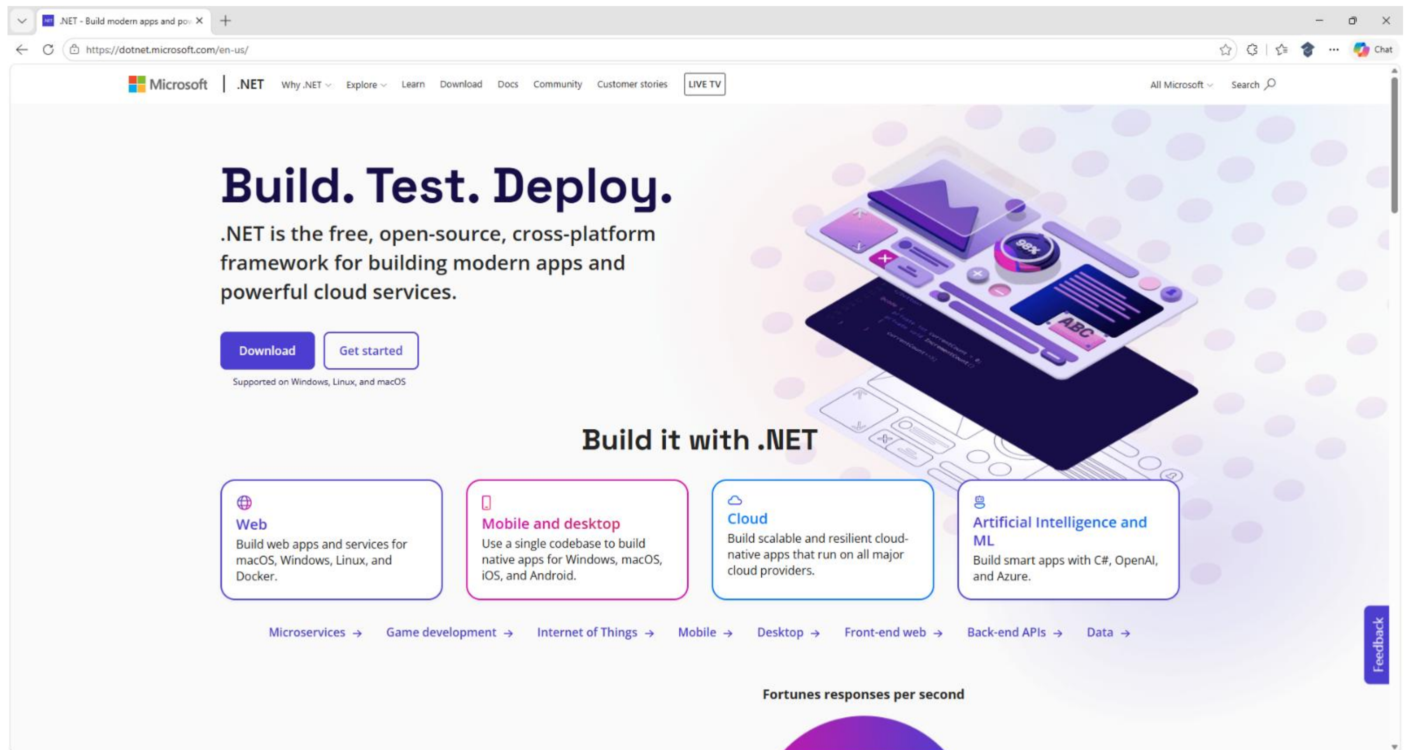
Implement ... 18
- Note Class ... 18
- Provider Class ... 21
- Service Class ... 27
- Tools Class ... 36
- Update Project ... 40
- Configure Project ... 44

Integrate ... 47
- Login GitHub Copilot ... 47
- Configure GitHub Copilot ... 50
- Demonstrate GitHub Copilot ... 52

Setup & Start

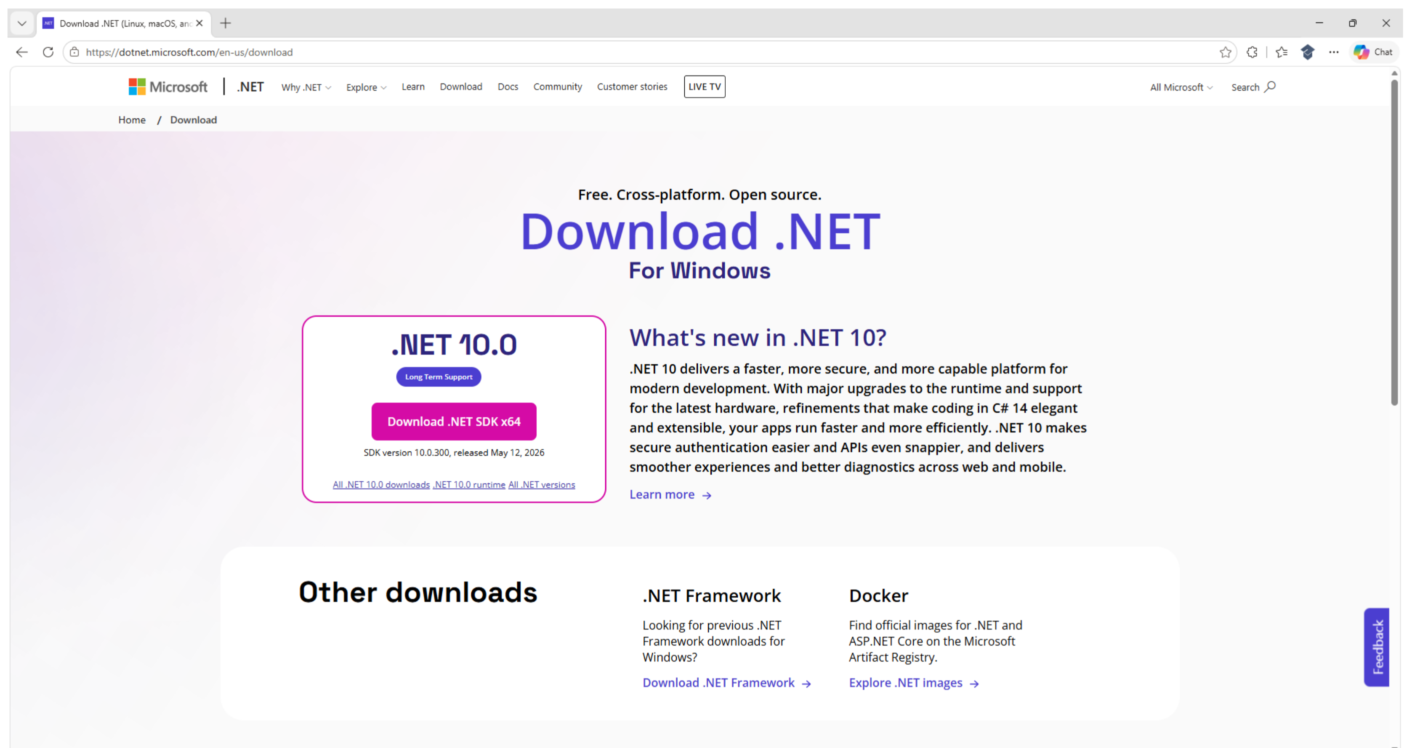
Install .NET SDK

First, you need to **Download** the latest **.NET SDK**, to do so use a **Browser** and visit the **Website** at dot.net.



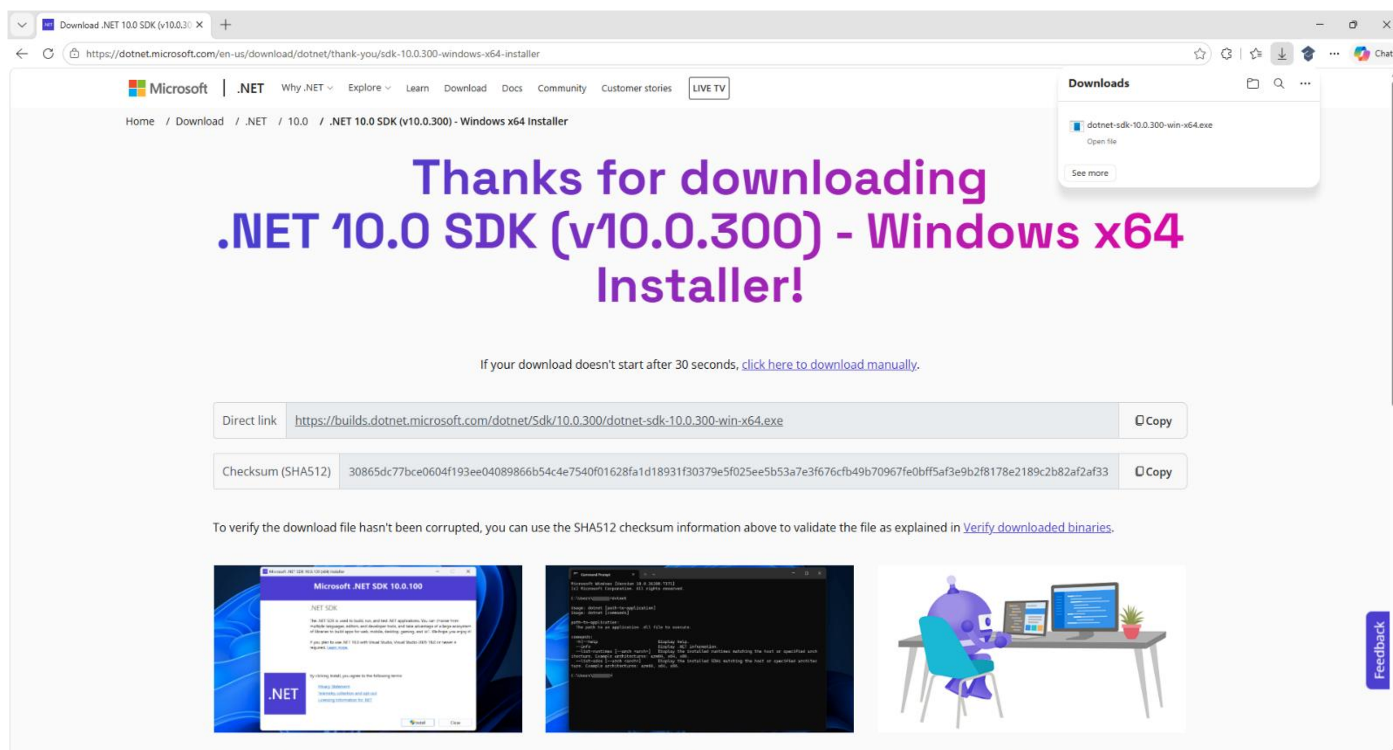
Information - **.NET** is the free, open-source cross-platform framework for building modern applications and powerful services from **Microsoft** including intelligent experiences using or working with **AI**.

Then choose **Download** which should display the **.NET SDK** for your platform of **Windows** or **Mac**.



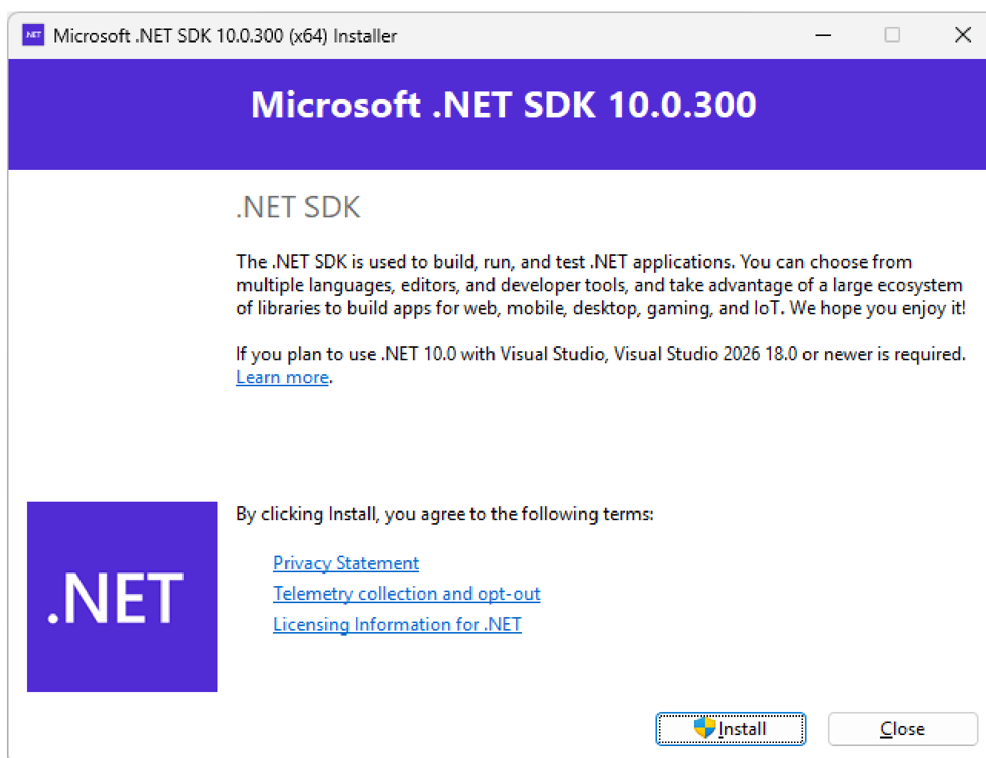
CC BY SA

Next select **Download** for the .**NET SDK** for **.NET 10.0** which varies based on your exact platform of **Mac** or **Windows** for example **Download .NET SDK x64** although your exact **Version** may be different or newer.

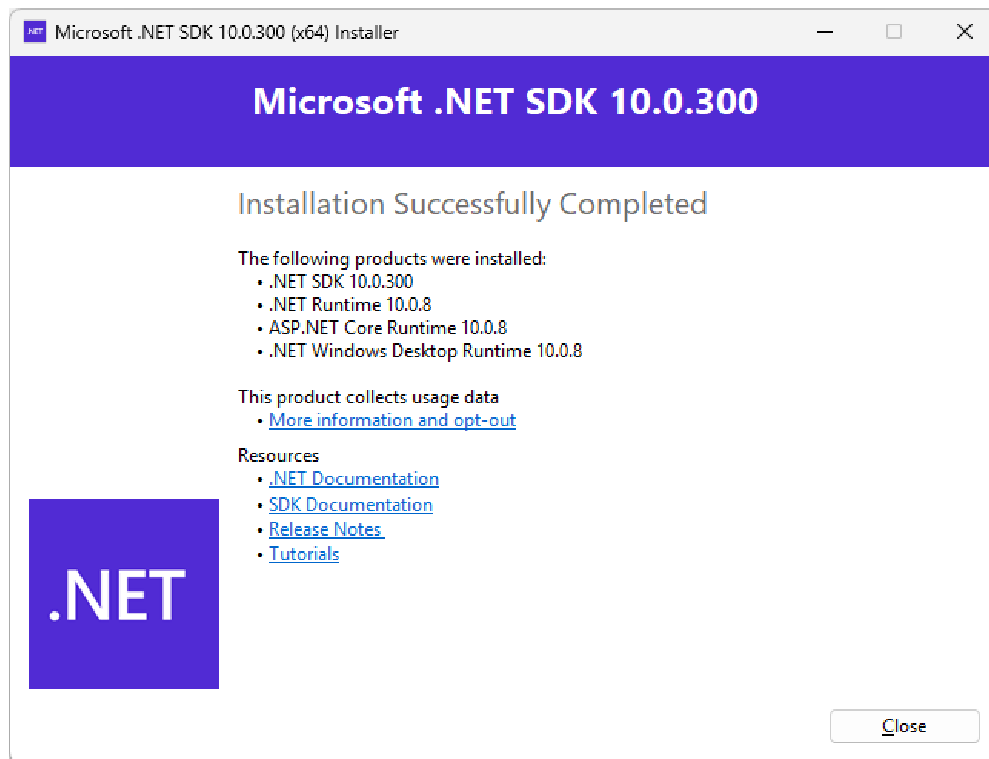


Information - **.NET 10.0 SDK** here is *v10.0.300* for *Windows x64* which was version used in this **Workshop**.

Then the **Installer** for **.NET SDK** will begin **Downloading** and once it has been **Downloaded** it will show in **Downloads** for your **Browser** where you can **Open** it to launch the **Installer** for **.NET SDK** as follows:



After **Opening** the **Installer** for the **.NET SDK** select **Install** to begin the installation process for **.NET SDK**.

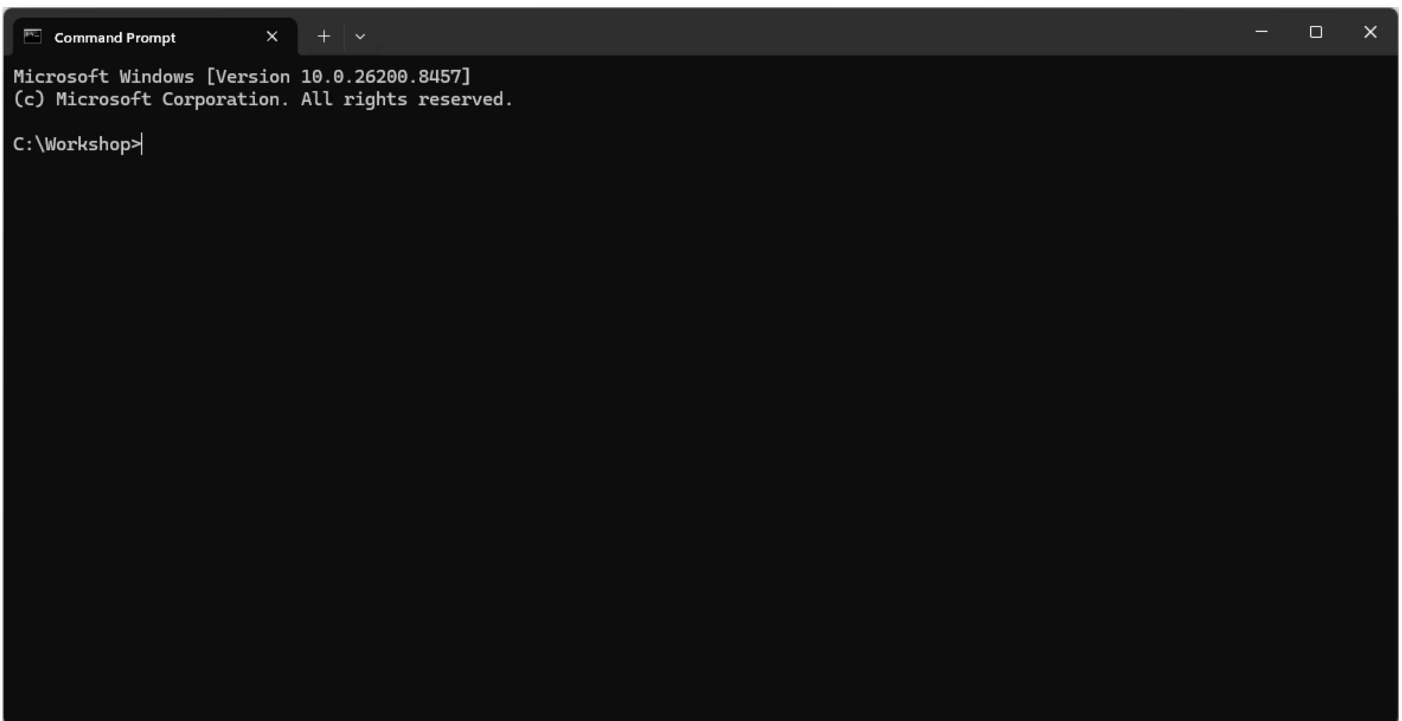


Finally, when installation has completed for the **.NET SDK**, you can select **Close** in the **Installer** as this completes the process of **Downloading** and **Installing** the **.NET SDK** for **Windows** or **Mac**.

Information - **.NET SDK** has a variety of built-in **Templates** you can use to create many different types of application such as modern applications for the web with **Blazor** and **ASP.NET Core**, for mobile or cross-platform with **.NET MAUI**, the desktop for **Console** applications, cloud with **Aspire** and many more.

Install Template

Once **.NET SDK** has been **Installed** you will need to install the **Template** to create **Projects** for **MCP**, to do this, if using **Mac** you need to go to **Finder**, search for **Terminal** and then select it to **Open** it, or if using **Windows** you need to go to **Start**, search for **Command Prompt** and then select it to **Open** as follows:



Information – You can choose a different location for the **Project** in **Terminal** on **Mac** or **Command Prompt** on **Windows** if needed, but the default location should be fine for the **Workshop**.

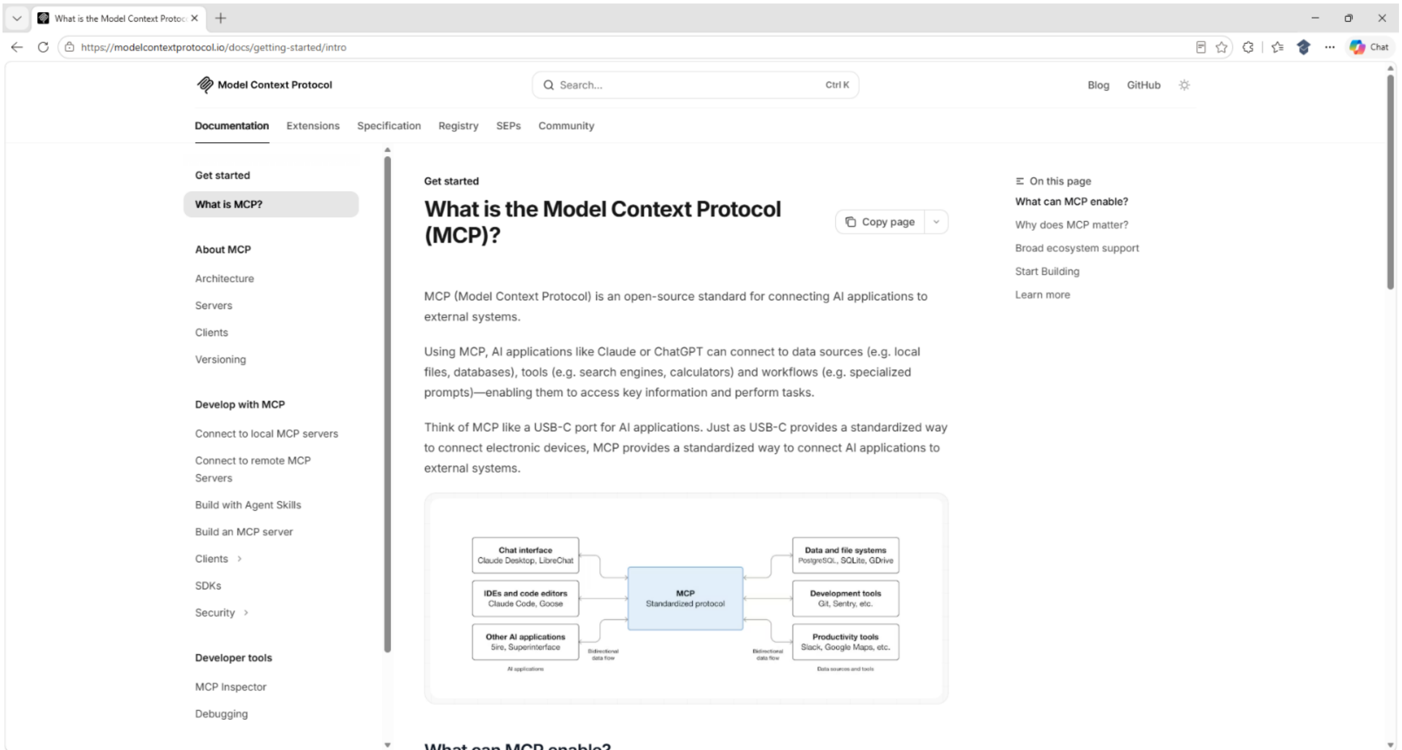
Then you will need to install the **Template** to allow you to create a **Server** for **MCP**, to do this from the **Terminal** on **Mac** or **Command Prompt** on **Windows** you need to *Copy* and *Paste* the following **Command** then press **Enter**:

```
dotnet new install Microsoft.McpServer.ProjectTemplates
```

Information – This will install the **Template** to be able to create **Projects** for **MCP Server** contained within **Microsoft.McpServer.ProjectTemplates** in addition to the existing **Templates** built into the **.NET SDK**.

Don't **Close** the **Terminal** on **Mac** or **Command Prompt** on **Windows** as need it for the entire **Workshop**.

Create Project



Information – **Model Context Protocol** or **MCP** is the standard for connecting **AI** applications to external systems such as data, workflows and tools. **MCP** reduces time and complexity when building or working with **Agents** using **AI**. **MCP** enables a wider ecosystem with **Servers** that can be consumed with **Clients** such as **GitHub Copilot**. To find out more about **MCP** you can visit modelcontextprotocol.io.

Then you will need to create the **Project** using the **.NET SDK**, to do this from the **Terminal** on **Mac** or **Command Prompt** on **Windows** you need to *Copy* and *Paste* the following **Command** then press **Enter**:

```
dotnet new mcpserver --transport remote -o DotNet.Notes.Mcp
```

Information – This will create a **Project** for an **MCP Server App** called **DotNet.Notes.Mcp** which is a special kind of application that will work with **Agents** using **AI** or **Assistants** such as **GitHub Copilot**.

Don't **Close** the **Terminal** on **Mac** or **Command Prompt** on **Windows** as need it for the entire **Workshop**.

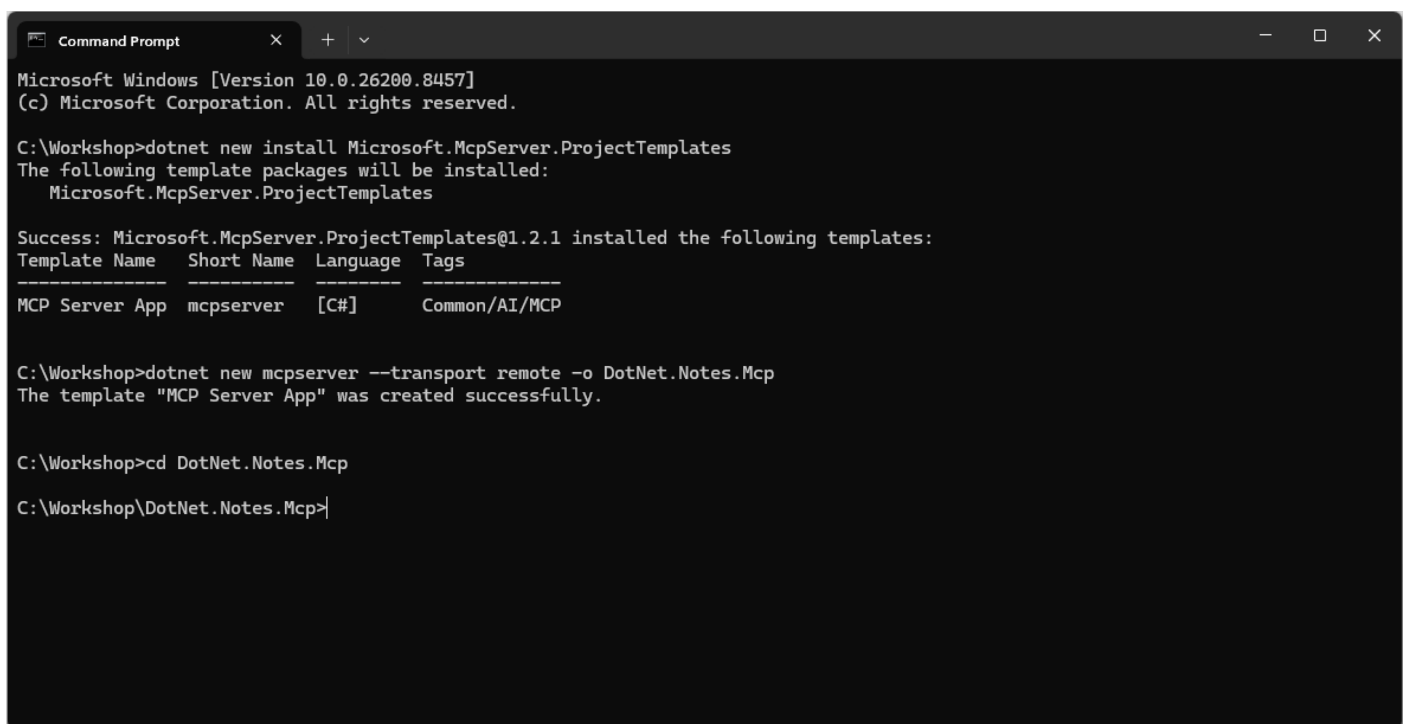
Install Package

Once the **Project** for **DotNet.Notes.Mcp** for an **MCP Server App** has been created successfully, then within the **Terminal** on **Mac** or **Command Prompt** on **Windows** you need to switch to the **Folder** for the **Project** of **DotNet.Notes.Mcp**. To do this *Copy* and *Paste* the following **Command** and then press **Enter**:

```
cd DotNet.Notes.Mcp
```

Information – The **Command** of **cd** means change directory which is common to both the **Terminal** on **Mac** and **Command Prompt** on **Windows** to switch to a specified **Folder**.

Once done you should have switched to the **Folder** for the **Project** of **DotNet.Notes.Mcp** as follows:



Next in **Terminal** on **Mac** or **Command Prompt** on **Windows** you need to add a **Package** to the **Project** for **Microsoft.Data.Sqlite**, to do so *Copy* and *Paste* the following **Command** and then press **Enter**:

```
dotnet add package Microsoft.Data.Sqlite
```

Information – This will add the **Package** for **Microsoft.Data.Sqlite** which is a is a lightweight **Package** for **SQLite**, which is a **Database** that is used to store or retrieve information, which will be used in the **Project**. To find out more about the **Package** then visit nuget.org/packages/Microsoft.Data.Sqlite.

Don't **Close** the **Terminal** on **Mac** or **Command Prompt** on **Windows** as need it for the entire **Workshop**.

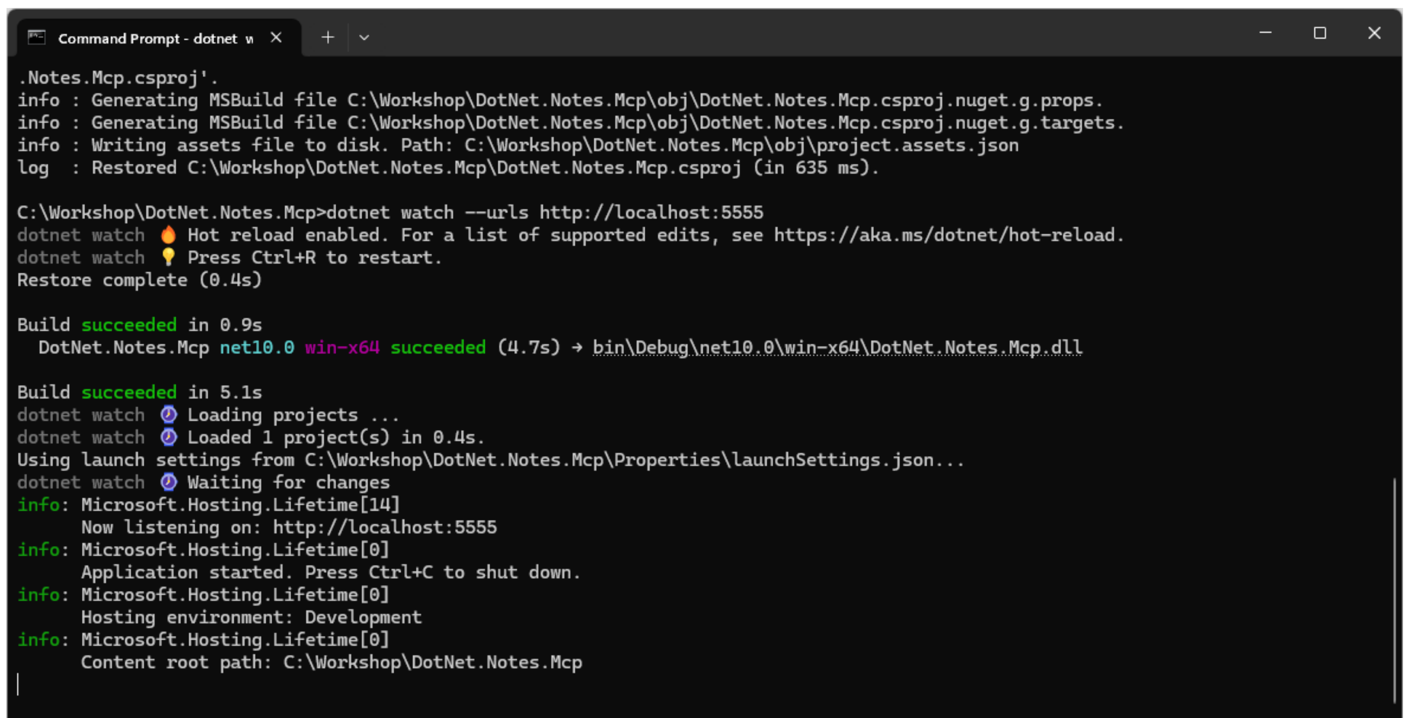
Launch Project

Then in **Terminal** on **Mac** or **Command Prompt** on **Windows** you then need to *Copy* and *Paste* the following **Command** and then press **Enter**:

```
dotnet watch --urls http://localhost:5555
```

Information – If there is any problem using **5555** then pick another reasonable number such as **5556**, just remember to match this value when it is needed later in the **Workshop**.

Once the **Command** has been performed, this will **Build** your **Project** and then start a **Watch** for changes in the **Command Prompt** on **Windows** or **Terminal** on **Mac** as follows:



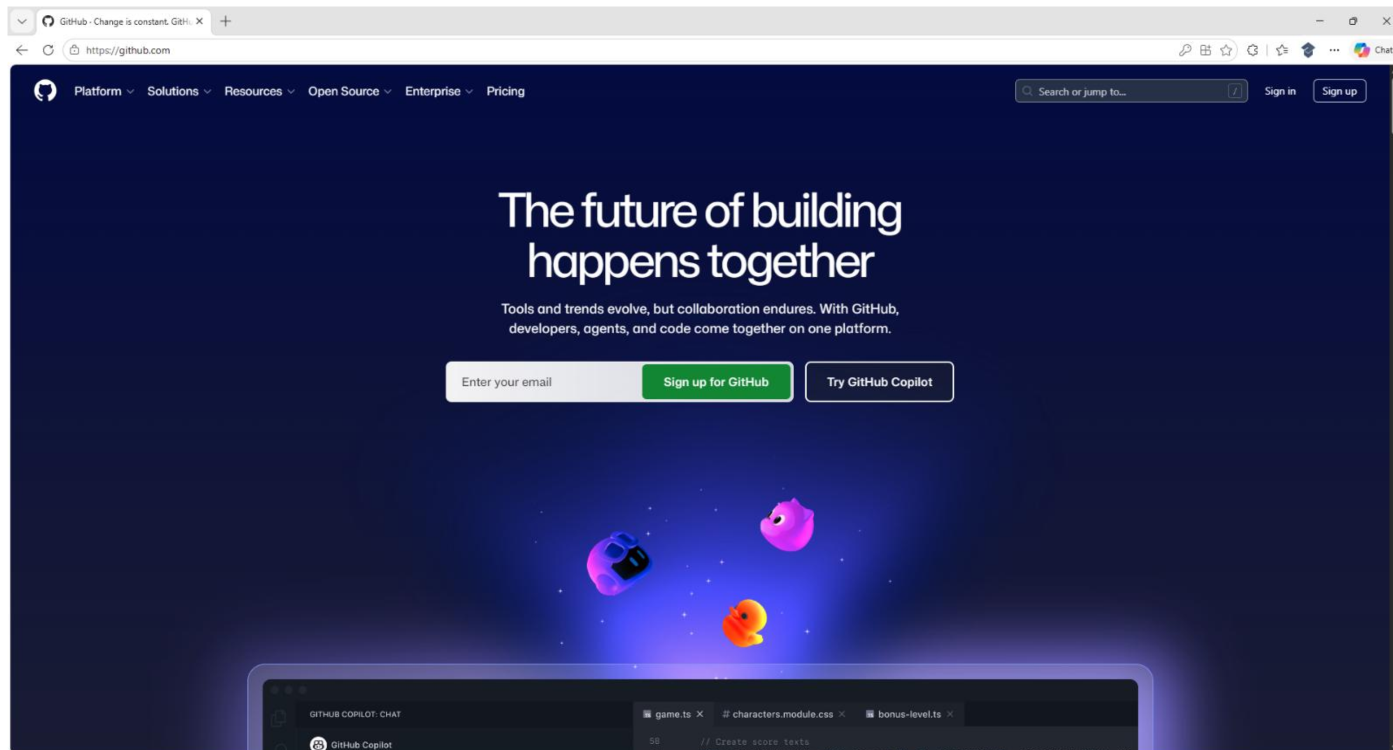
Information – If the **Terminal** on **Mac** or **Command Prompt** on **Windows** ever displays any **Errors** then make sure that everything you entered was correct from the **Workshop** by going over the **Steps** check it matches what you have, then once any corrections have been made and **Saved** and there are no **Errors** then the **Watch** and **Build** should proceed.

Don't **Close** the **Terminal** on **Mac** or **Command Prompt** on **Windows** as need it for the entire **Workshop**, but if it is **Closed** then you need to go to **Finder**, search for **Terminal** and then select it to **Open** it, or if you **Closed** the **Command Prompt** on **Windows** you need to go to **Start**, search for **Command Prompt** and then select it to **Open** it. Then once opened you need to change directory using **cd** to the location for your **Project**, for example *cd DotNet.Notes.Mcp* and then you need to type **`dotnet watch --urls http://localhost:5555`** followed by **Enter**.

Once the **Command** to **Launch** the **Project** has been performed in either **Terminal** on **Mac** or **Command Prompt** on **Windows**, this completes the process of creating the **Project** of **DotNet.Notes.Mcp**, adding the **Package** for **Microsoft.Data.Sqlite** and **Launch** of the **Project**.

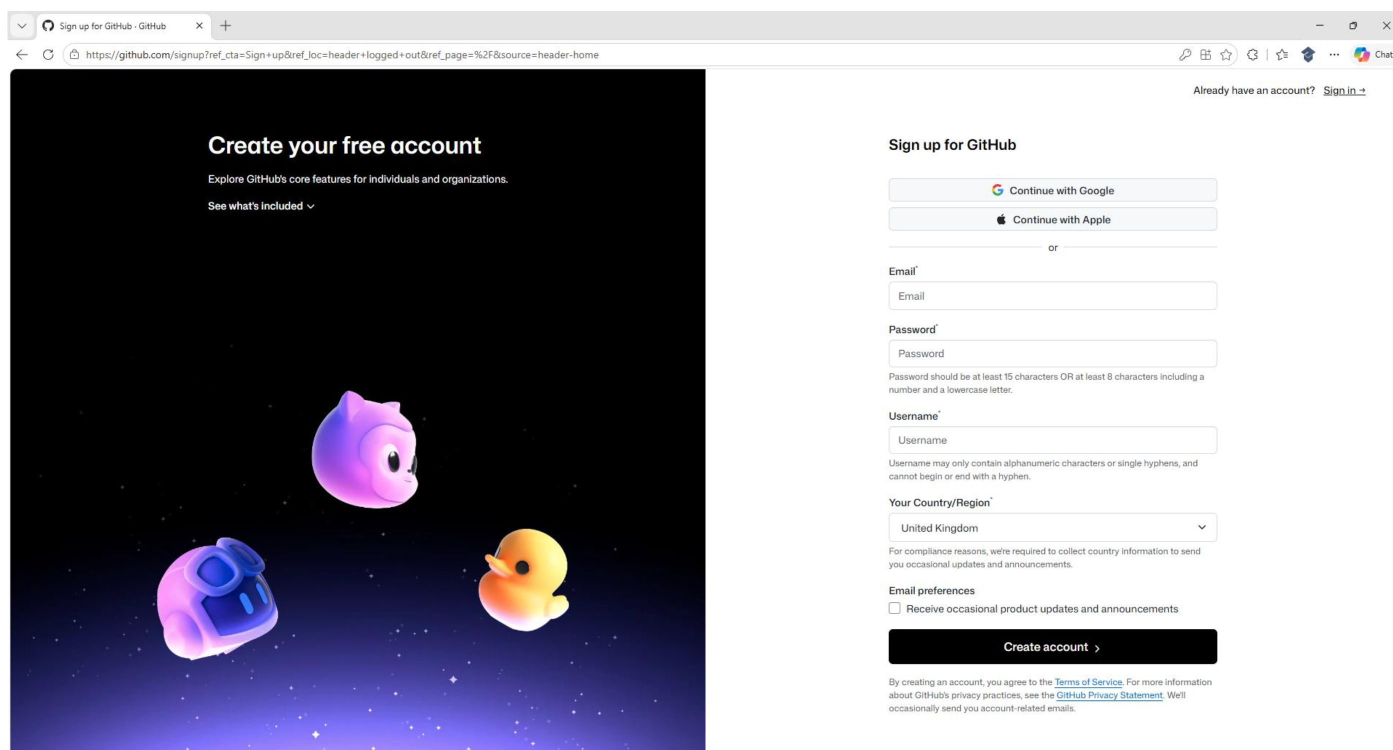
Create GitHub Account

If you don't have an existing **GitHub Account** you will need to **Sign up** for one, to do so use a **Browser** and visit the **Website** at github.com which also includes more about **GitHub**.

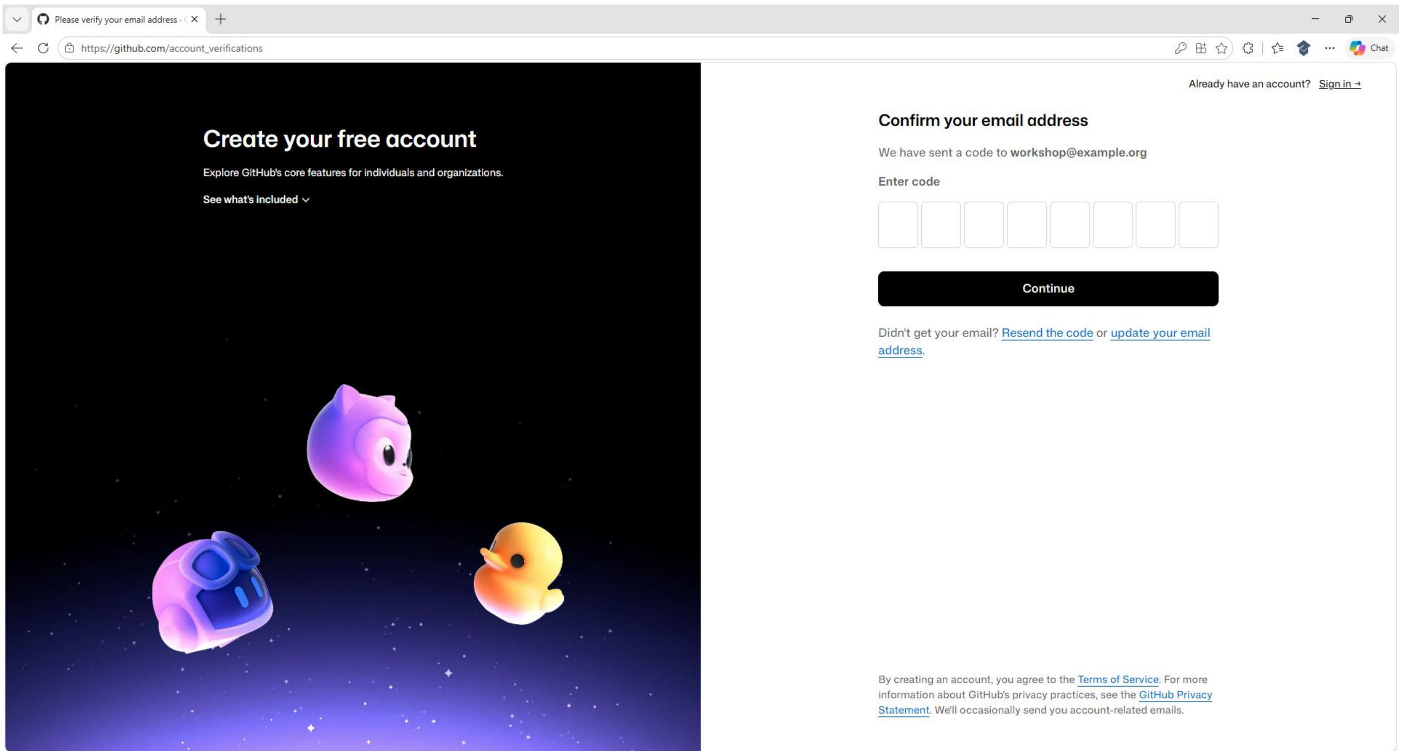


Information – **GitHub** is where you can collaboratively store and share code for any language or platform such as **.NET** with **Repositories** where you can track and propose changes to help build and ship software.

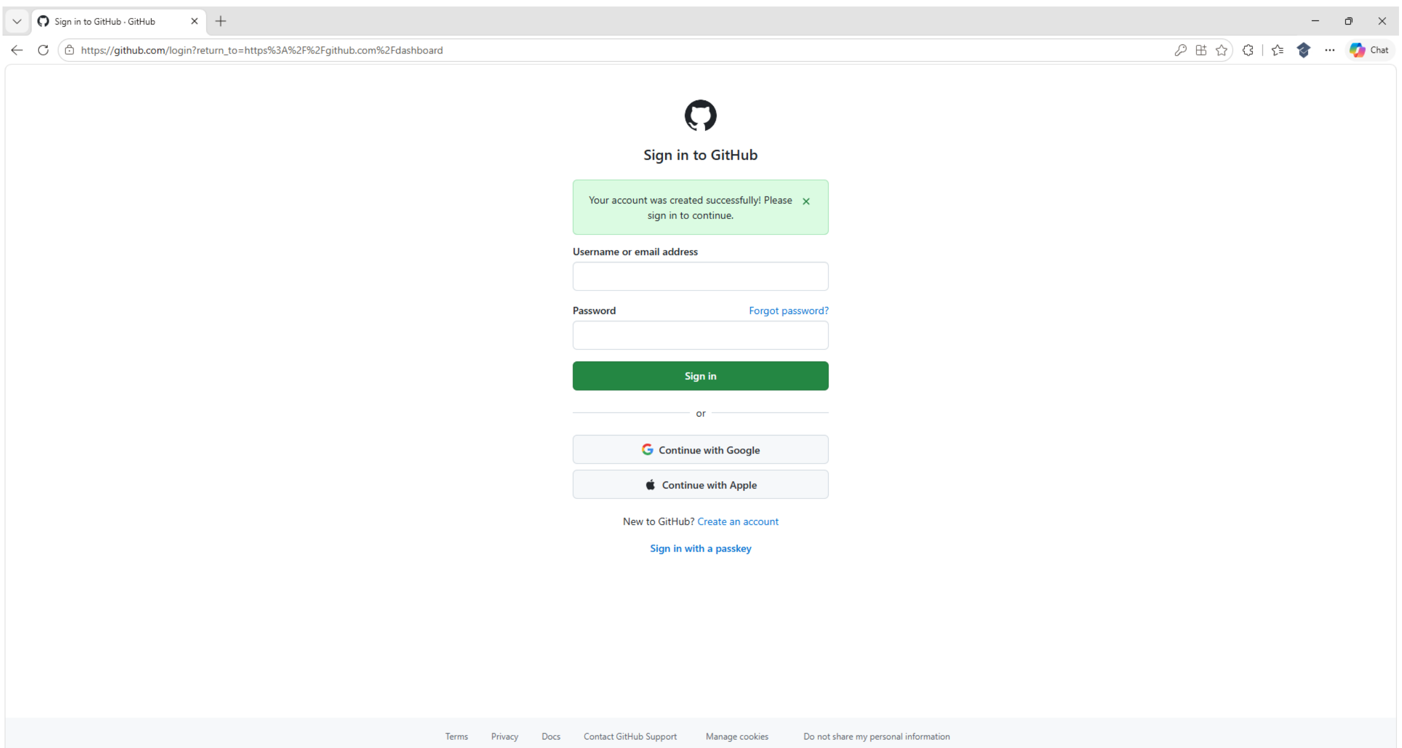
Then select **Sign up** where you can input your **Email** then a **Password** and **Username** to use for **GitHub** along with your **Country/Region** such as **United Kingdom** and then select **Continue**.



Then you may be asked to **Verify** your **Account** and then to **Confirm** the **Email Address** for your **Account**.



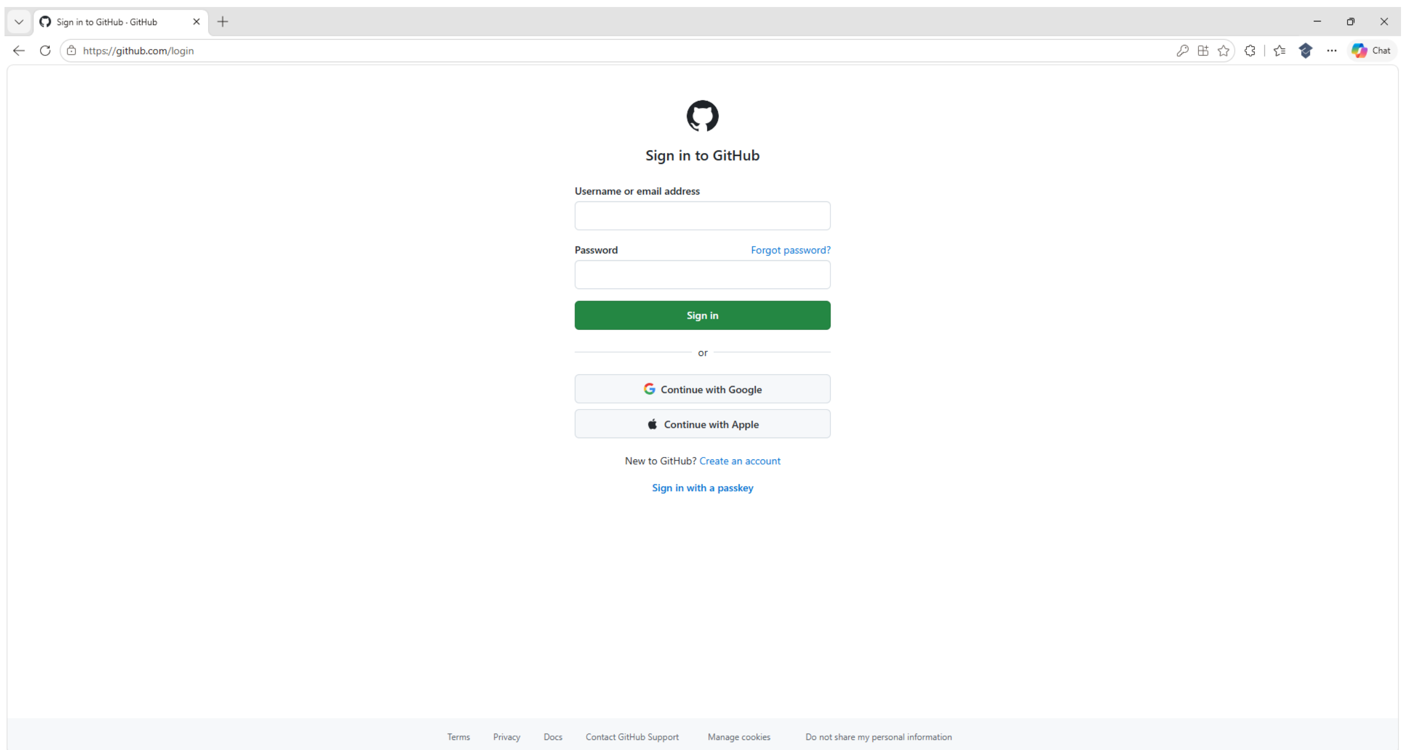
Next you need to check your **Email** for one for **Your GitHub launch code** that would have been **Sent** to the **Email Address** you provided for your new **Account** which you can then *Copy* and *Paste* below **Enter Code** on the **Confirm your email address** and then select **Continue** to create your **GitHub Account**.



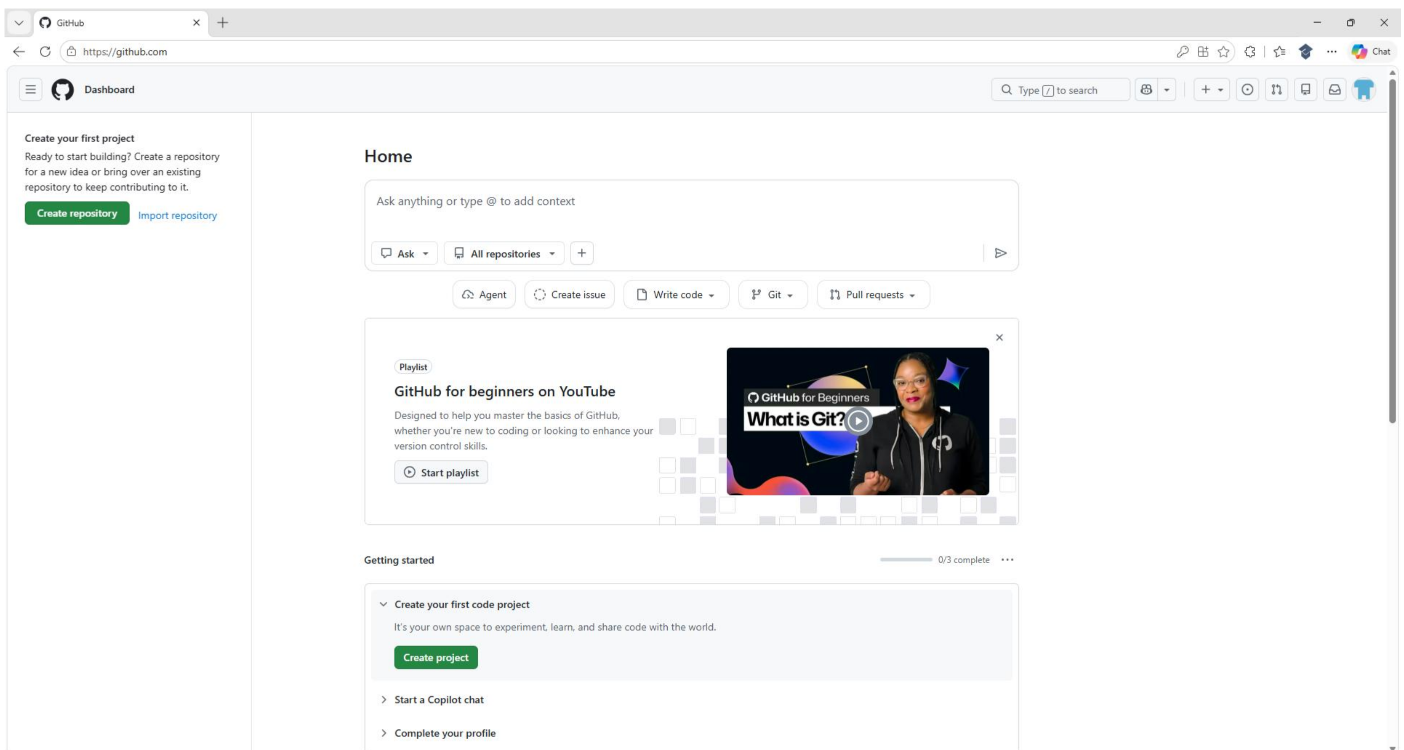
That completes creating a new **GitHub Account** that will be used for **GitHub Copilot** in the **Workshop**.

Enable GitHub Copilot

You will need to enable **GitHub Copilot**, if has not been enabled for **GitHub Account**, which you can do by using a **Browser** and visiting the **Website** at github.com and select **Sign in**, or once you have created a new **GitHub Account** you can input your **Username or email address** and **Password**.



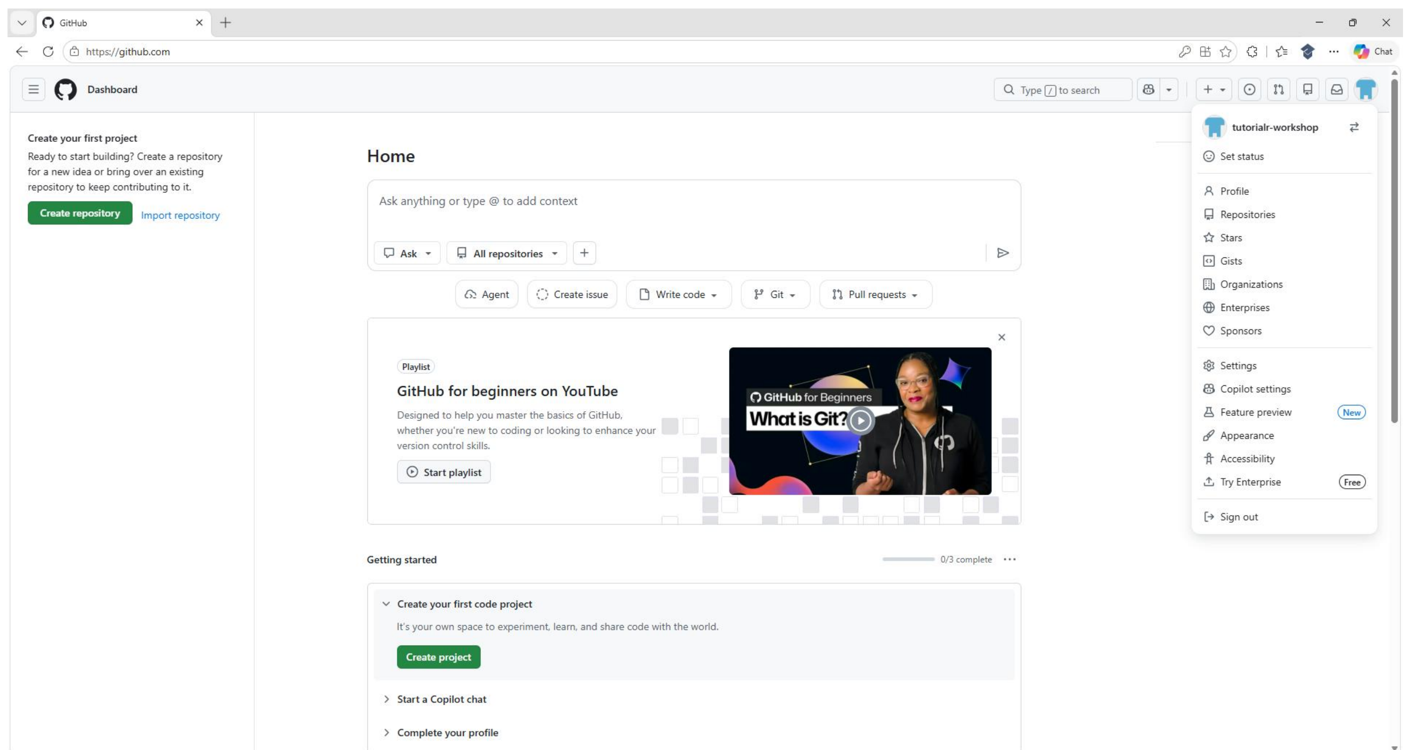
Then select **Sign in** to your **GitHub Account** and once done you will be taken to the **Dashboard** as follows:



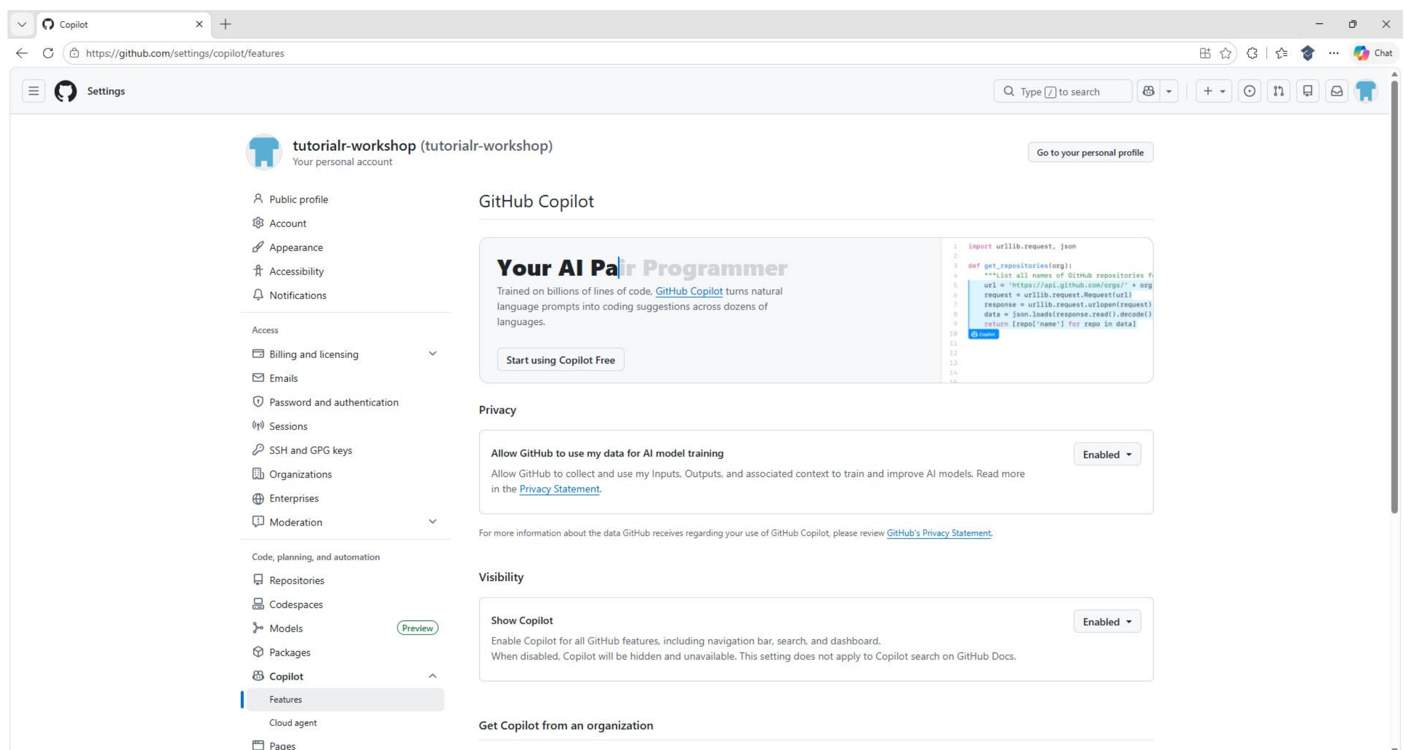
Information – **Dashboard** is where you get started with **GitHub** including creating your own **Repositories** where you can store and share **Code**, collaborate with others or contribute **Code** to others on **GitHub**.

CC BY SA

Once in the **Dashboard** for your **GitHub Account** select the **Avatar** from the top of the **Dashboard** to display the **Account Menu** as follows:

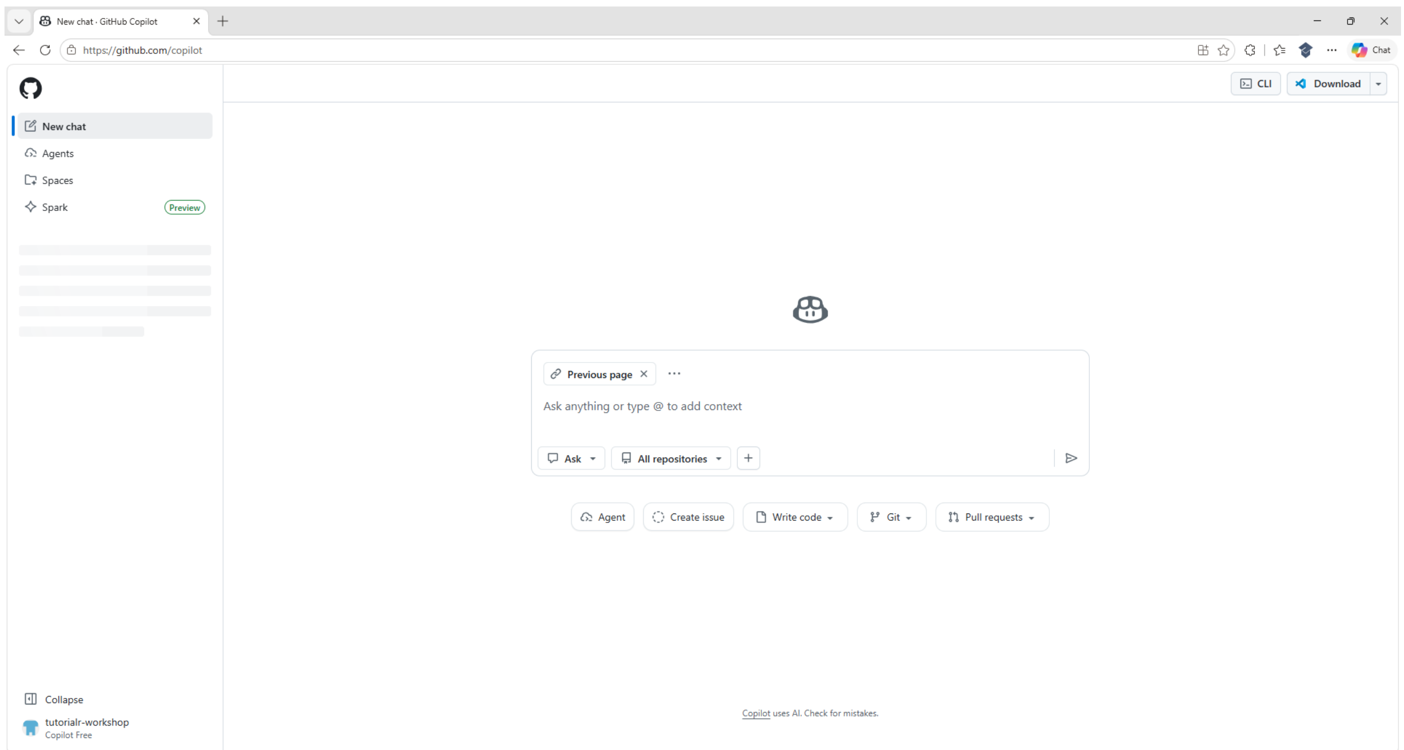


Then from the **Account Menu** select **Copilot Settings** for your **GitHub Account**.



Information – Once you have enabled **GitHub Copilot** then the **Copilot Settings** can be used to track your usage, which is more limited with **Copilot Free**, or adjust any settings for **GitHub Copilot**.

Then in **GitHub Copilot** select **Start using Copilot Free** which will take you to the online **GitHub Copilot**:

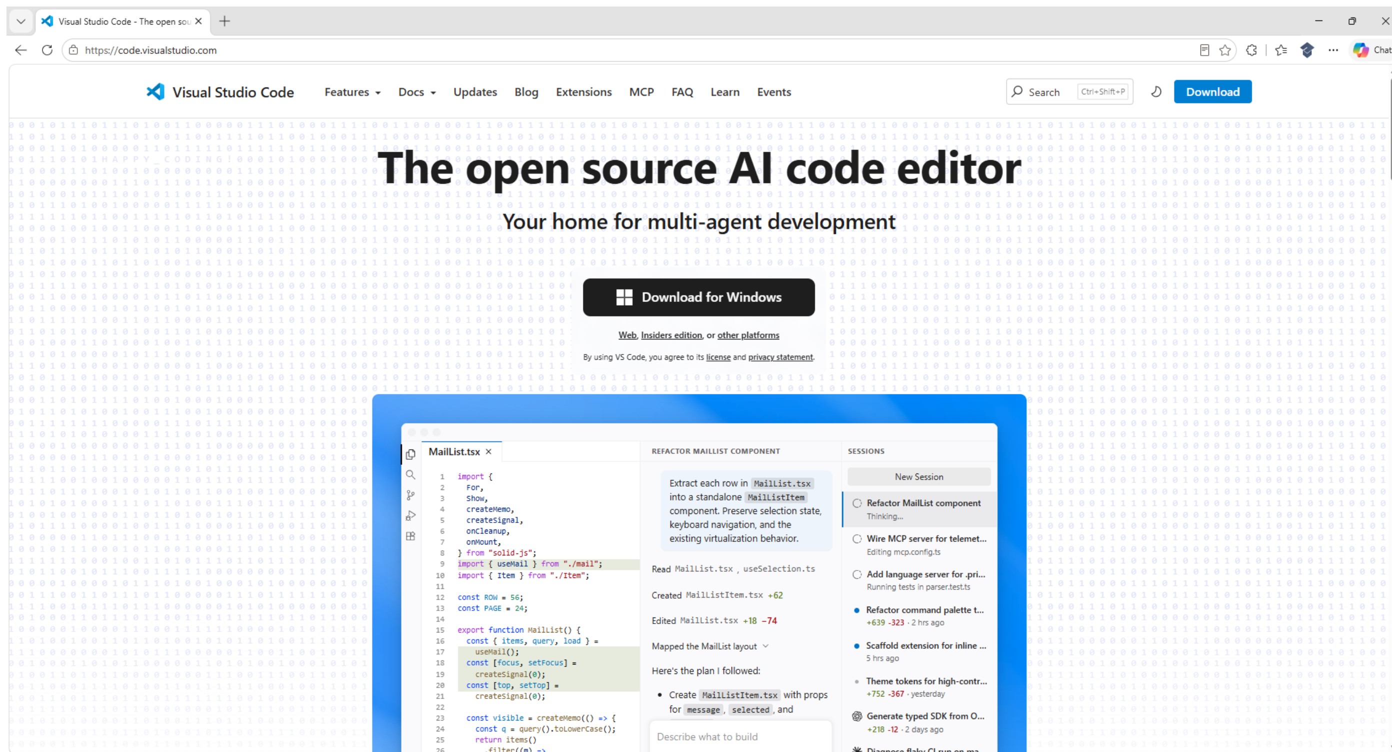


Information – **GitHub Copilot** is an AI assistant that builds with you that can generate code, manage tasks and explore your projects, it can also take advantage of data and tools using **Model Context Protocol** from an **MCP Server**.

That completes enabling a **GitHub Copilot** for your new or existing **GitHub Account** for the **Workshop**.

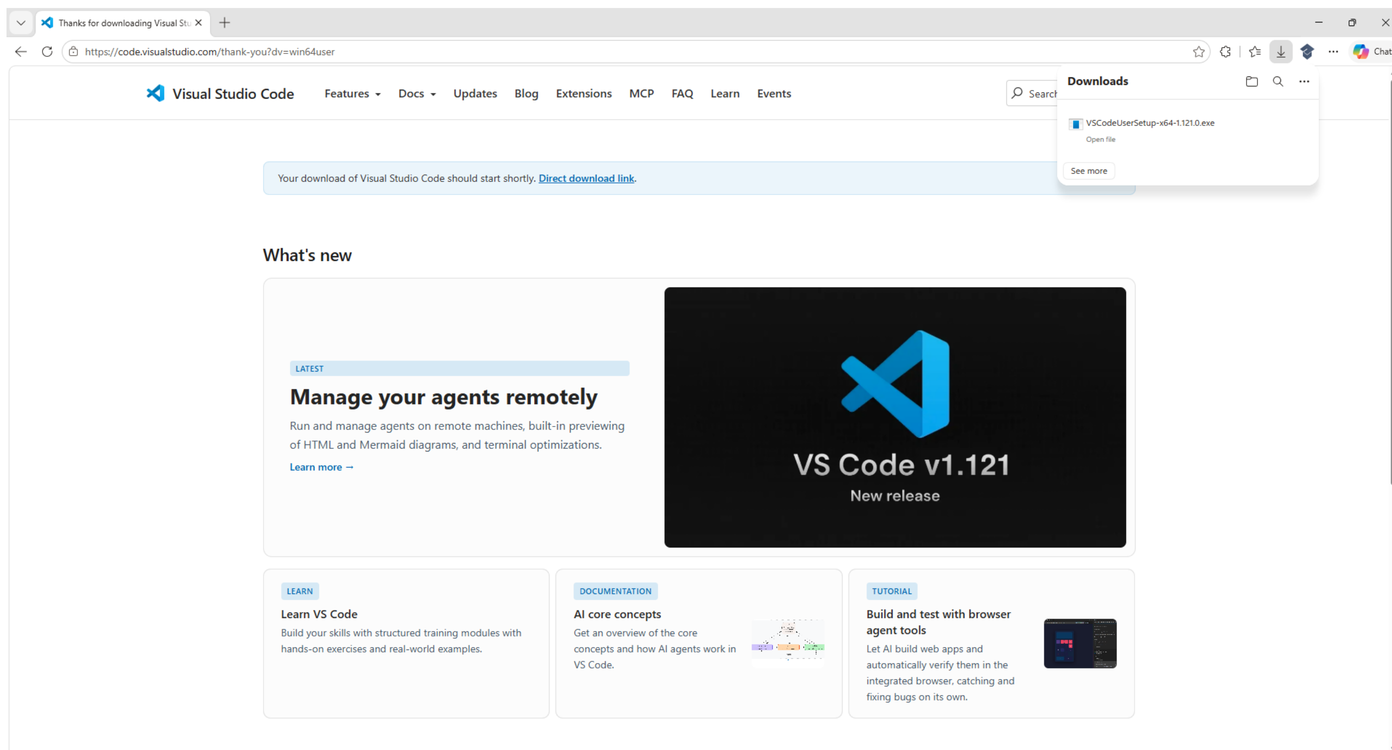
Install Visual Studio Code

You will need to **Download** the latest **Visual Studio Code** for **Windows** or **Mac**, to do this use a **Browser** and visit the **Website** at code.visualstudio.com where you'll also find out more about **Visual Studio Code**.

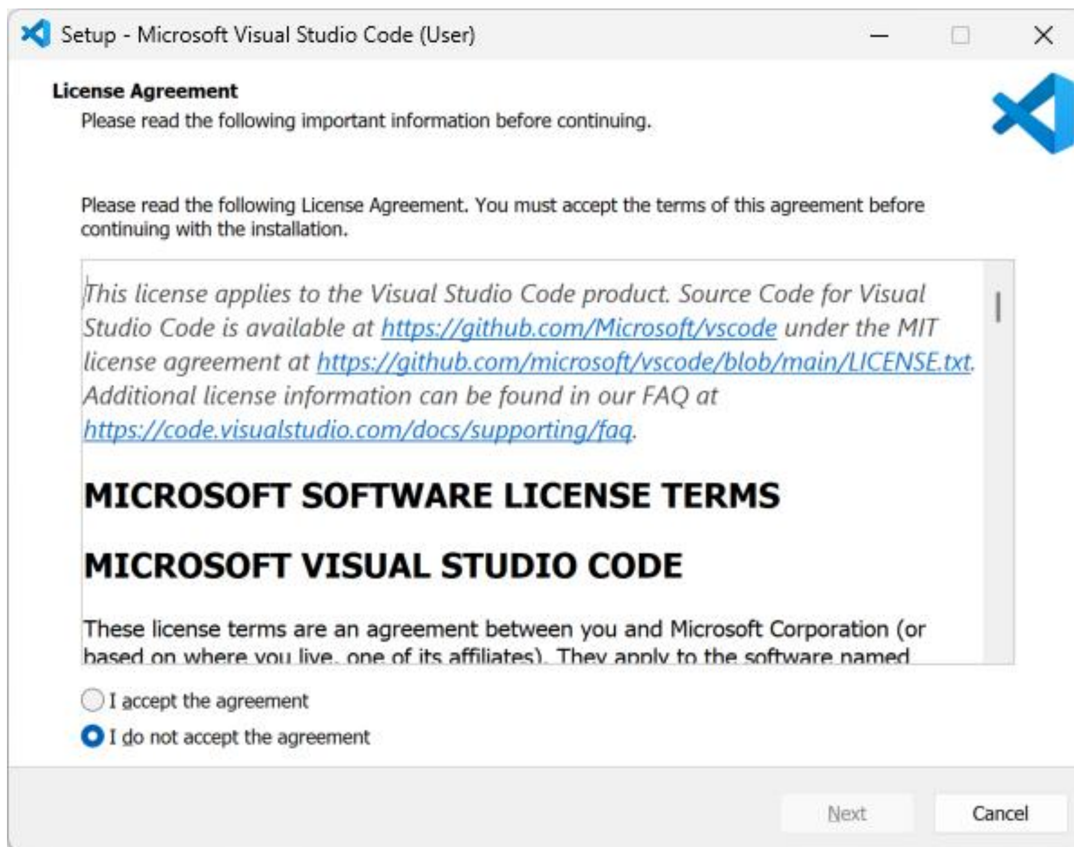


Information – **Visual Studio Code** is the free and open-source code editor with support for every major programming language including **C#** which is used with **.NET** and with optional support for AI features.

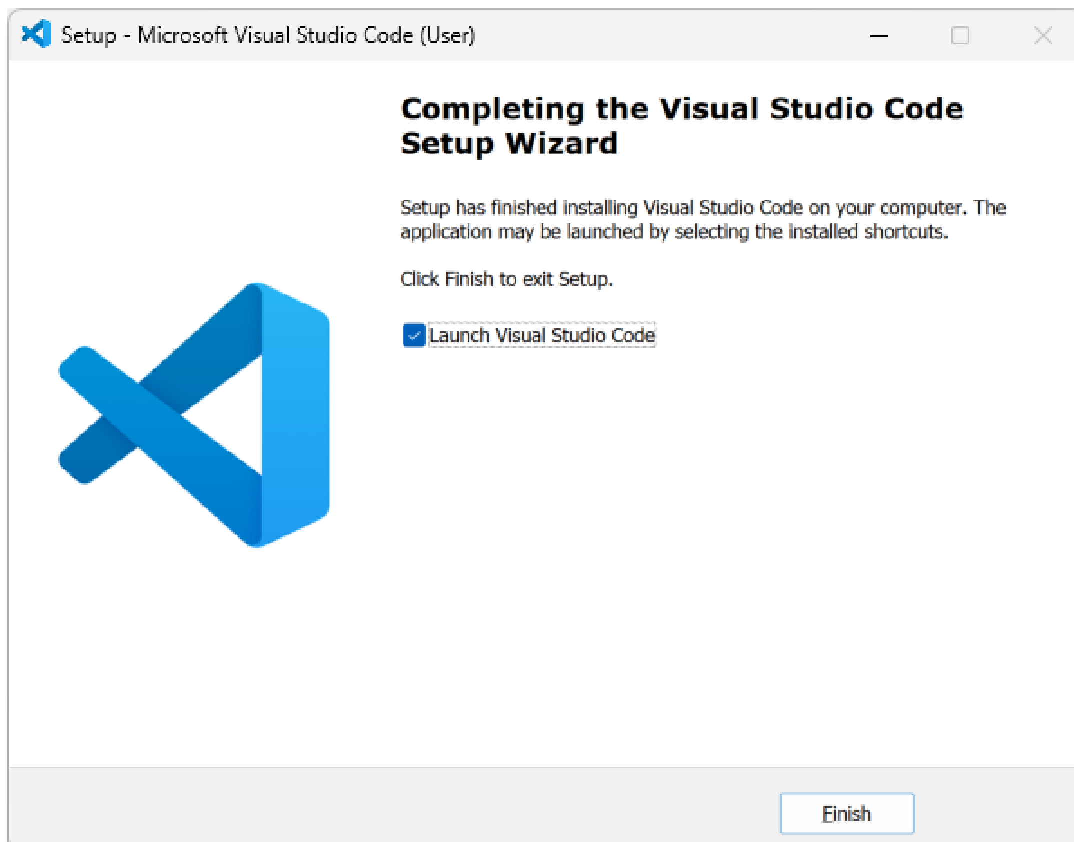
Next select the **Download for Windows** option in this case for **Windows** or the option to **Download** for **Mac** although your exact **Version** of **Visual Studio Code** may be different or newer.



The **Installer** for **Visual Studio Code** will begin **Downloading** and once it has been **Downloaded** it will show in **Downloads** in your **Browser** where you can **Open** it to launch the **Installer** as follows:

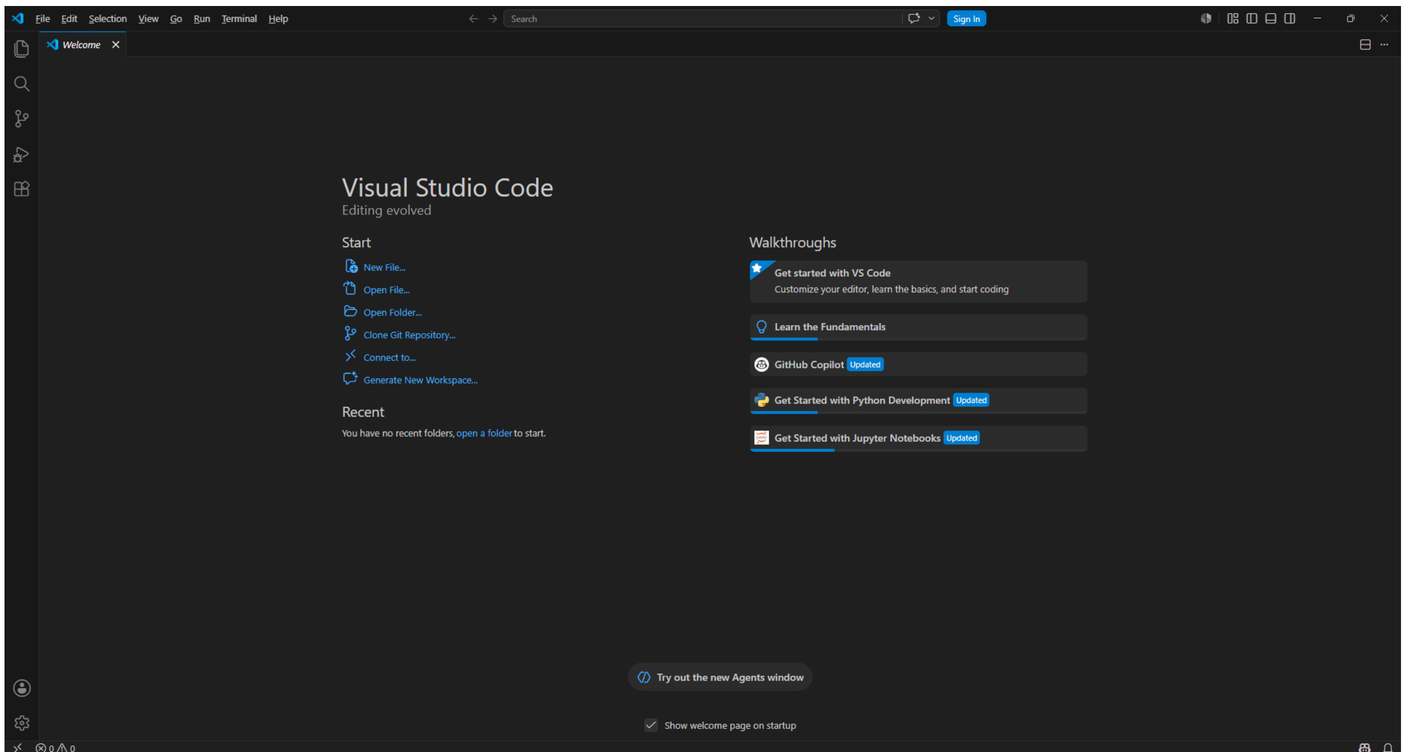


You will need to select **I accept the agreement** and then select **Next** and keep selecting **Next** for the rest of the **Installer** as you don't need to change anything, once you get to the end select **Finish** as follows:

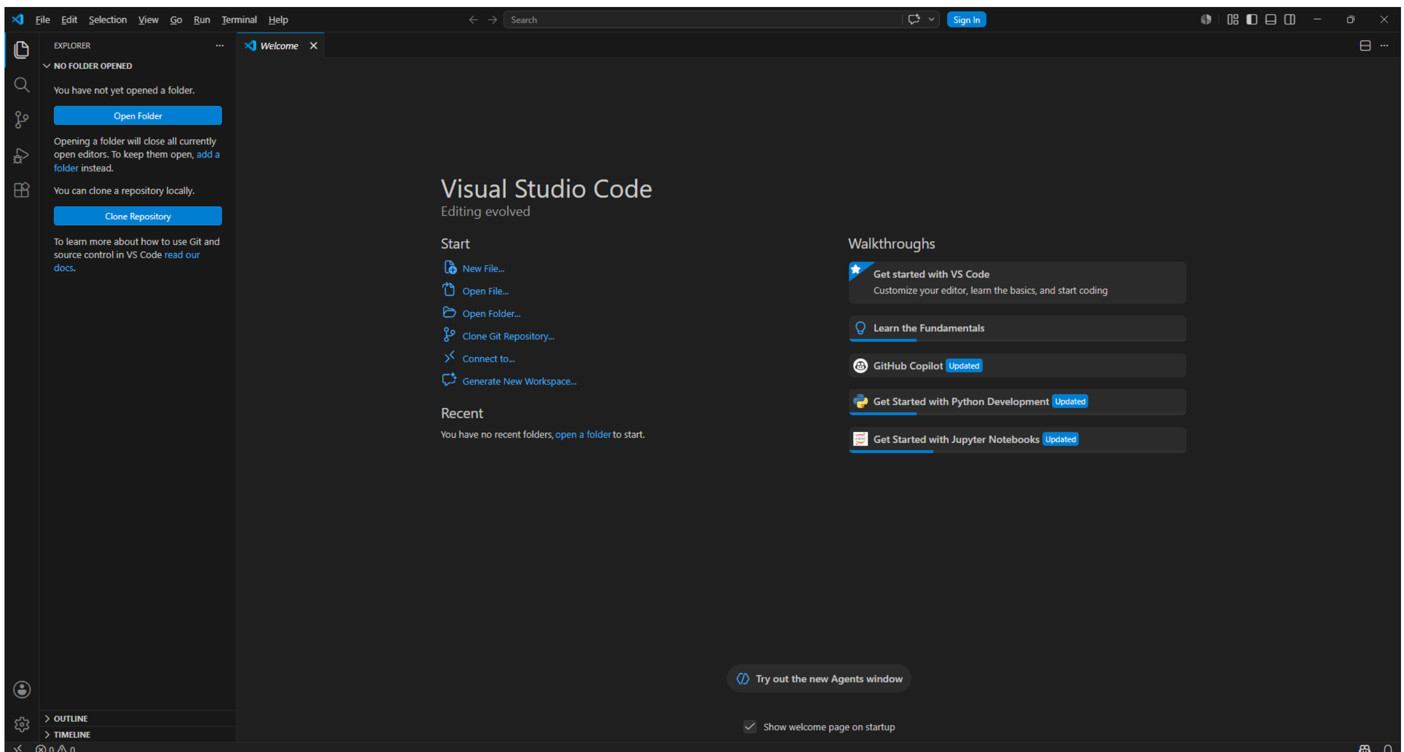


Launch Visual Studio Code

Once you have installed **Visual Studio Code** the **Installer** will have launched **Visual Studio Code**, but if **Visual Studio Code** was already **Installed**, then on **Mac** you need to go to **Finder** and then search for **Visual Studio Code** and then select it to **Launch** it, or if using **Windows** you need to go to **Start**, and then search for **Visual Studio Code** and then select it to **Launch** it as follows:

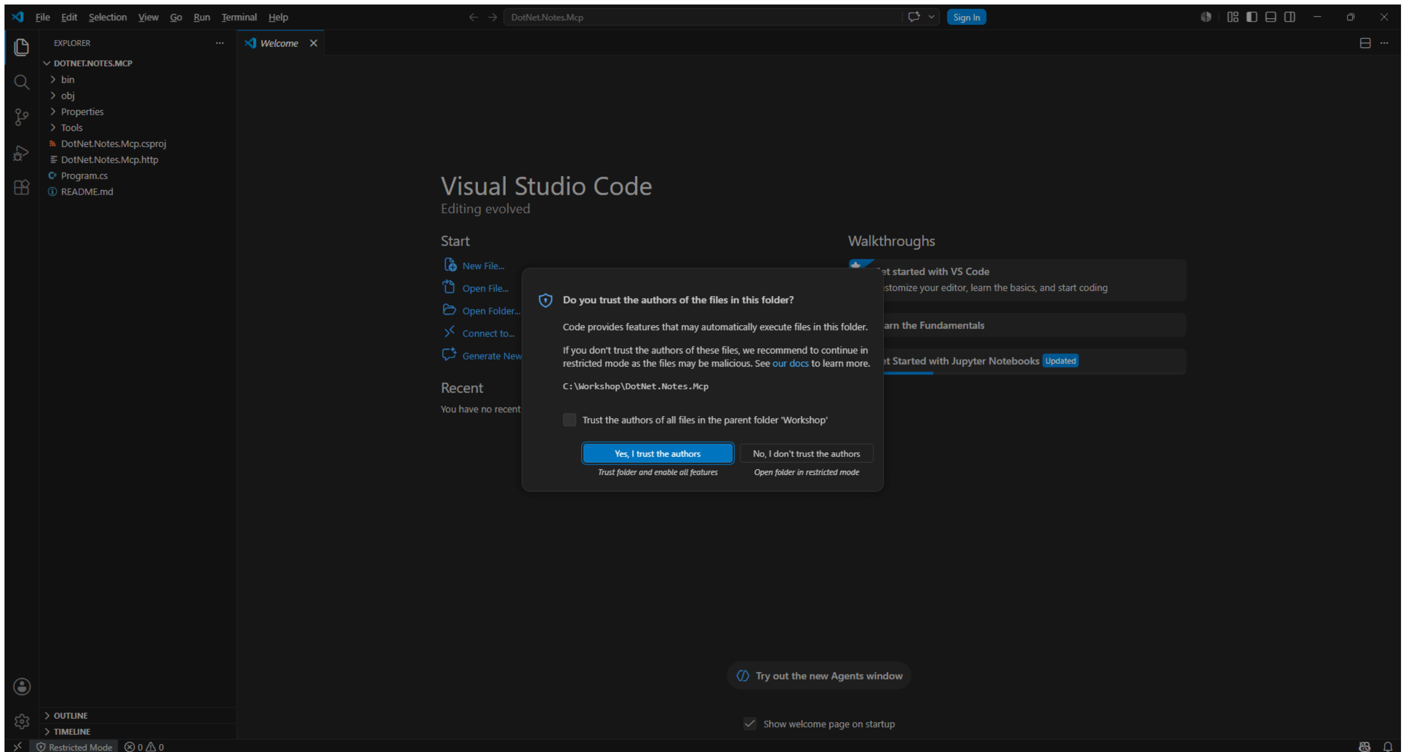


Then in **Visual Studio Code** from the **Side Bar** select the top option of **Explorer** as follows:

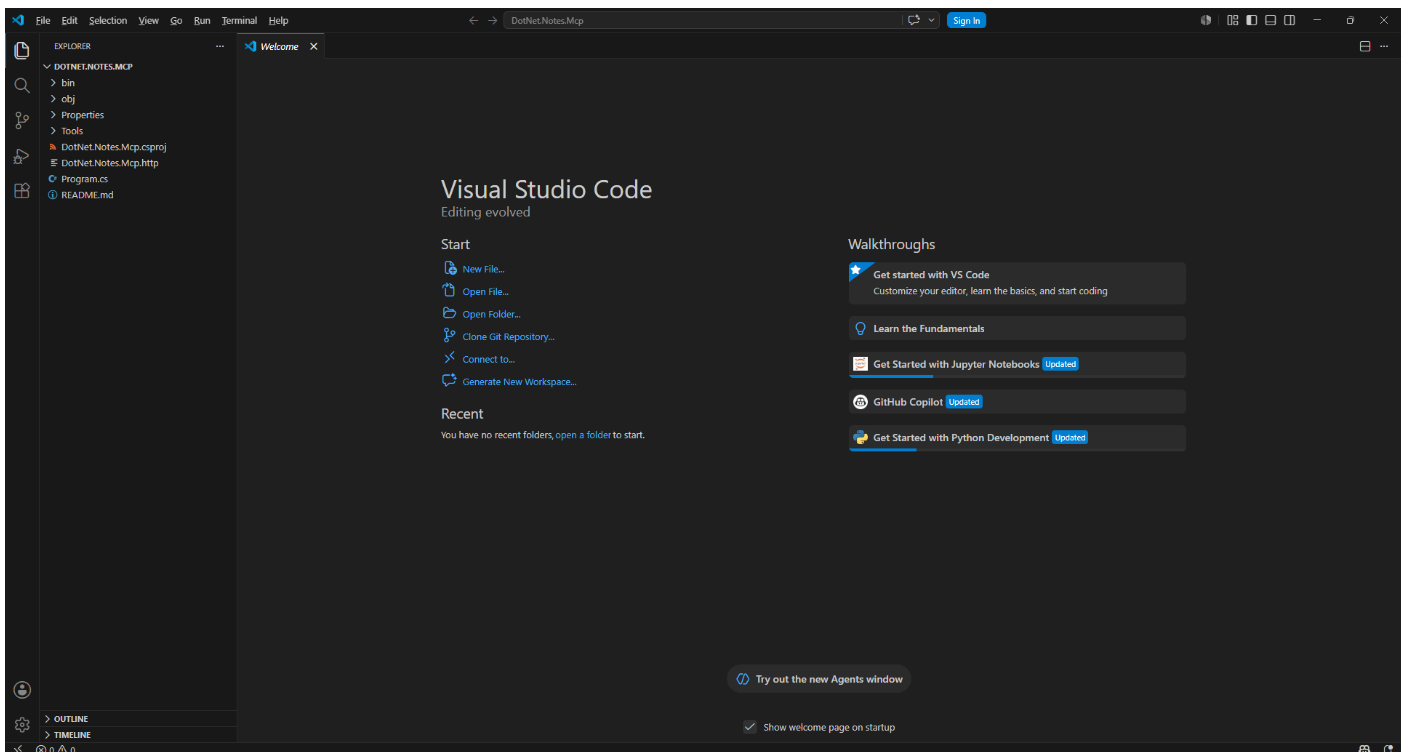


CC BY SA

Next in **Visual Studio Code** within **Explorer** select **Open Folder** and locate the **Folder** for your **Project**, using the one in **Terminal** on **Mac** or **Command Prompt** on **Windows** e.g. *C:\Workshop\DotNet.Notes.Mcp* and then choose **Select Folder** or if you cannot find it *Type* in the **Terminal** on **Mac** or **Command Prompt** on **Windows** the **Command** of **code .** followed by **Enter**, which should open the **Folder** as follows:



Then choose **Yes, I trust the authors** in **Do you trust the authors of the files in this folder?** as this is the **Folder** for the **Project** you created as follows:



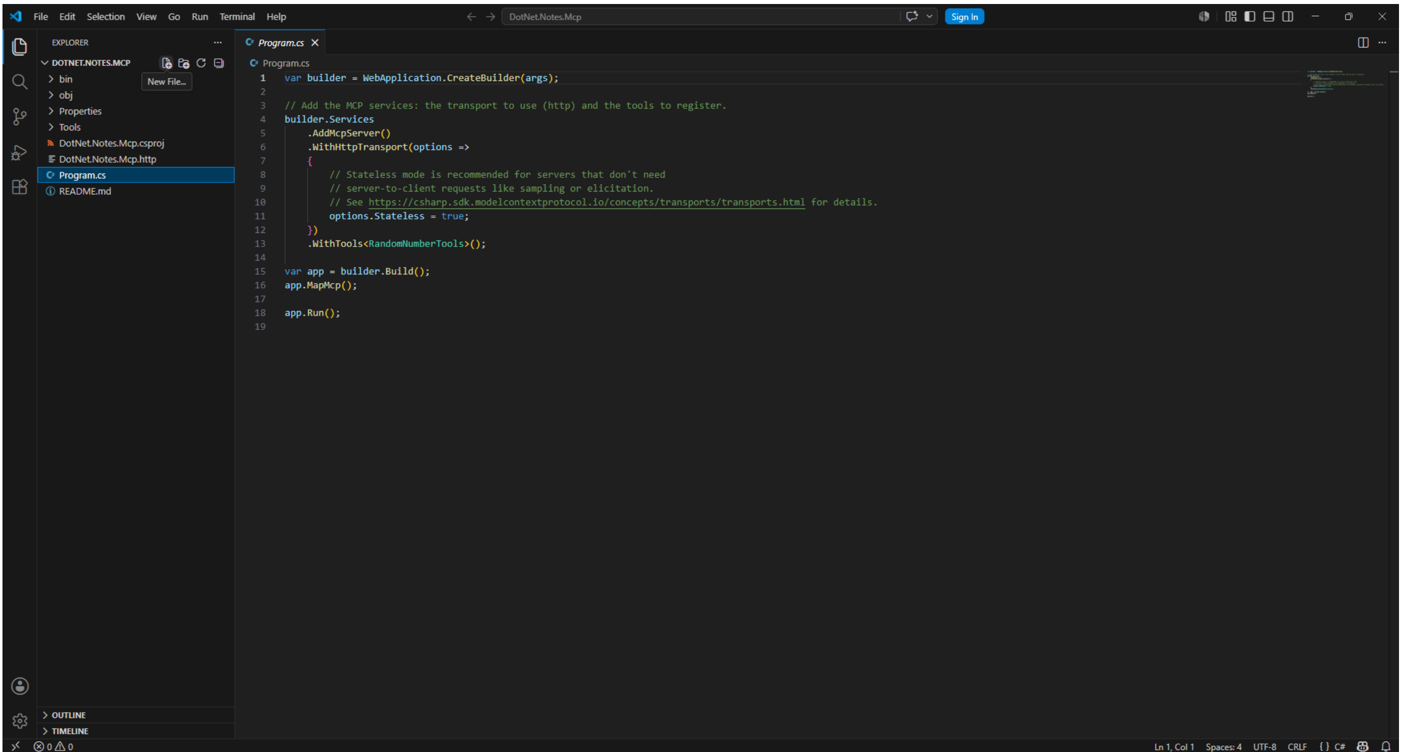
This completes launching **Visual Studio Code** and opening the **Project** of **DotNet.Notes.Mcp**.

CC BY SA

Implement

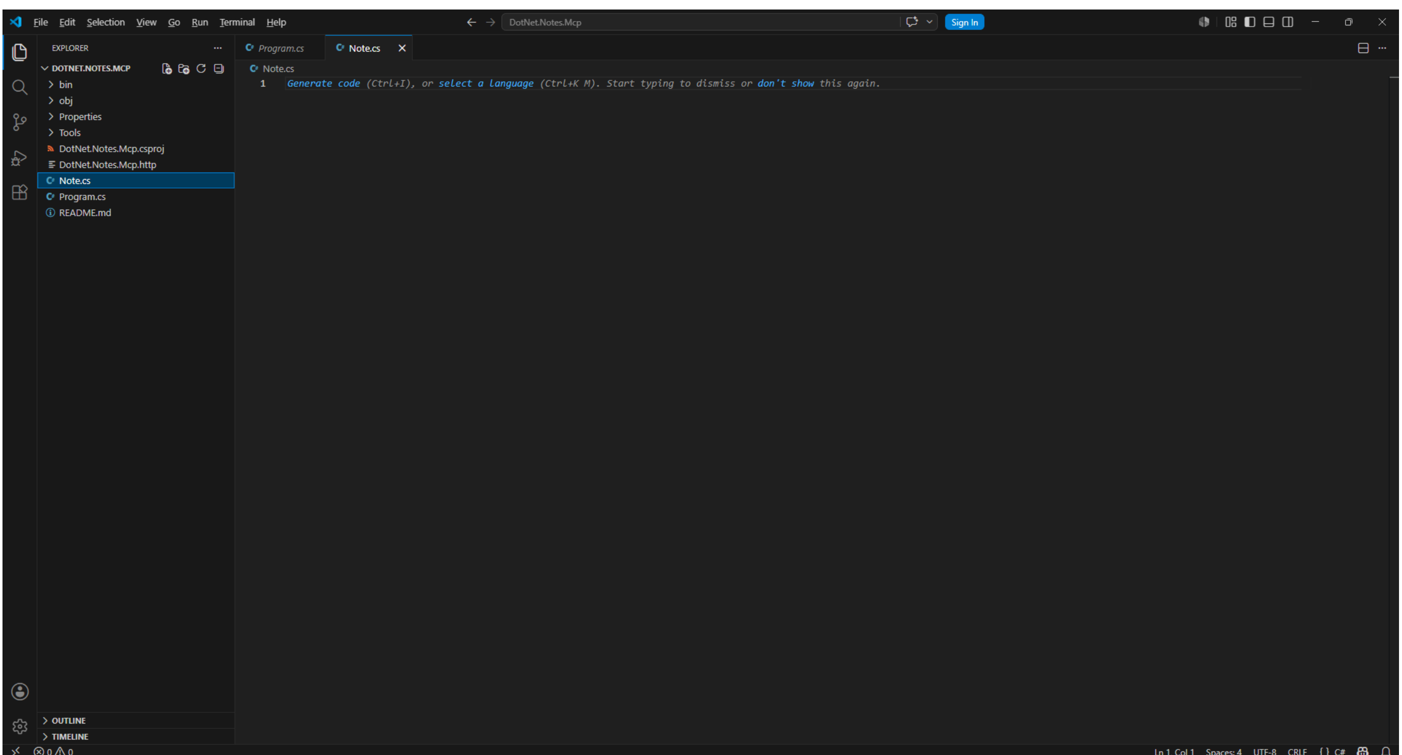
Note Class

In **Visual Studio Code** select **Program.cs** in **Explorer** then choose **New File...** next to **DotNet.Notes.MCP**.



Then *Type* in the following **Name** and press **Enter** after which you should see or select a blank **Provider.cs** in **Explorer** within **Visual Studio Code**.

```
Note.cs
```

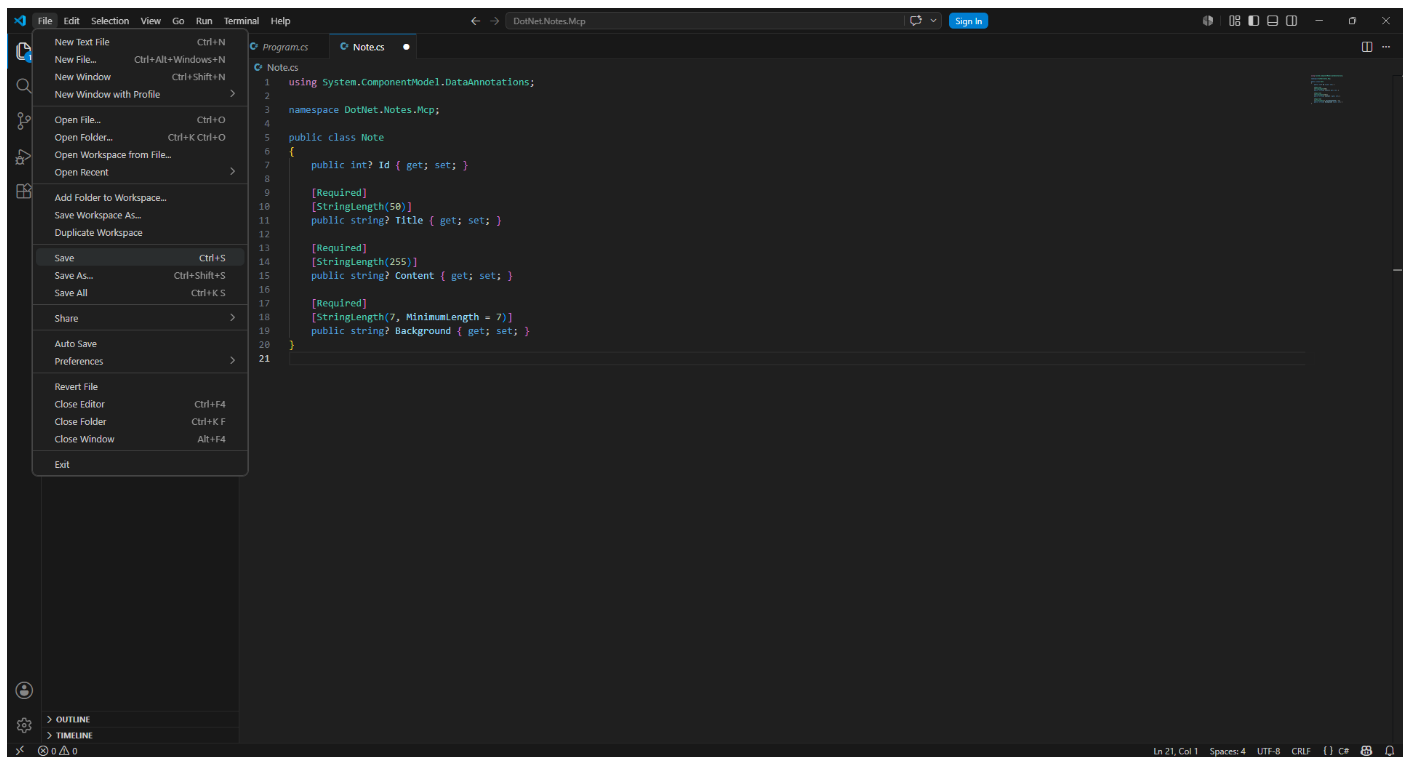


Then within **Visual Studio Code** in **Note.cs** you need to *Copy* and *Paste* the following **Code**:

```
using System.ComponentModel.DataAnnotations;

namespace DotNet.Notes.Mcp;

public class Note
{
    public int? Id { get; set; }

    [Required]
    [StringLength(50)]
    public string? Title { get; set; }

    [Required]
    [StringLength(255)]
    public string? Content { get; set; }

    [Required]
    [StringLength(7, MinimumLength = 7)]
    public string? Background { get; set; }
}
```

Information – This is the `class` for `Note`, a **Class** is used to represent an **Object** or used to group together **Code**. The first line is a `using` of `System.ComponentModel.DataAnnotations` needed for **Attributes** used for **Validation**, then there is a `namespace` for **DotNet.Notes.Mcp** used to identify the **Code** for the **Project**. This is followed by the `class` to define a `Note`, it contains **Properties** for the **Note**, starting with `Id` that will help identify a **Note** using an `int?`, an `int` represents a number and the `?` means it can support having no value which is known as `null`. The next **Property** is for `Title`, which is a `string?`, that represents text, this will be for the **Title** of the **Note**, it has an **Attribute** denoted with square brackets of `[` and `]` for `Required`, meaning it will need be provided with a value, but for the **Class** itself it can be `null` denoted with the `?`, and another **Attribute** for `StringLength(50)` which means it can only be up to fifty characters long. The next **Property** is for `Content`, which is a `string?` which is for the **Content** of the **Note** and indicated by the **Attributes** will also need a value and be a maximum of two hundred and fifty-five characters. The final **Property** is for `Background`, used to represent the **Colour** of the **Note**, with the **Attributes** indicating needing a value and be maximum and minimum of seven characters, with addition of `MinimumLength`.

Next, within **Visual Studio Code** from the **Menu** select **File** and then **Save** as follows:



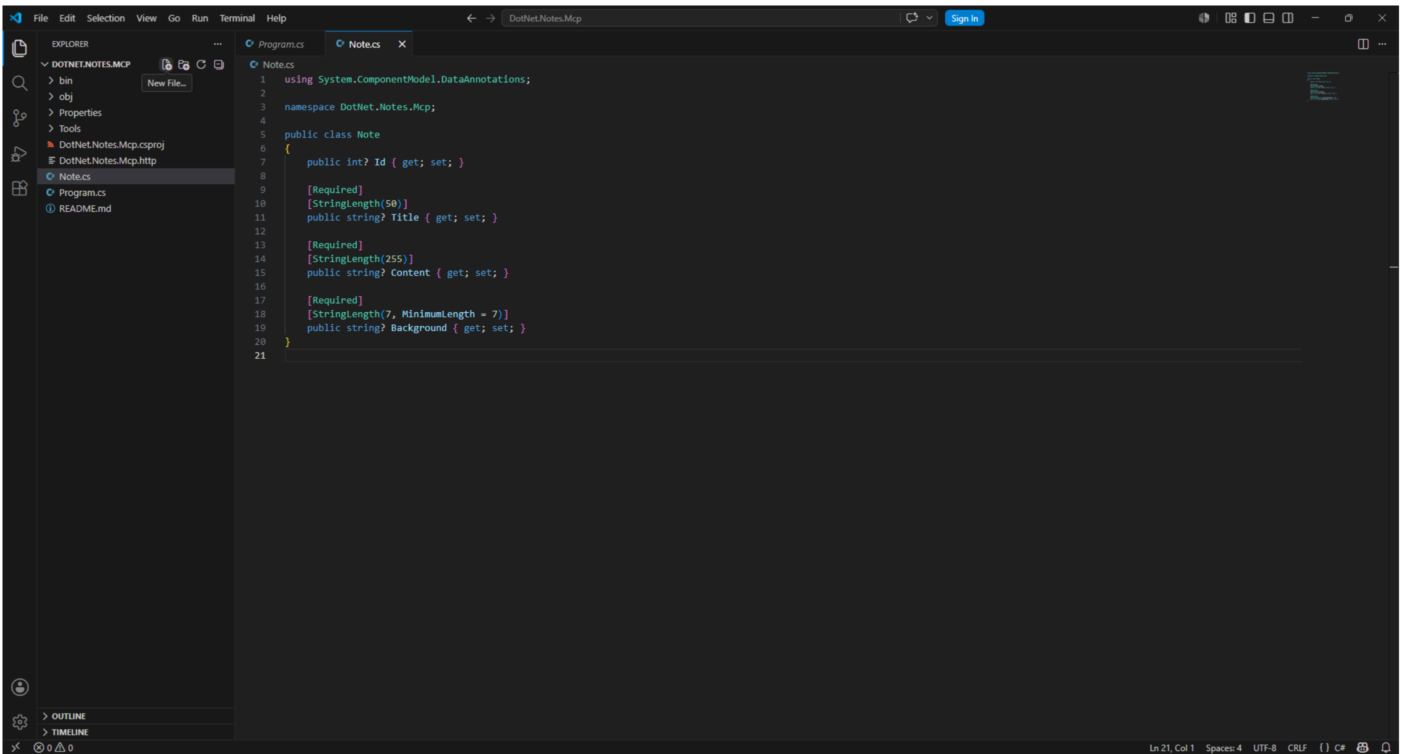
Information – You will be using *Copy* and *Paste* for each piece of **Code** but to avoid any issues the key thing to remember in **C#** is balance, so a **Curly Brace** of **{** will always have a counterpart of **}** also applies to **Square Brackets** which should have both **[** and **]** and **Brackets** which should have always have **(** and **)** along with **Double Quotes** which should be in pairs so if see **"** at the start there should be another **"** at the end. Also make sure where you see any **Semi-Colons** or **;** to include them where needed as it can often be the smallest mistake that is easiest to fix makes your **Code** work when corrected although any indentation including **Tabs** won't affect any behaviour. **Errors** will give you an idea where to look including the line number of the **File** which will make them easier to find and give you some idea of what you did wrong so you can correct any mistake.

You should double check that everything was entered correctly as if the **Terminal** on **Mac** or **Command Prompt** on **Windows** displays any **Errors** this will be only for any **Code** that was entered into **Note.cs**, so go over the previous **Steps** to double check it matches what you have but once any corrections have been made and **Saved** or there are no **Errors** then the **Build** should proceed.

This completes the process in **Visual Studio Code** of creating the **Class** to represent the **Note**.

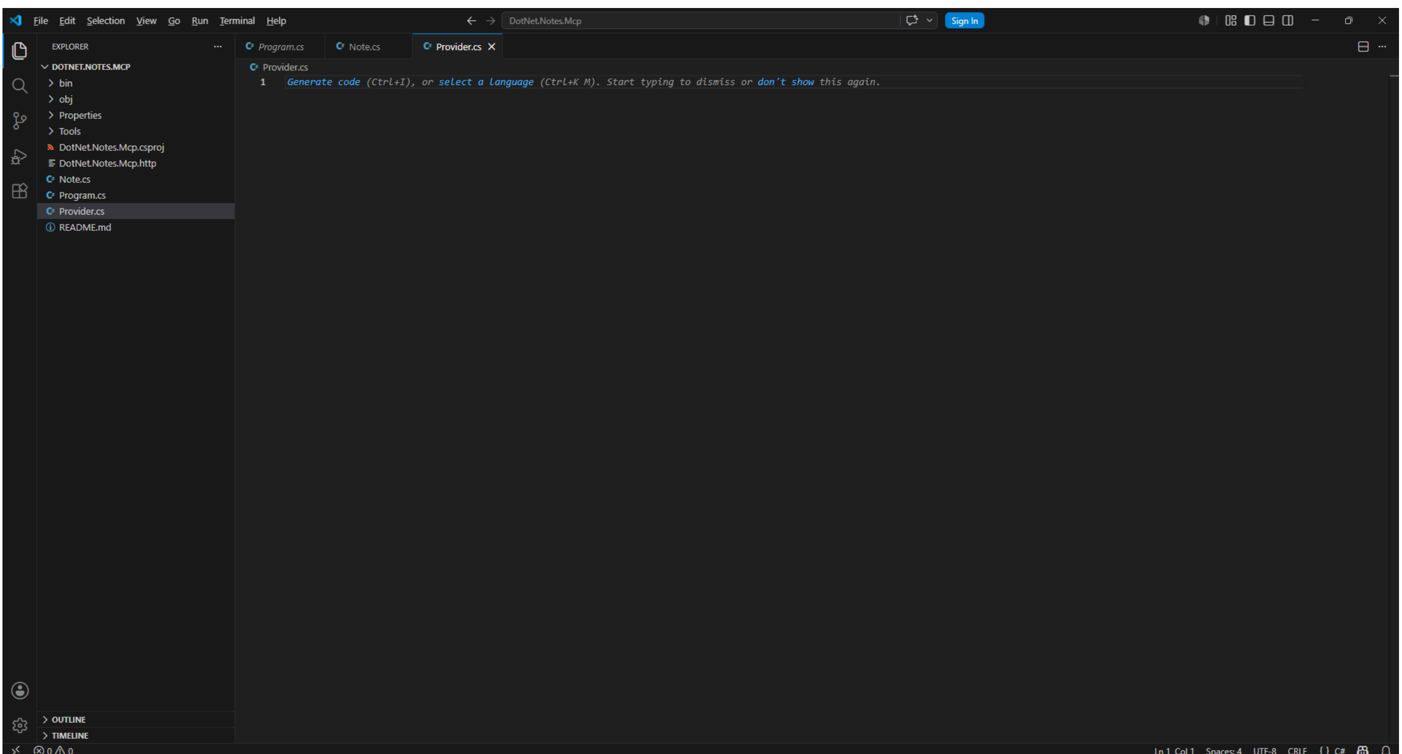
Provider Class

In **Visual Studio Code** select **Note.cs** in **Explorer** then choose **New File...** next to **DotNet.Notes.Mcp**.



Then *Type* in the following **Name** and press **Enter** after which you should see or select a blank **Provider.cs** in **Explorer** within **Visual Studio Code**.

```
Provider.cs
```



Then within **Visual Studio Code** in **Provider.cs** you need to *Copy* and *Paste* the following **Code**:

```
using Microsoft.Data.Sqlite;

namespace DotNet.Notes.Mcp;

public class Provider
{
    // Constants

    // Run & Read Methods

    // Create & Add Methods

    // Get & List Methods

    // Edit & Delete Methods
}
```

Information – This forms the outline of the **`class`** for **`Provider`**, a **Class** is used to group together **Code** or can be used to represent an **Object**. The first line is a **`using`** for functionality needed from the **Package** of **Microsoft.Data.Sqlite**. Then there is a **`namespace`** for **DotNet.Notes.Mcp** which is followed by the **`class`** which is defined for **`Provider`**. There are also **Comments** which are the lines starting with **`//`** which will help you place **Code** from the next few **Steps** of the **Workshop**.

Next within **Visual Studio Code** in **Provider.cs** underneath **`// Constants`** you need to *Copy* and *Paste* the following **Code**:

```
private const string note_id = "@Id";
private const string note_title = "@Title";
private const string note_content = "@Content";
private const string note_background = "@Background";
private const string connection = "Filename=notes.db";
private const string create = @"CREATE TABLE IF NOT EXISTS Notes
(Id INTEGER PRIMARY KEY AUTOINCREMENT, Title NVARCHAR(50) NULL,
Content NVARCHAR(255) NULL, Background NVARCHAR(7) NULL);";
private const string add = @"INSERT INTO Notes VALUES
(NULL, @Title, @Content, @Background); SELECT last_insert_rowid();";
private const string get = @"SELECT Id, Title, Content, Background
FROM Notes WHERE Id = @Id;";
private const string list = @"SELECT Id, Title, Content, Background
FROM Notes;";
private const string edit = @"UPDATE Notes SET Title = @Title,
Content = @Content, Background = @Background WHERE Id = @Id;";
private const string delete = "DELETE FROM Notes WHERE Id = @Id;";
```

Information – This **Code** defines **Constants** or **`const`**, values that do not change when your application is running, and they are **`private`** as they are only used within the **`class`** of **`Provider`**, each is a **`string`** that represent text. The first four are names of **Fields** in the **Database** for a **Note**, the next is for the **Connection** to the **Database** and those remaining are **Commands** written in **SQL** used by **SQLite** which will be used to create the **Table** in the **Database** mirroring **`class`** of **`Note`**, followed by other **Commands** in **SQL** that will be used for **Notes** to add with **INSERT**, get or list with **SELECT**, edit with **UPDATE** or delete with **DELETE**.

Then within **Visual Studio Code** in **Provider.cs** underneath **`// Run & Read Methods`** you need to *Copy* and *Paste* the following **Code**:

```
private static async Task<TValue?> Run<TValue>(string sql,
    Func<SqliteCommand, Task<TValue>> action)
{
    try
    {
        using SqliteConnection conn = new(connection);
        await conn.OpenAsync();
        using SqliteCommand cmd = new(sql, conn);
        return await action(cmd);
    }
    catch
    {
        return default;
    }
}

private static Note Read(SqliteDataReader r) => new()
{
    Id = r.GetInt32(0),
    Title = r.GetString(1),
    Content = r.GetString(2),
    Background = r.GetString(3)
};
```

Information – This **Code** defines two **Methods**, which are **`private`**, so they are only used within the **`class`** of **`Provider`**. The first **Method** is **`Run`** which will be used with **Commands** passed in with the **Parameter** of **`sql`** and the other **Parameter** of **`action`** will be provided with a **Block** of **Code** that will use the **Command**, by opening a **Connection** to the **Database** with **`SqliteConnection`** and then preparing the **Command** with **`SqliteCommand`** then performing the **`action`** for the **Command** using **`SqliteCommand`** with the provided **Block** of **Code** and the output value, defined by **`TValue`**, which can be any **Type** such as a **`string?`** or **`int?`** returned using **`return`**. This **Method** performs any **Commands** within a **`try`** so if there are problems such as an **Error** then the **`catch`** will be performed to **`return`** the **`default`** of the **`TValue`** which for **`string?`** or **`int?`** which would be **`null`**. The second **Method** is for **`Read`** which will be used to read values into a **`Note`** from the **Row** of the **Table** in the **Database** with **`SqliteDataReader`** where it uses **`GetInt32`** for an **`int`** and **`GetString`** for a **`string`**, the numbers in brackets of **`(`** and **`)`** are the position of the value or **Field** in the **Row** of the **Table** in the **Database**, with **`0`** being the first, **`1`** the second and so on.

Next within **Visual Studio Code** in **Provider.cs** underneath **`// Create & Add Methods`** you need to *Copy* and *Paste* the following **Code**:

```
public Task<bool> Create() => Run(create, async cmd =>
{
    await cmd.ExecuteNonQueryAsync();
    return true;
});

public Task<int?> Add(Note note) => Run(add, async cmd =>
{
    cmd.Parameters.AddWithValue(note_title, note.Title);
    cmd.Parameters.AddWithValue(note_content, note.Content);
    cmd.Parameters.AddWithValue(note_background, note.Background);
    var result = await cmd.ExecuteScalarAsync();
    return result is not null ? Convert.ToInt32(result) : (int?)null;
});
```

Information – These **Methods** are **`public`** meaning they can be used inside and outside the **`class`** of **`Provider`**. The first **Method** is **`Create`** which will use the **Command** to create the **Table** used to store the **Notes** in the **Database**, with the **Block** of **Code** using the **Method** of **`ExecuteNonQueryAsync`** to do this and then **`return`** the value of **`true`**, should something go wrong then the **`default`** of **`false`** would be returned instead. The second **Method** is **`Add`** which uses the **Command** to insert a **Note** into the **Database** with the **Block** of **Code** using **`ExecuteScalarAsync`** to do this, the **Command** is provided values for the **Note** with **`AddWithValue`** and if successful then a **`result`** is returned converted to an **`int`** using the **Method** of **`Convert.ToInt32`** which is the **Id** for a **`Note`**, or if nothing or **`null`** came back or for any problem then the **`default`** of **`null`** would be returned. This is done using by **`is not null`** to check the **`result`** with a conditional **Operator** of question mark or **`?`** where if the check is **`true`** then the first part is returned, otherwise it is **`false`** and the second part after the colon or **`:`** is returned instead.

While still within **Provider.cs** in **Visual Studio Code** underneath **`// Get & List Methods`** you need to *Copy* and *Paste* the following **Code**:

```
public Task<Note?> Get(int id) => Run(get, async cmd =>
{
    cmd.Parameters.AddWithValue(note_id, id);
    using var result = await cmd.ExecuteReaderAsync();
    return await result.ReadAsync() ? Read(result) : null;
});

public Task<List<Note>?> List() => Run(list, async cmd =>
{
    using var reader = await cmd.ExecuteReaderAsync();
    List<Note> results = [];
    while (await reader.ReadAsync())
        results.Add(Read(reader));
    return results;
});
```

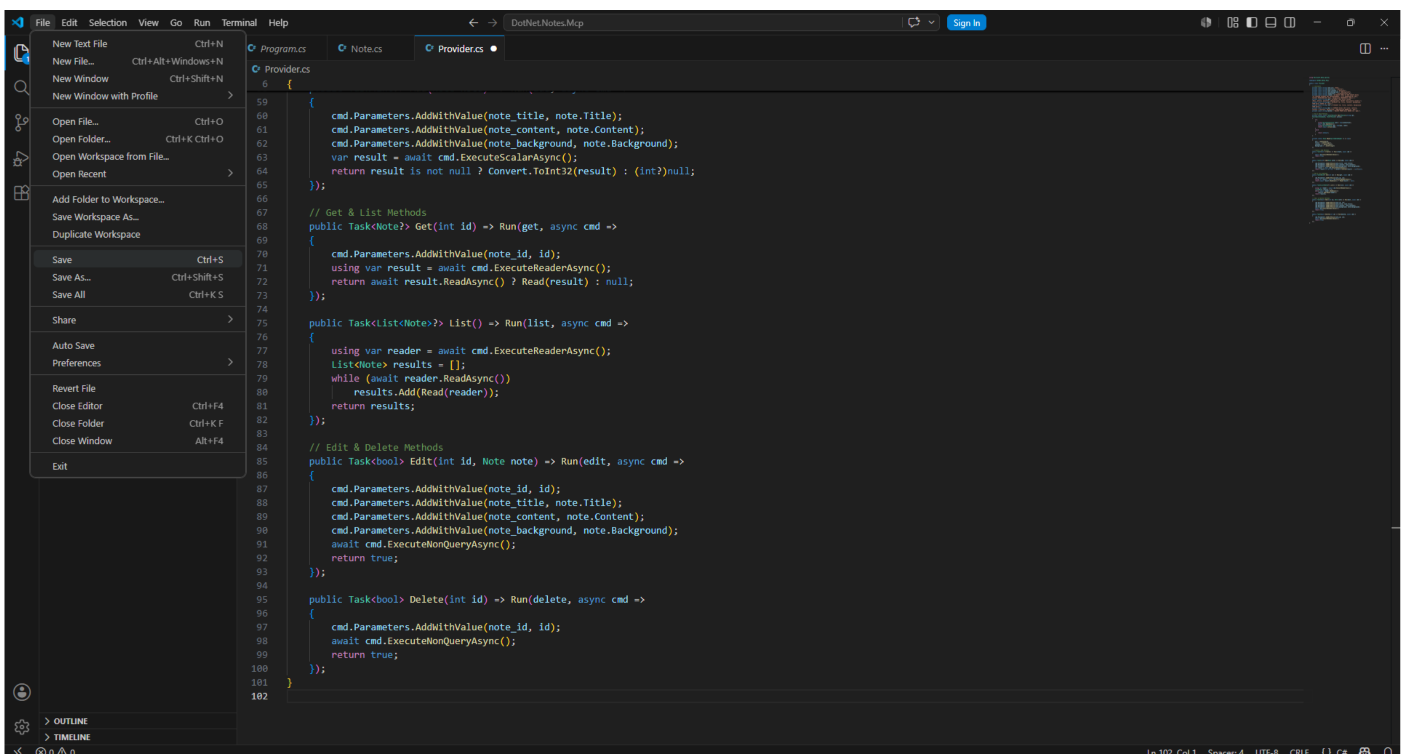
Information – **`Get`** is a **Method** which uses the **Command** to select a **Note** from the **Database** with the **Block** of **Code** using **`ExecuteReaderAsync`** to do this with **`AddWithValue`** to provide the **Id** of a **`Note`** to the **Command** and then using the **Method** of **`Read`** so the **Note** is returned, or if nothing or **`null`** came back or there is a problem then the **`default`** of **`null`** would be returned instead. **`List`** is a **Method** which uses the **Command** to select all **Notes** from the **Database** with the **Block** of **Code** using the **Method** of **`ExecuteReaderAsync`** to do this, then setting up a **`List`** of **`Note`** with no items, denoted by **`[]`**, then using **`while`**, it will **Loop** though the items or **Rows** returned from the **Table** in the **Database** and then use the **Method** of **`Read`** and **`Add`** to include those **Notes** in the **`List`** returned using **`return`**, if there is a problem then the **`default`** of **`null`** will be returned instead.

While still within **Provider.cs** in **Visual Studio Code** underneath **`// Edit & Delete Methods`** you need to *Copy* and *Paste* the following **Code**:

```
public Task<bool> Edit(int id, Note note) => Run(edit, async cmd =>
{
    cmd.Parameters.AddWithValue(note_id, id);
    cmd.Parameters.AddWithValue(note_title, note.Title);
    cmd.Parameters.AddWithValue(note_content, note.Content);
    cmd.Parameters.AddWithValue(note_background, note.Background);
    await cmd.ExecuteNonQueryAsync();
    return true;
});

public Task<bool> Delete(int id) => Run(delete, async cmd =>
{
    cmd.Parameters.AddWithValue(note_id, id);
    await cmd.ExecuteNonQueryAsync();
    return true;
});
```

Information – The **Method** of **`Edit`** will use the **Command** to update a **Note** in the **Database** provided with the values for the **Note** with **`AddWithValue`** and along with the **Method** of **`Delete`**, which will use the **Command** to delete a **Note**, will then use **`ExecuteNonQueryAsync`** and then **`return`** the value of **`true`**, should something go wrong then the **`default`** of **`false`** would be returned instead.

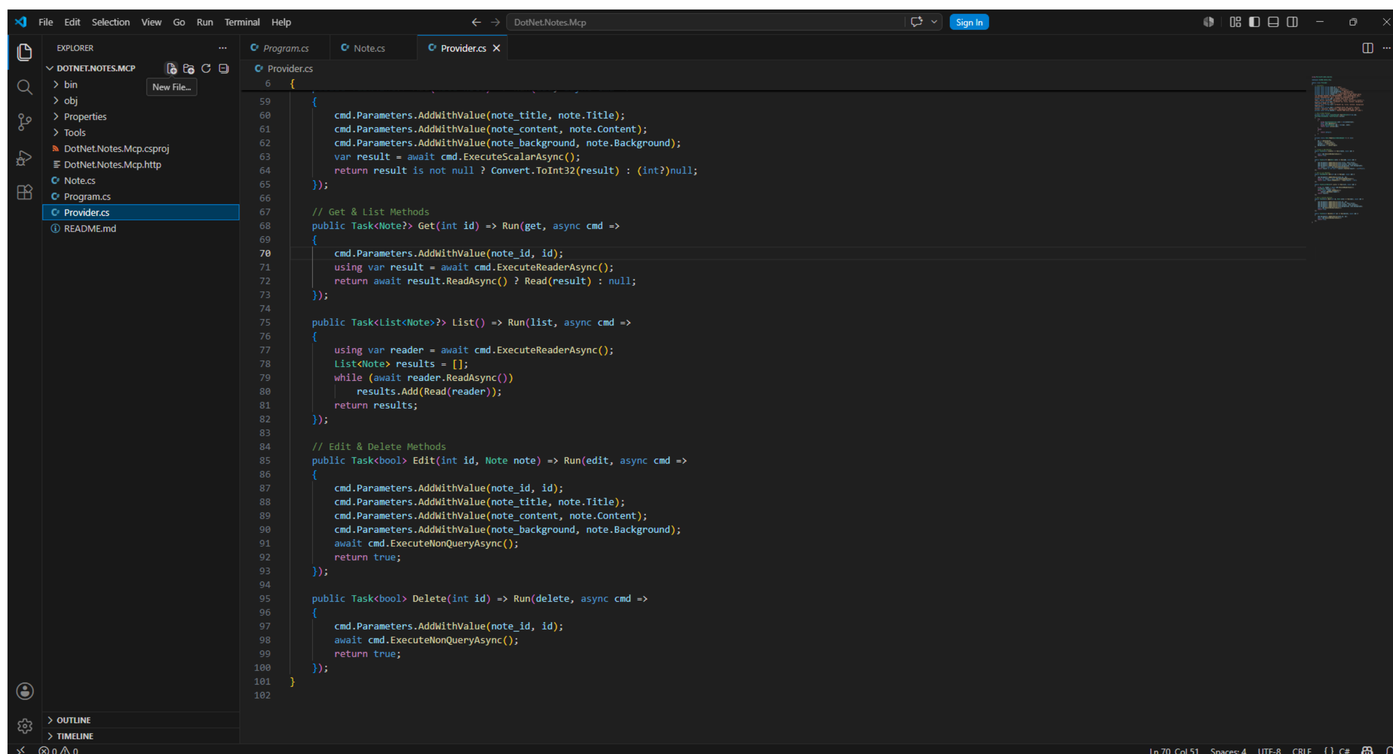
Next, within **Visual Studio Code** from the **Menu** select **File** and then **Save** as follows:



Information – If the **Terminal** on **Mac** or **Command Prompt** on **Windows** displays any **Errors**, then make sure that everything was entered correctly into **Provider.cs** by going over previous **Steps** to double check it matches what you have but once any corrections have been made and **Saved** or there are no **Errors** then the **Build** should proceed.

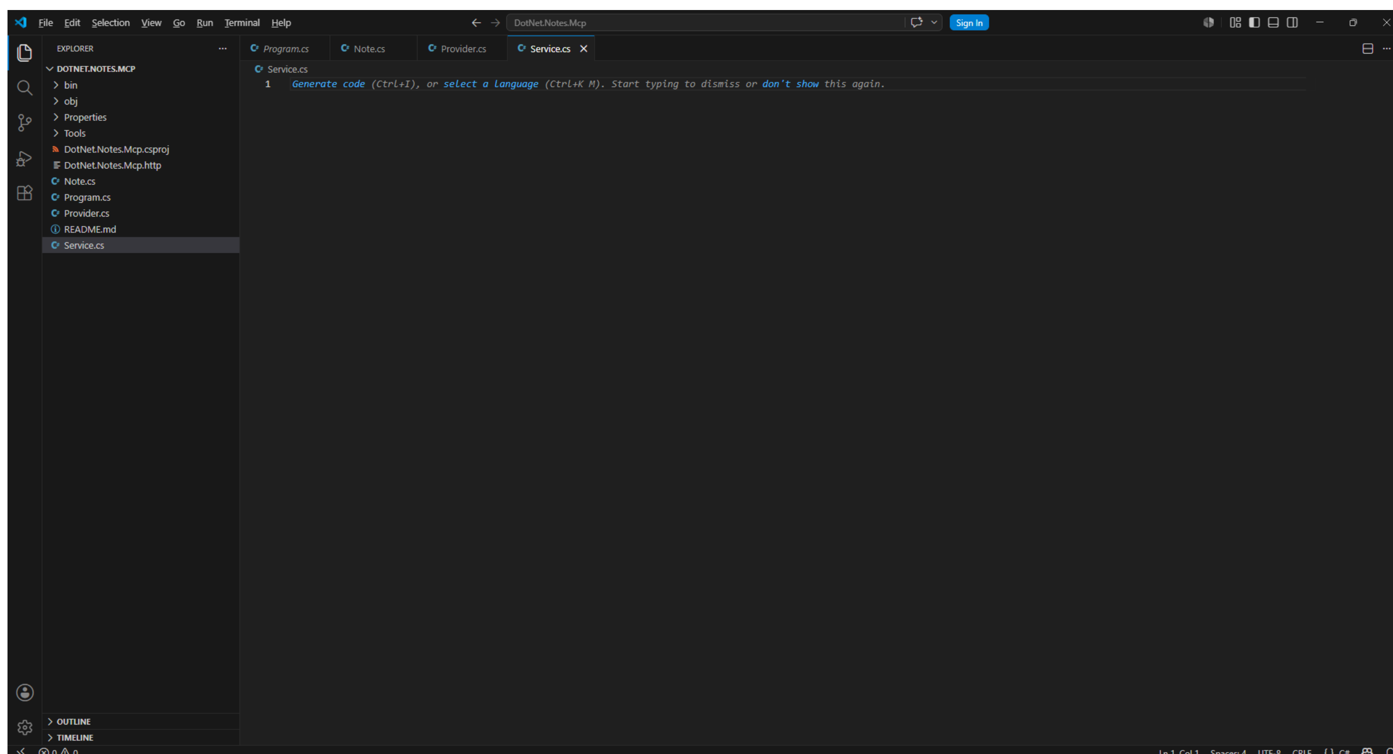
Service Class

In **Visual Studio Code** select **Provider.cs** in **Explorer** then choose **New File...** next to **DotNet.Notes.Mcp**.



Then *Type* in the following **Name** and press **Enter** after which you should see or select a blank **Service.cs** in **Explorer** within **Visual Studio Code**.

```
Service.cs
```



Then within **Visual Studio Code** in **Service.cs** you need to *Copy* and *Paste* the following **Code**:

```
using System.ComponentModel.DataAnnotations;
using System.Security;
using System.Text;

namespace DotNet.Notes.Mcp;

public class Service(Provider provider)
{
    // Constants & Read Only Members

    // Response, Resolve & Validate Methods

    // Backgrounds, List & Get Methods

    // Add Method

    // Edit Method

    // Image Method

    // Delete Method
}
```

Information – This forms the outline of a **`class`** for **`Service`** to group together **Code**, the first line is a **`using`** of **`System.ComponentModel.DataAnnotations`** needed for **Annotations** used for **Validation** and the rest are for other functionality needed. Then there is a **`namespace`** for **DotNet.Notes.Mcp** which is followed by the **`class`** which is defined for **`Service`** providing the **`class`** of **`Provider`** with **Dependency Injection** in a **Primary Constructor**, which is used to provide anything needed for a **`class`** in a concise readable manner. There are also **Comments** which are the lines starting with **`//`** which will help you place **Code** from the next few **Steps** of the **Workshop**.

Next within **Visual Studio Code** in **Service.cs** underneath **`// Constants & Read Only Members`** you need to *Copy* and *Paste* the following **Code**:

```
private const char hash = '#';
private const string separator = ", ";

private static readonly (string Name, string Value)[] bg_options =
[
    ("Red",     "#ff4947"),
    ("Orange", "#f57900"),
    ("Yellow", "#ffc476"),
    ("Green",   "#6dc0a4"),
    ("Blue",    "#6f9acd"),
    ("Indigo", "#833db8"),
    ("Violet", "#c693c2")
];
private static readonly List<string> bg_names =
    [.. bg_options.Select(o => o.Name)];
private static readonly List<string> bg_values =
    [.. bg_options.Select(o => o.Value)];
private static readonly Dictionary<string, string> bg_to_value =
    bg_options.ToDictionary(o => o.Name, o => o.Value,
    StringComparer.OrdinalIgnoreCase);
private static readonly Dictionary<string, string> bg_to_name =
    bg_options.ToDictionary(o => o.Value, o => o.Name,
    StringComparer.OrdinalIgnoreCase);
private static readonly string bg_output =
    string.Join(separator, bg_names);
```

Information – This **Code** defines **Constants** or **`const`** which is something that does not change when your application is running and they are **`private`** as they are only used within the **`class`** of **`Service`** for values such as a **`char`**, which is a single character in some text needed when dealing with colour values and a **`string`** for helping with outputting colours. There are also **Members** that are **`static`** and **`readonly`** which serve a similar purpose to **Constants** including **`bg_options`** for colour names and HTML colour values used for the **Background** of a **Note** along with a **`List`** for names of **`bg_names`** and colour values as **`bg_values`**. There is also a **`Dictionary`** of **`bg_to_value`** that is used to convert from name to a colour value and there is **`bg_to_name`** to convert from a colour value to name plus a **`string`** of **`bg_output`** to output names separated by the **`separator`**.

Then within **Visual Studio Code** in **Service.cs** underneath **`// Response, Resolve & Validate Methods`** you need to *Copy* and *Paste* the following **Code**:

```
private static object Response(Note note) => new
{
    id = note.Id,
    title = note.Title,
    content = note.Content,
    background = note.Background,
    backgroundName = bg_to_name.TryGetValue(note.Background ??
        string.Empty, out var name) ? name : null
};

private static string? Resolve(string? background, out string? message)
{
    message = null;
    var bg = background ?? string.Empty;
    if (bg_to_value.TryGetValue(bg, out var value))
        return value;
    if (bg.StartsWith(hash) && bg_values.Contains(bg))
        return bg;
    message = $"Unsupported background colour. Use one of: {bg_output}";
    return null;
}

private static object? Validate(Note note)
{
    var results = new List<ValidationResult>();
    return Validator.TryValidateObject(note,
        new ValidationContext(note), results, true)
    ? null : new
    {
        error = string.Join(separator,
            results.Select(r => r.ErrorMessage))
    };
}
```

Information – The **Method** of **`Response`** will take a **`Note`** and convert it to something an **Agent** using **AI** will be able to understand, including mapping the **Background** of a **`Note`** to the name using **`bg_to_name`** with **`TryGetValue`** for a **`backgroundName`** or with **`null`** when not valid for any reason, along with the other **Properties**. The **Method** of **`Resolve`** will take an input of **`background`** then use **`if`** which does anything within should the value be **`true`** using **`bg_to_value`** and **`TryGetValue`** it will then try to map the name of background provided by an **Agent** using **AI**, or **`if`** provided a HTML colour, which begins with **#** defined as **`hash`** and, indicated with **`&&`**, is one of the colour values from **`bg_values`**, otherwise it will set the **`message`** to indicate an unsupported background colour including colours supported from **`bg_output`**. The **Method** of **`Validate`** will use **`Validator.TryValidateObject`** to validate a **`Note`** which will use the **Attributes**, and any issues will be set to the **`List`** of **`ValidationResult`** to be returned, using **`separator`** as an **`error`** to be returned to the **Agent** using **AI**. All these **Methods** are **`private`** as are not used outside this **`class`**.

Next within **Visual Studio Code** in **Service.cs** underneath **`// Backgrounds, List & Get Methods`** you need to *Copy* and *Paste* the following **Code**:

```
public object Backgrounds() =>
bg_options.Select(o => new
{
    name = o.Name,
    value = o.Value
}).ToList();

public async Task<object> List()
{
    var notes = await provider.List();
    return notes?.Select(Response).ToList() ?? [];
}

public async Task<object> Get(int id)
{
    var note = await provider.Get(id);
    return note is not null ? Response(note) : new
    {
        error = $"Note with id {id} not found."
    };
}
```

Information – These **Methods** are **`public`** as they will be used outside the **Class**, with the **Method** of **`Backgrounds`** used to output **`bg_options`** in a way an **Agent** using **AI** can understand. The **Method** of **`List`** is used with the **Method** of **`List`** from the **`Provider`** to get all **Notes** and use **`Response`** to output them as needed. The **Method** of **`Get`** is used to obtain a specific **Note** by **Id** or if not will return an **`error`**.

Next within **Visual Studio Code** in **Service.cs** underneath `// Add Method` you need to *Copy* and *Paste* the following **Code**:

```
public async Task<object> Add(string? title, string? content, string? background)
{
    var note = new Note
    {
        Title = title,
        Content = content,
        Background = background
    };
    if (Validate(note) is { } validation)
        return validation;
    var resolved = Resolve(background, out var message);
    if (resolved is null) return new
    {
        error = message
    };
    note.Background = resolved;
    if (!await provider.Create() || await provider.Add(note) is null) return new
    {
        error = "Failed to create note."
    };
    return Response(note);
}
```

Information – **Method** of **`Add`** is **`public`** and is used by an **Agent** using **AI** to create a **Note**, by providing a **`title`**, **`content`** and **`background`** which is then used to create a **`Note`**, which is **Validated** with the **Method** of **`Validate`**, if validation fails this will be returned with **`return`**. The **`background`** will be resolved with **`Resolve`**, should this fail then an **`error`** will be returned with the **`message`**, otherwise the **Property** of **`Background`** will be set. This **Method** will then use **`Create`** along with **`Add`** from **`Provider`** to create **Table** in the **Database** if needed and add the new **Note**, if there is a problem when calling **Methods** of **`Provider`** such as **`Create`** returning **`false`** or **`Add`** returning **`null`** then an **`error`** will be returned, otherwise the new **Note** will be returned with **`return`** using **`Response`**, where **`||`** means **Or** and **`!`** means **Not**.

Next within **Visual Studio Code** in **Service.cs** underneath **`// Edit Method`** you need to *Copy* and *Paste* the following **Code**:

```
public async Task<object> Edit(int id, string? title,
    string? content, string? background)
{
    var note = await provider.Get(id);
    if (note is null) return new
    {
        error = $"Note with id {id} not found."
    };
    note.Title = !string.IsNullOrWhiteSpace(title) ?
        title : note.Title;
    note.Content = !string.IsNullOrWhiteSpace(content) ?
        content : note.Content;
    note.Background = !string.IsNullOrWhiteSpace(background) ?
        background : note.Background;
    if (Validate(note) is { } validation)
        return validation;
    if (!string.IsNullOrEmpty(background))
    {
        var resolved = Resolve(background, out var message);
        if (resolved is null) return new
        {
            error = message
        };
        note.Background = resolved;
    }
    if (!await provider.Edit(id, note)) return new
    {
        error = "Failed to edit note."
    };
    return Response(note);
}
```

Information – This **Method** of **`Edit`** is **`public`** and is used by an **Agent** using **AI** to edit a **Note** which will provide the **`id`**, **`title`**, **`content`** and **`background`**, the **Method** will obtain the existing **Note** by **Id**, which if not found then an **`error`** will be returned. Otherwise, the intent to change the value by checking if one of those values was providing by seeing if it is not a **`null`**, an empty **`string`** or one with just whitespace using the **Method** of **`string.IsNullOrWhiteSpace`** which would return **`true`** if this was the case and the **`!`**, which represents **Not**, will turn it into **`false`** to indicate that value wasn't provided. Any values are not provided will be assigned to the original such as **`Title`**, **`Content`** or **`Background`** of the **Note**. The edited **Note** is then **Validated** with the **Method** of **`Validate`**, if this fails this will be returned with **`return`**, if not the **`background`** will be resolved with **`Resolve`**, should this fail then an **`error`** will be returned with the **`message`**, otherwise the **Property** of **`Background`** will be set, then the **Method** will use **`Edit`** from **`Provider`** to update the existing **Note**, should there be a problem an **`error`** will be returned, otherwise the edited **Note** will be returned with **`return`** using **`Response`**.

Next within **Visual Studio Code** in **Service.cs** underneath `// Image Method` you need to *Copy* and *Paste* the following **Code**:

```
public async Task<string> Image(int id, bool isDataUri = false)
{
    var note = await provider.Get(id);
    if (note is null)
        return $"Note image with id {id} not found.";
    string title = SecurityElement.Escape(note.Title) ?? string.Empty;
    string content = SecurityElement.Escape(note.Content) ?? string.Empty;
    string svg = $"""
    <svg viewBox="0 0 466.67 466.67" xmlns="http://www.w3.org/2000/svg">
        <path d="M 0,233.33333 V 0 H 233.33333 466.66666 V 183.69805 367.3961
        l -57.66667,49.60273 -57.66666,49.60274 -175.66667,0.0325 L 0,466.66666 Z"
        fill="{note.Background}" />
        <path d="m 351.70982,366.69855 114.94703,0.77358 -115.33333,99.12108 z"
        fill="rgba(255,255,255,0.3)" />
        <rect x="0" y="0" width="466.67" height="70" fill="rgba(0,0,0,0.15)" />
        <text x="233.33" y="47" font-family="Segoe Script, cursive"
        text-anchor="middle" font-size="32" fill="#333">{title}</text>
        <foreignObject x="15" y="80" width="442" height="272">
            <div xmlns="http://www.w3.org/1999/xhtml"
            style="font-family: 'Segoe Script', cursive; font-size: 24px;
            color: #333; max-width: 442; word-wrap: break-word;">{content}</div>
        </foreignObject>
    </svg>
    """;
    return isDataUri ? $"data:image/svg+xml;base64,{Convert.ToBase64String(
        Encoding.UTF8.GetBytes(svg))}"
    : svg;
}
```

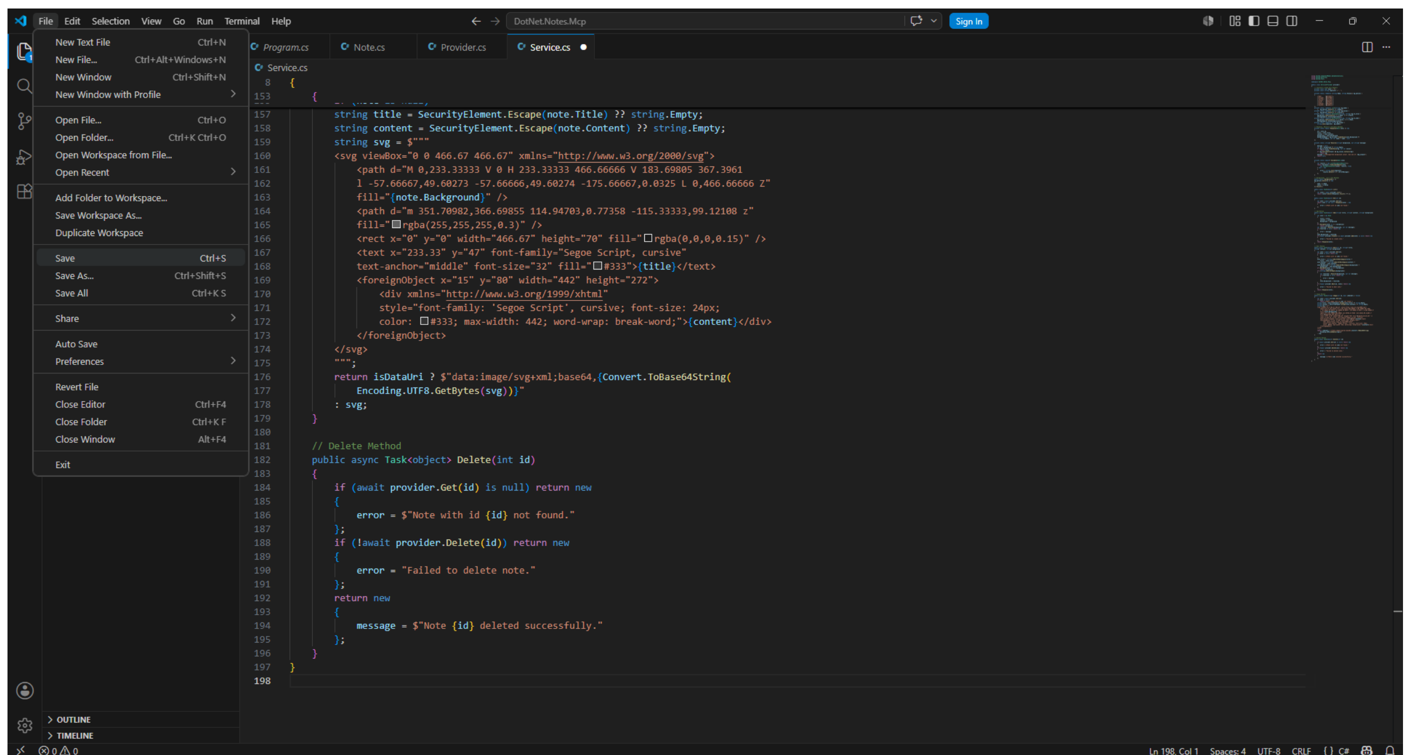
Information – The **Method** of **`Image`** is **`public`** and is used by an **Agent** using **AI** to get the image for a **Note** represented by an **SVG** by providing the **Id** for the **Note**, or this can be returned as a **Base-64** encoded **Data URI** by optionally providing **`true`** to **`isDataUri`**. The **Method** will check for an existing **Note** with **`Get`** from **`Provider`**, if not found an **`error`** will be returned. Otherwise, **`SecurityElement.Escape`** will be used to ensure the **`Title`** and **`Content`** of the **Note** can be displayed correctly as **`title`** and **`content`**. There is **`string`** of **`svg`** that represents an **SVG** for the **Note**, which is a scalable image format, using **`title`** and **`content`** of the **Note** along with **`Background`**. This **Method** will check the value of **`isDataUri`**, if this is **`true`** then will get the **Bytes** of the **SVG** using **`Encoding.UTF8.GetBytes`** and **`Convert.ToBase64String`** to **Base-64** encode and returned as a **Data URI**, otherwise if **`isDataUri`** is **`false`** then the **SVG** is returned.

While still within **Service.cs** in **Visual Studio Code** underneath **`// Delete Method`** you need to *Copy* and *Paste* the following **Code**:

```
public async Task<object> Delete(int id)
{
    if (await provider.Get(id) is null) return new
    {
        error = $"Note with id {id} not found."
    };
    if (!await provider.Delete(id)) return new
    {
        error = "Failed to delete note."
    };
    return new
    {
        message = $"Note {id} deleted successfully."
    };
}
```

Information – This **Method** of **`Delete`** is **`public`** and is used by an **Agent** using **AI** to delete a **Note** by providing an **Id** for the **Note**, it will use **`Get`** from **`Provider`** to check if the **Note** exists, if not an **`error`** will be returned. Otherwise, it will use **`Delete`** from **`Provider`** to delete the **Note**, if there is a problem an **`error`** will be returned, or if the delete was successful a **`message`** will be returned.

Next, within **Visual Studio Code** from the **Menu** select **File** and then **Save** as follows:



Information – If the **Terminal** on **Mac** or **Command Prompt** on **Windows** displays any **Errors**, then make sure that everything was entered correctly into **Service.cs** by going over previous **Steps** to double check it matches what you have but once any corrections have been made and **Saved** or there are no **Errors** then the **Build** should proceed.

Tools Class

In **Visual Studio Code** select **Service.cs** in **Explorer** then choose **New File...** next to **DotNet.Notes.Mcp**.

Then *Type* in the following **Name** and press **Enter** after which you should see or select a blank **Tools.cs** in **Explorer** within **Visual Studio Code**.

```
Tools.cs
```

Then within **Visual Studio Code** in **Tools.cs** you need to *Copy* and *Paste* the following **Code**:

```
using ModelContextProtocol.Server;
using System.ComponentModel;

namespace DotNet.Notes.Mcp;

public class Tools(Service service)
{
    // List Backgrounds, List Notes & Get Note Methods

    // Add Note & Edit Note Methods

    // Note Image & Delete Note Methods

}
```

Information – This forms the outline of a `class` for `Tools` to group together **Code**, the first line is a `using` of `ModelContextProtocol.Server` needed for **MCP** and the other is for functionality needed. There is a `namespace` for **DotNet.Notes.Mcp** followed by the `class` which is defined for `Tools` providing the `class` of `Service` to this `class` with **Dependency Injection** in a **Primary Constructor** which is used to provide anything needed for a `class` in a concise readable manner. There are also **Comments** which are lines starting with `//` which will help you place **Code** from the next few **Steps** of the **Workshop**.

Next within **Visual Studio Code** in **Tools.cs** underneath `// List Backgrounds, List Notes & Get Note Methods` you need to *Copy* and *Paste* the following **Code**:

```
[McpServerTool]
[Description("Get the list of available background colours for notes")]
public object ListBackgrounds() =>
    service.Backgrounds();

[McpServerTool]
[Description("Get a list of all notes with their details")]
public Task<object> ListNotes() =>
    service.List();

[McpServerTool]
[Description("Get a specific note by its ID")]
public Task<object> GetNote([Description("The ID of the note to get")] int id) =>
    service.Get(id);
```

Information – This **Code** defines some of the **Tools** to be used via **MCP** in an **Agent** using **AI**, they have an **Attribute** of `McpServerTool` which indicates it is a **Tool** and another to give **Context** of `Description`**.** The **Method** of `ListBackgrounds` uses the **Method** of `Backgrounds` from `Service`. **Method** of `ListNotes` uses the **Method** of `List` from `Service`, with the `Description` with **Context** indicating their purpose to an **Agent** using **AI** knows how to use them. **Method** of `GetNote` provides a **Parameter** for the **Id** of a **Note** which has an **Attribute** for `Description` to indicate what needs to be provided by the **Agent** using **AI**.

Then within **Visual Studio Code** in **Tools.cs** underneath **`// Add Note & Edit Note Methods`** you need to *Copy* and *Paste* the following **Code**:

```
[McpServerTool]
[Description("Create a new note")]
public Task<object> AddNote(
[Description("The title of the note (max 50 characters)")] string title,
[Description("The content of the note (max 255 characters)")] string content,
[Description("The background colour (from available colours)")] string background) =>
    service.Add(title, content, background);

[McpServerTool]
[Description("Update an existing note")]
public Task<object> EditNote(
[Description("The ID of the note to update")] int id,
[Description("The new title (optional)")] string? title = null,
[Description("The new content (optional)")] string? content = null,
[Description("The new background colour (optional)")] string? background = null) =>
    service.Edit(id, title, content, background);
```

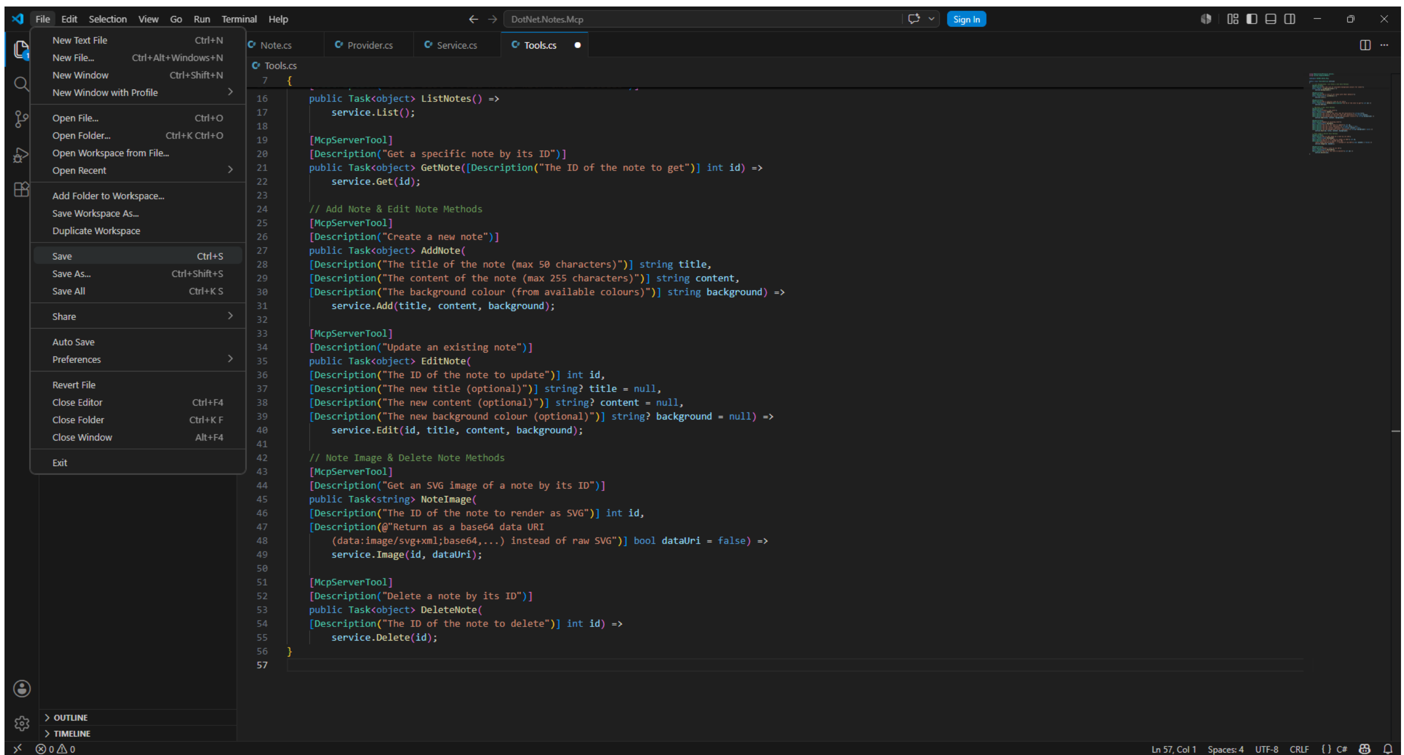
Information – The **Method** of **`AddNote`** is a **Tool** used by the **Agent** using **AI** to add a **Note** featuring an **Attribute** of **`Description`** that describes what it is for and then an **Attribute** for each of the **Parameters** of **`title`**, **`content`** and **`background`** indicating what it needs and acceptable values, it uses the **Method** of **`Add`** in **`Service`** where these are **Validated** if incorrect values are provided. The **Method** of **`EditNote`** is a **Tool** that uses the **Method** of **`Edit`** in **`Service`** with **Attributes** of **`Description`** to indicate purpose and values that can be provided by the **Agent** using **AI**.

While still within **Tools.cs** in **Visual Studio Code** underneath **`// Note Image & Delete Note Methods`** you need to *Copy* and *Paste* the following **Code**:

```
[McpServerTool]
[Description("Get an SVG image of a note by its ID")]
public Task<string> NoteImage(
[Description("The ID of the note to render as SVG")] int id,
[Description(@"Return as a base64 data URI
    (data:image/svg+xml;base64,...) instead of raw SVG")] bool dataUri = false) =>
    service.Image(id, dataUri);

[McpServerTool]
[Description("Delete a note by its ID")]
public Task<object> DeleteNote(
[Description("The ID of the note to delete")] int id) =>
    service.Delete(id);
```

Information – The **Method** of **`NoteImage`** is a **Tool** that enables the **Agent** using **AI** to get an image of a **Note** as a raw **SVG** or as a **Base-64** encoded **Data URI** with the **Attributes** for **`Description`** indicating usage options and uses **`Image`** from **`Service`**. The **Method** of **`DeleteNote`** is the final **Tool** which allows the **Agent** using **AI** to delete a **Note** using **`Delete`** from **`Service`**. Normally performing destructive operations such as delete, would use confirmation, but has been excluded for simplicity.

Next, within **Visual Studio Code** from the **Menu** select **File** and then **Save** as follows:



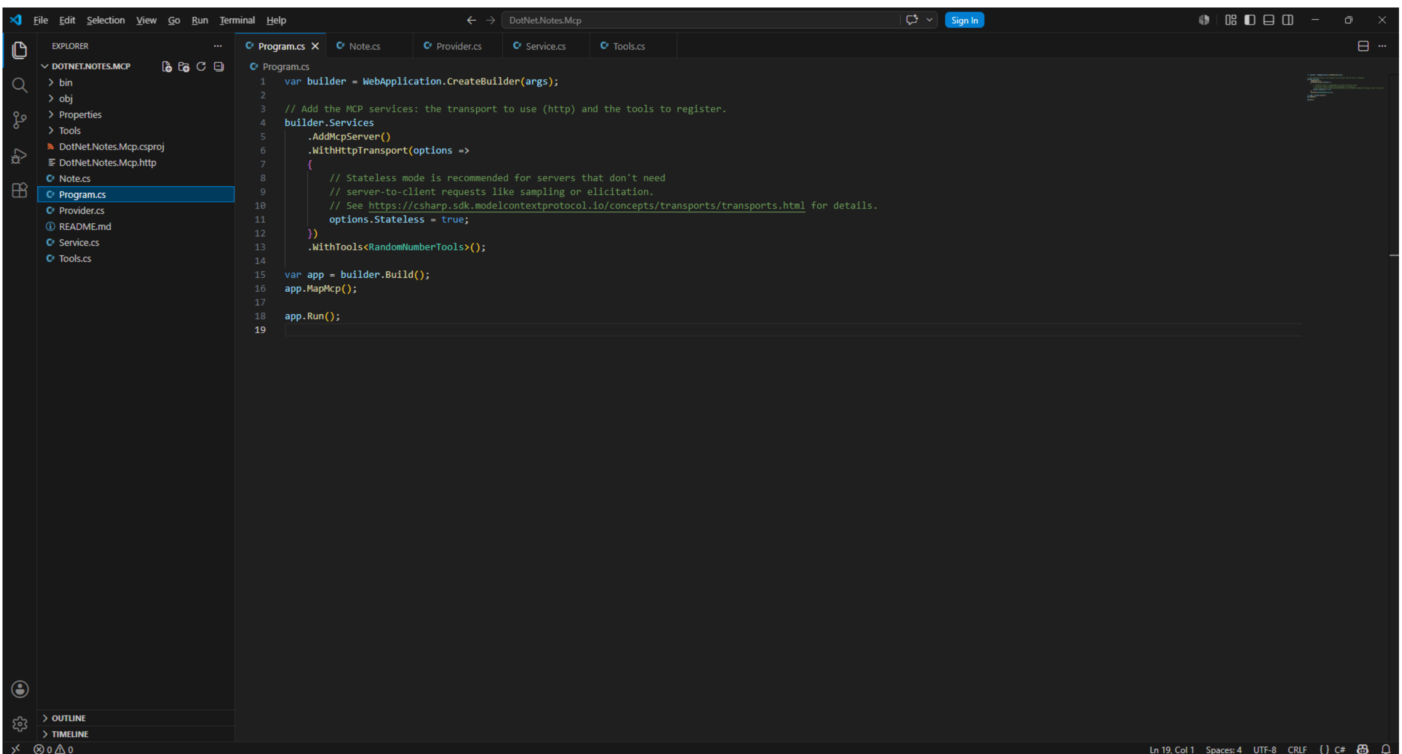
Information – If the **Terminal** on **Mac** or **Command Prompt** on **Windows** displays any **Errors**, then make sure that everything was entered correctly into **Tools.cs** by going over previous **Steps** to double check it matches what you have but once any corrections have been made and **Saved** or there are no **Errors** then the **Build** should proceed.

Once the **Tools** have been added, that completes the process of adding the **Classes** of **`Provider`**, **`Service`** and **`Tools`** to the **Project** for the **Workshop**.

CC BY SA

Update Project

In **Visual Studio Code** select **Program.cs** in **Explorer** as follows:



Information – **Program.cs** is where an application is **Initialised** for features in **.NET**, **Templates** or **Packages** along with **Services** which are set up by default and an example **Tool**.

With **Program.cs** selected in **Explorer** you will need to add a **`using`** statement, so above the line of **`var builder = WebApplication.CreateBuilder(args);`** you need to *Copy* and *Paste* the following **Using**:

```
using DotNet.Notes.Mcp;
```

Information – This **`using`** statement will include the functionality for the **`class`** of **`Provider`**, **`Service`** and **`Tools`** so they can be used in **Program.cs**.

In **Program.cs** below the line of **`builder.Services`** you will need to *Copy* and *Paste* the following **Code**:

```
.AddScoped<Provider>()
.AddScoped<Service>()
```

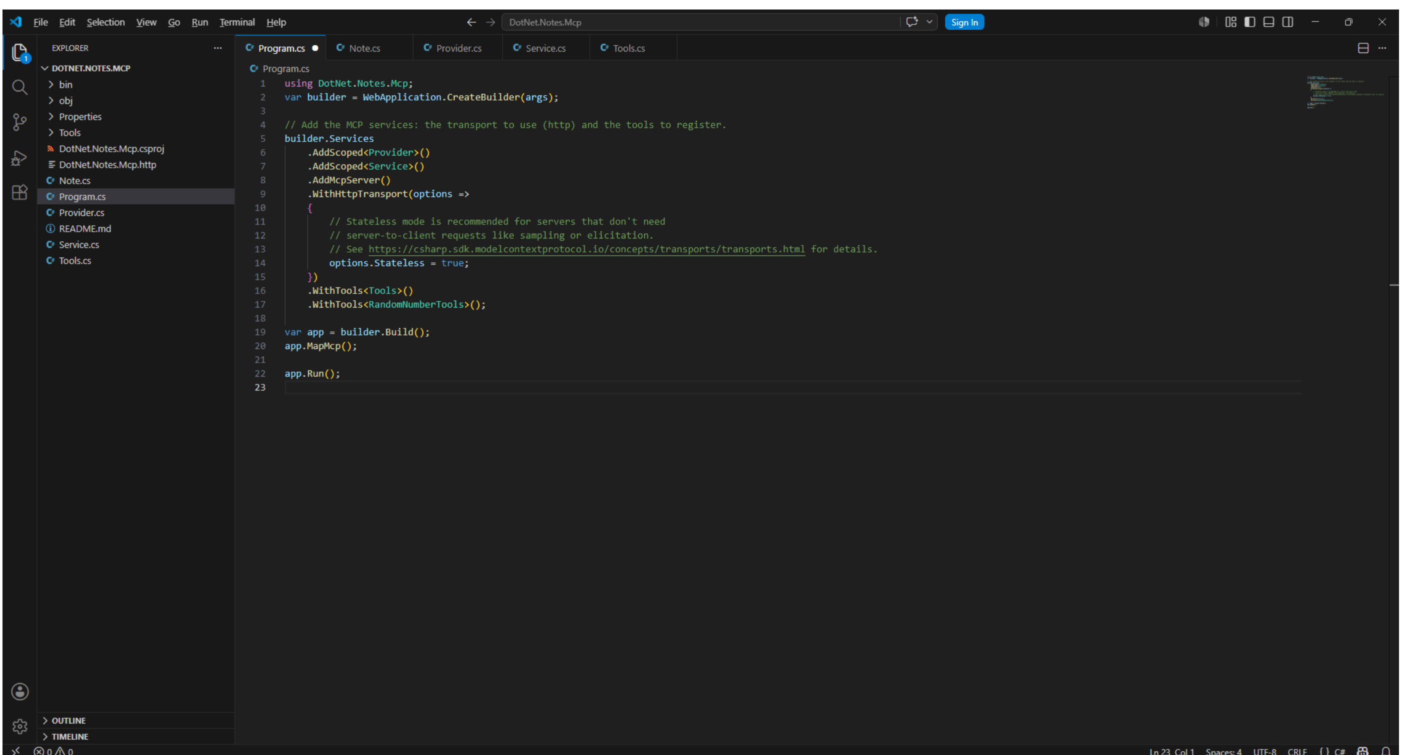
Information – This **Code** will register the **`class`** of **`Provider`** and **`Service`** for **Dependency Injection**, also notice unlike many other lines of **Code** there is no semi-colon or **`;`** as they are part of a chain of **Methods**.

While still within **Program.cs** in **Visual Studio Code** above the line **`.WithTools<RandomNumberTools>();`** you will need to *Copy* and *Paste* the following **Code**:

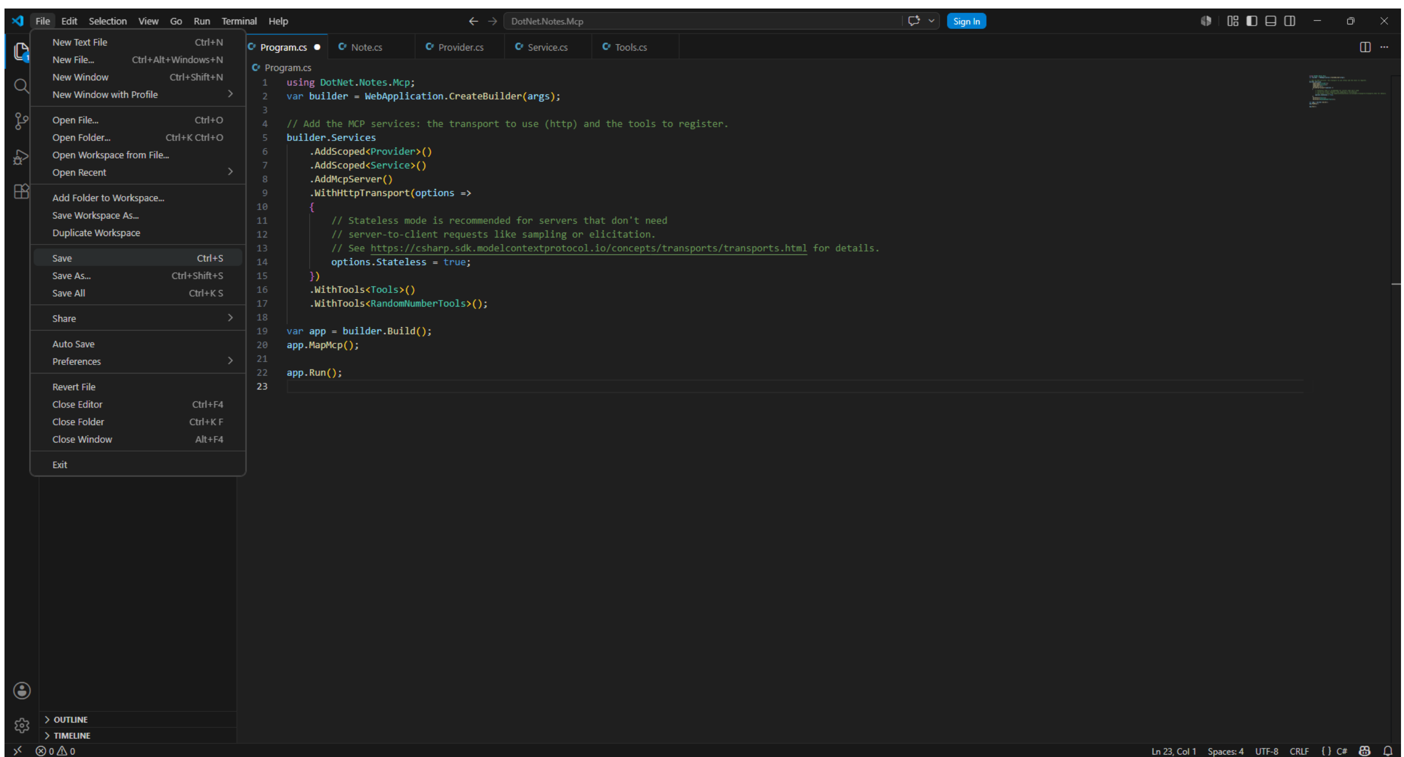
```
.WithTools<Tools>()
```

Information – This **Code** with register the **`class`** of **`Tools`** for use via **Model Context Protocol** or **MCP** for use by an **Agent** using **AI**, also notice unlike many other lines of **Code** there is no semi-colon or **`;`** it is part of a chain of **Methods**.

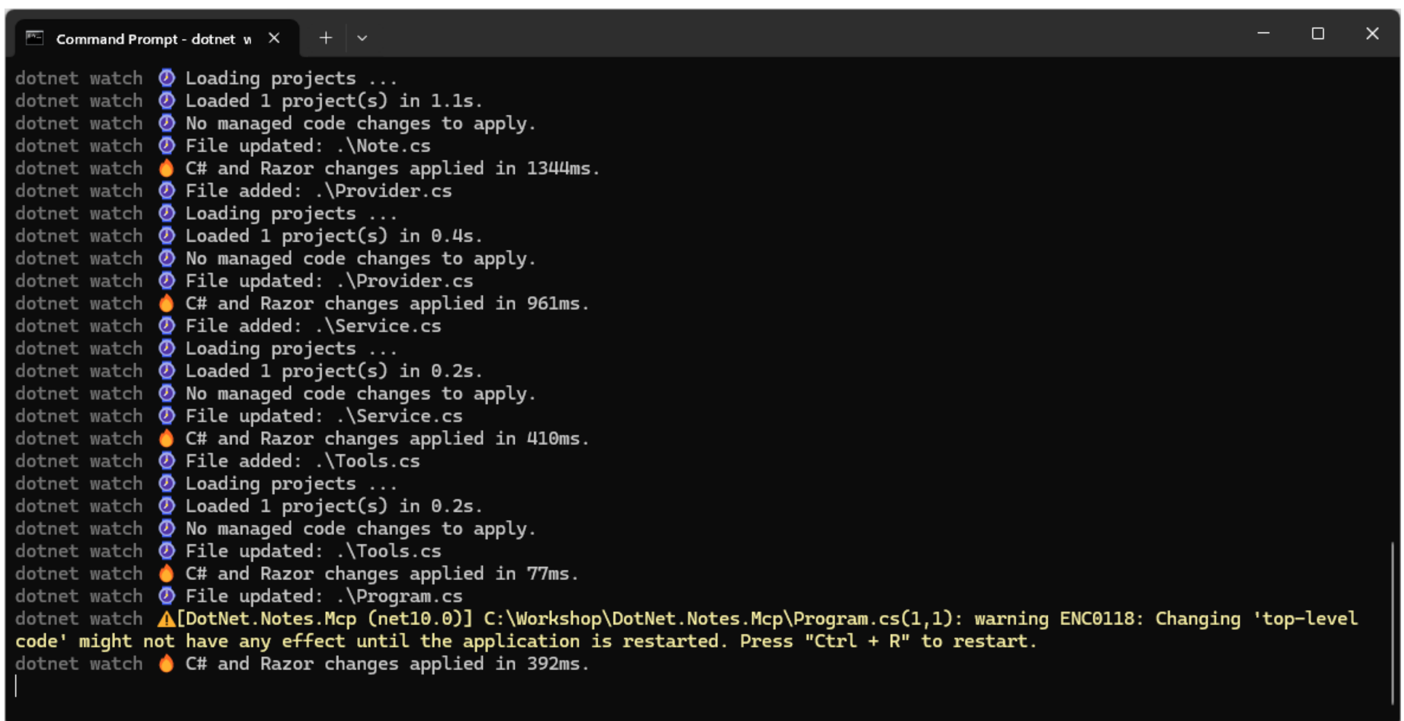
Once you have updated **Program.cs** with **Usings** and **Code** in **Visual Studio Code** it should be as follows:



While still within **Visual Studio Code** from the **Menu** select **File** and then **Save** as follows:

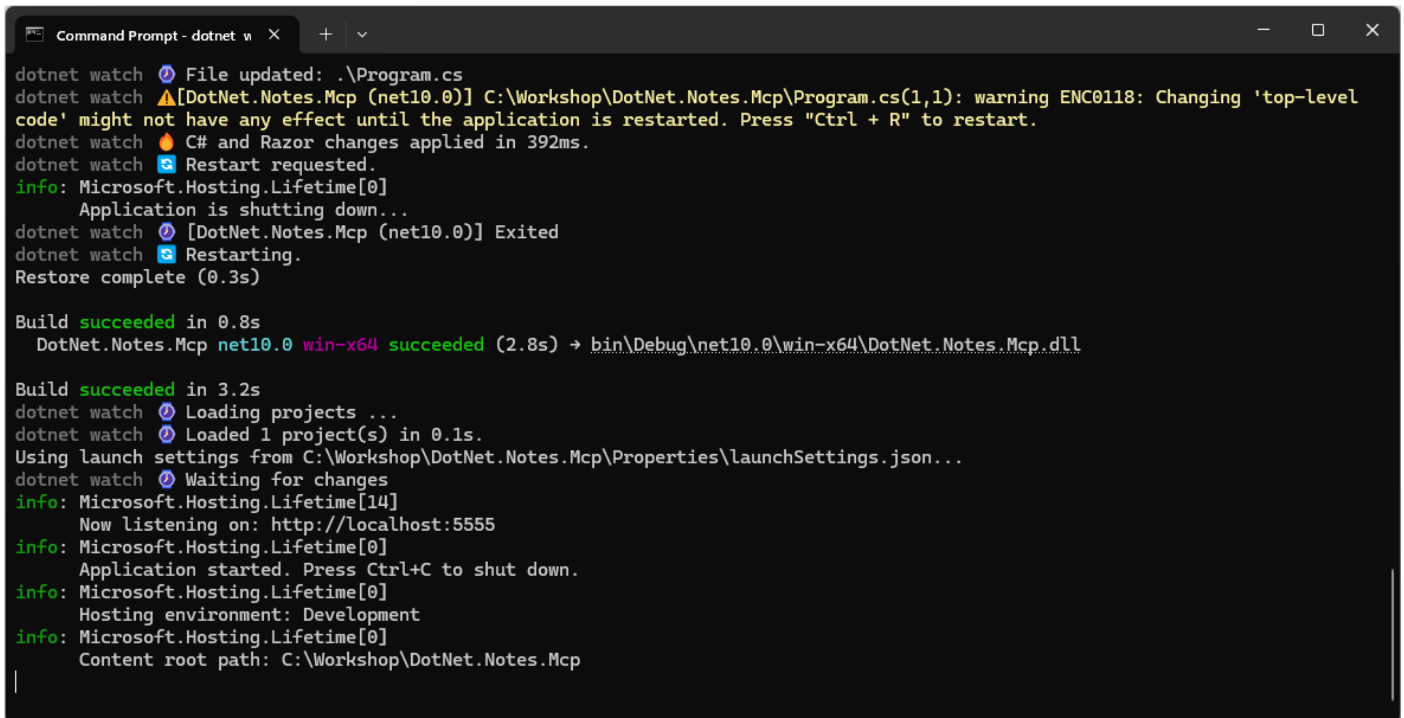


Once **Saved** then return to the **Terminal** on **Mac** or **Command Prompt** on **Windows** and you will see a **Warning** with the following message, **Changing `top-level-code` might not have any effect until the application is restarted. Press "Ctrl + R" to restart** as follows:



Information – If the **Terminal** on **Mac** or **Command Prompt** on **Windows** displays any **Errors** in addition to this **Warning** then make sure that everything was entered correctly into **Program.cs** by going over previous **Steps** to double check it matches what you have but once any corrections have been made and **Saved** or there are no **Errors** then the **Build** should proceed.

While still in the **Terminal** on **Mac** or **Command Prompt** on **Windows** press the **Keys** for **Ctrl** and **R** together which should **Restart** the **Project** as follows:



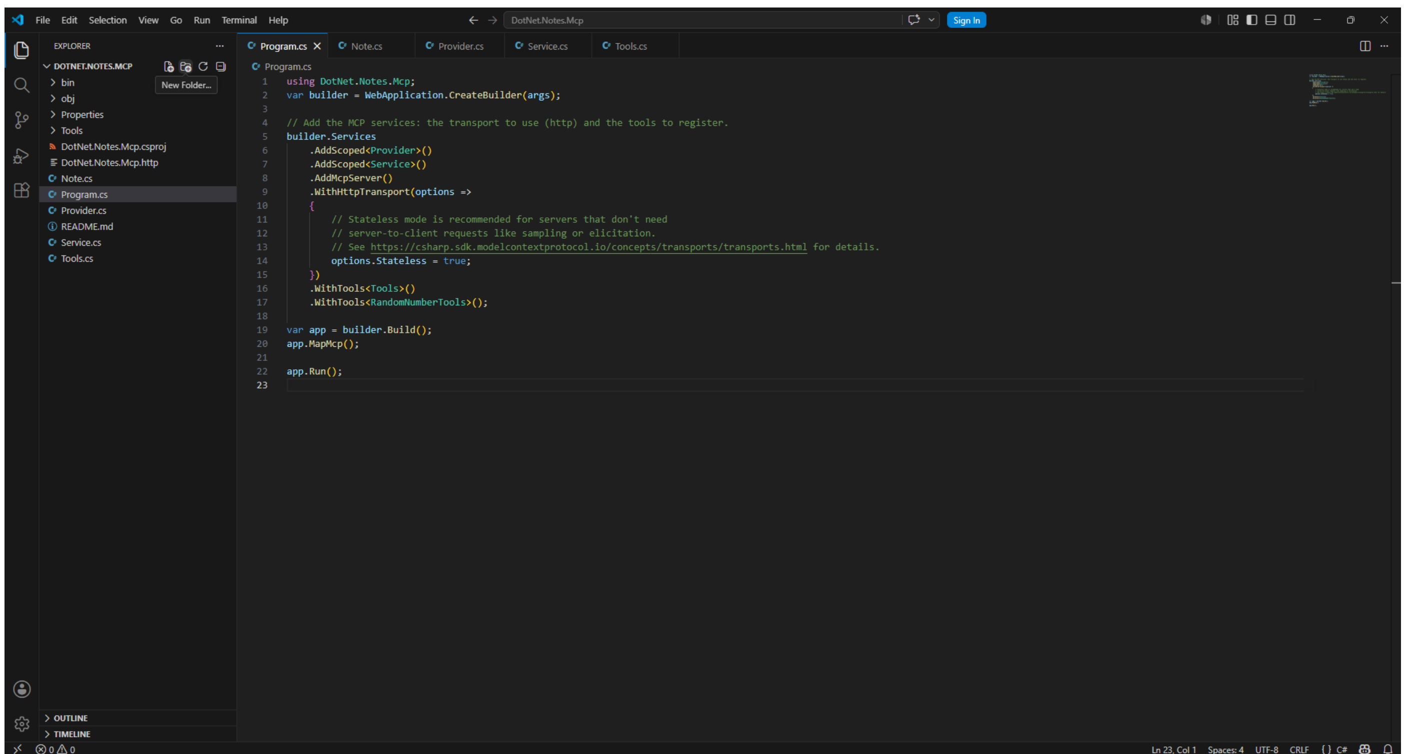
Information – If the **Terminal** on **Mac** or **Command Prompt** on **Windows** displays any **Errors** after the **Restart** then make sure that everything was entered correctly into **Program.cs** by going over previous **Steps** to double check it matches what you have but once any corrections have been made and **Saved** or there are no **Errors** then the **Build** should proceed.

Don't **Close** the **Command Prompt** on **Windows** or **Terminal** on **Mac** but if **Closed** then you need to go to **Finder**, search for **Terminal** and then select it to **Open** it, or if you **Closed** the **Command Prompt** on **Windows**, go to **Start**, search for **Command Prompt** and then select it to **Open** it. Then once opened you need to change directory using **cd** to the location for your **Project**, for example *cd DotNet.Notes.Mcp* and then you need to type **`dotnet watch --urls http://localhost:5555`** followed by **Enter**.

Once the **Project** has been **Restarted**, that completes the **Code** needed for the **Project** in the **Workshop**.

Configure Project

In **Visual Studio Code** select **Program.cs** in **Explorer** and select **New Folder...** next to **DotNet.Notes.Mcp**:

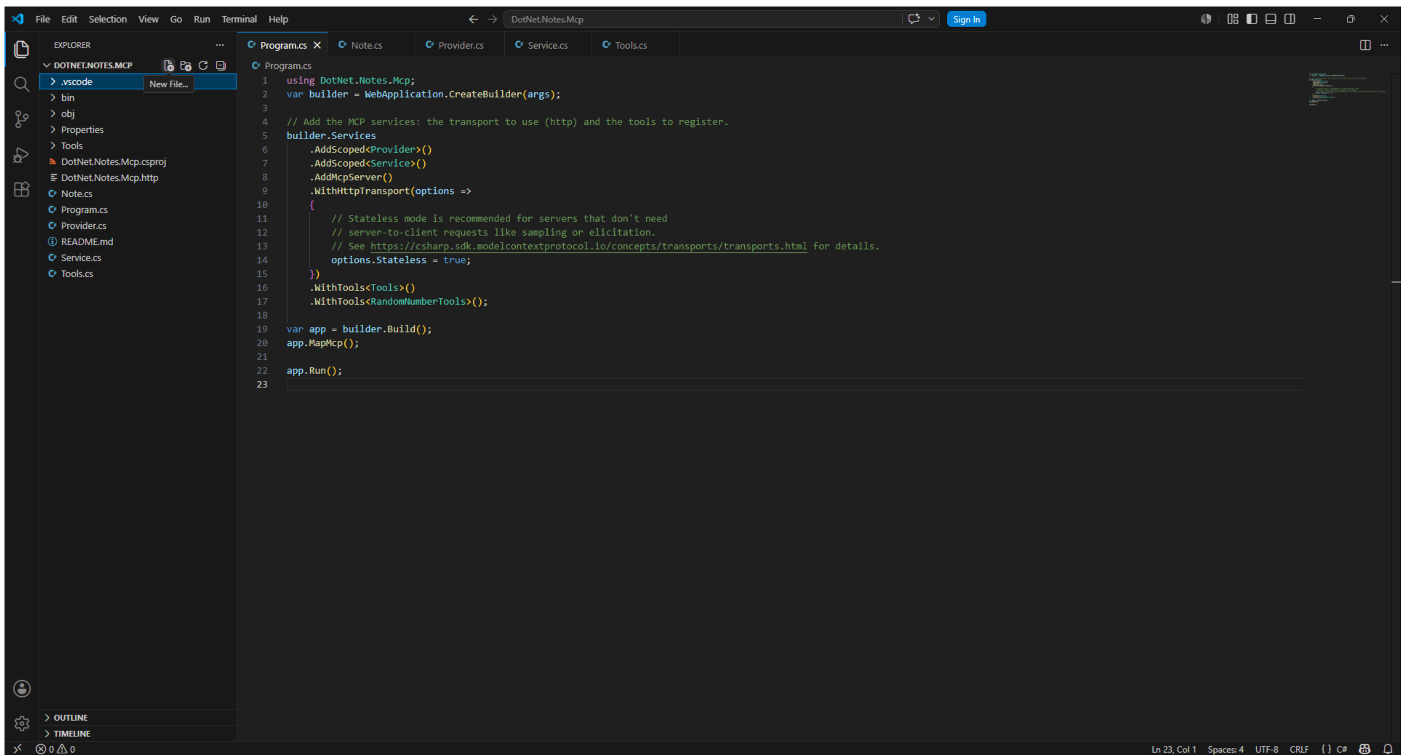


Then *Type* in the following **Name** and press **Enter** after which you should see a **Folder** for **.vscode** in **Explorer** within **Visual Studio Code**.

```
.vscode
```

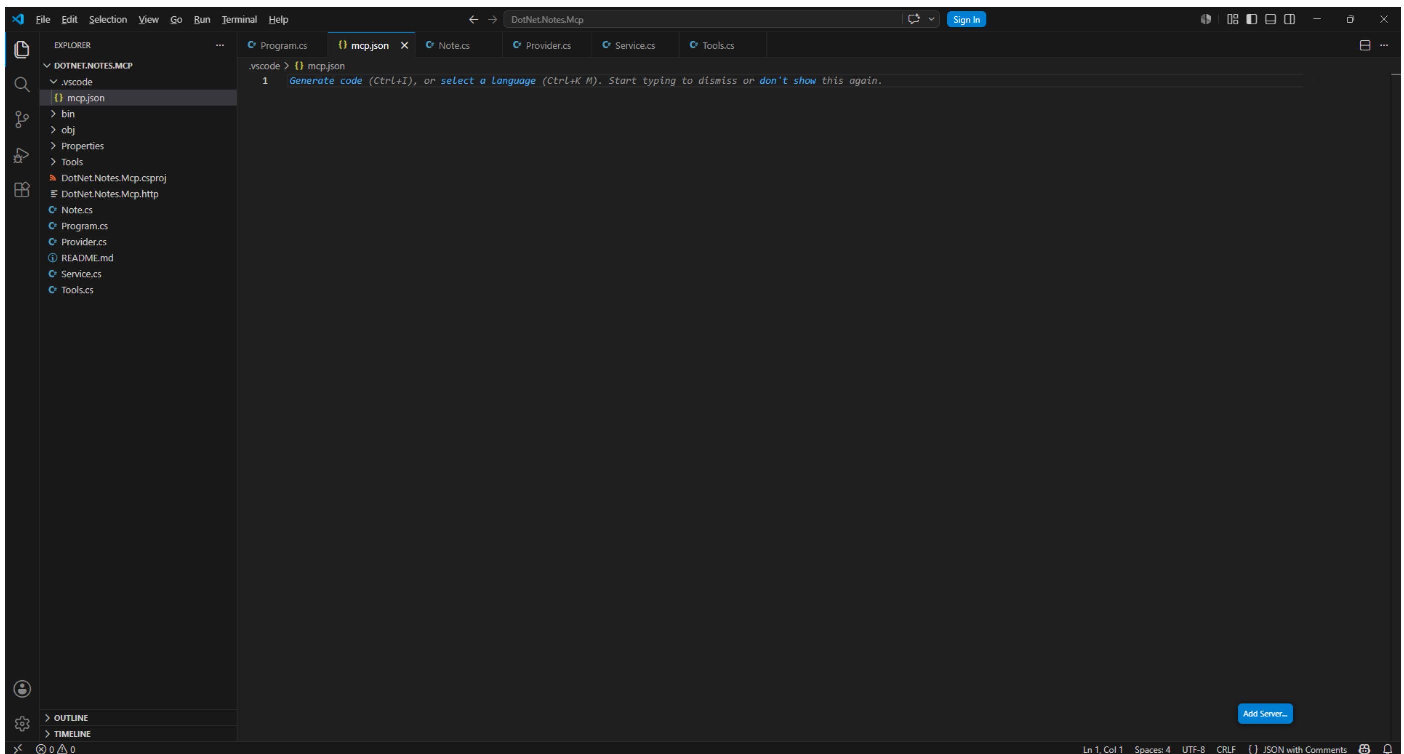
Information – **Folder** of **.vscode** is where things specific to the **Project** can used by **Visual Studio Code**.

In **Visual Studio Code** select **.vscode** in **Explorer** then choose **New File…** next to **DotNet.Notes.Mcp**.



Then *Type* in the following **Name** and press **Enter** after which you should see or select a blank **mcp.json** in **Explorer** within **Visual Studio Code**.

```
mcp.json
```

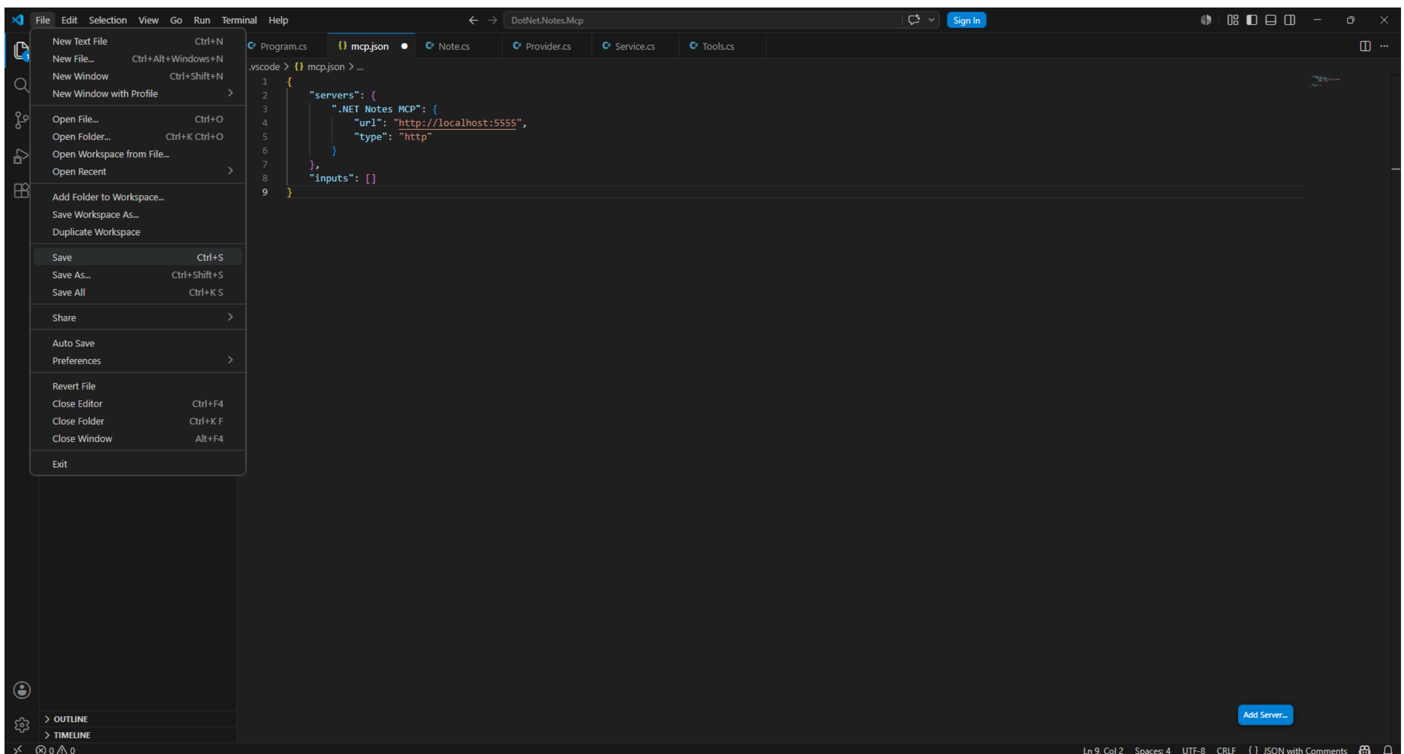


Then within **Visual Studio Code** in **mcp.json** you need to *Copy* and *Paste* the following **JSON**:

```
{
    "servers": {
        ".NET Notes MCP": {
            "url": "http://localhost:5555",
            "type": "http"
        }
    },
    "inputs": []
}
```

Information – This **JSON** represents the **Configuration** needed to use the **Server** using **Model Context Protocol** or **MCP** so that the **Tools** are available to the **Agent** using **AI** within **Visual Studio Code** which is **GitHub Copilot** where the **`url`** should match the one used with **`dotnet watch`** earlier in the **Workshop**.

Next, within **Visual Studio Code** from the **Menu** select **File** and then **Save** as follows:



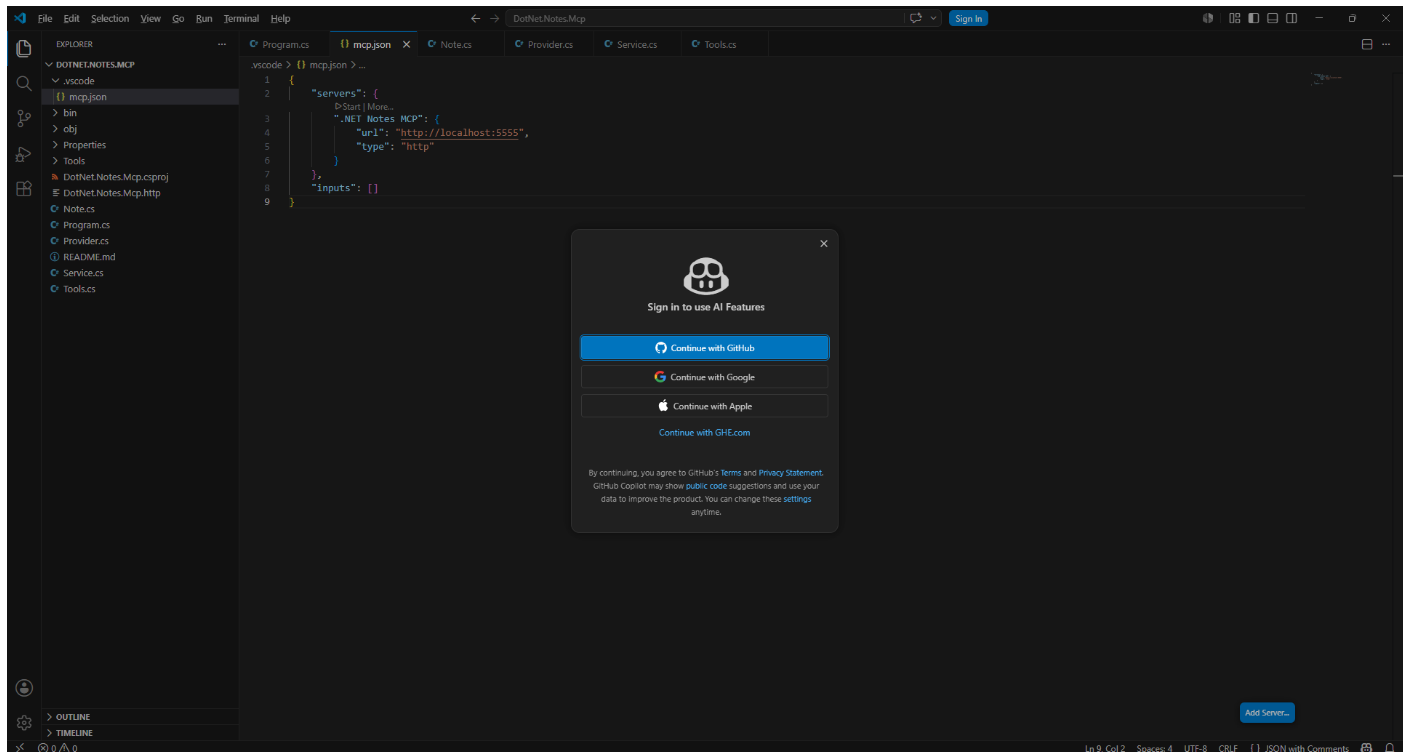
Don't **Close** the **Visual Studio Code** for your **Project** of **DotNet.Notes.Mcp** but if **Visual Studio Code** is **Closed**, then if using a **Mac** you need to go to **Finder**, search for **Visual Studio Code** and then select it to **Open** it again, or if using **Windows** you need to go to **Start**, search for **Visual Studio Code** and select it so it is **Open** again. Then from **Welcome** in **Visual Studio Code** select **DotNet.Notes.Mcp** from **Recent**.

This completes configuring **.NET Notes MCP** to be used by **GitHub Copilot** for the **Workshop**.

Integrate

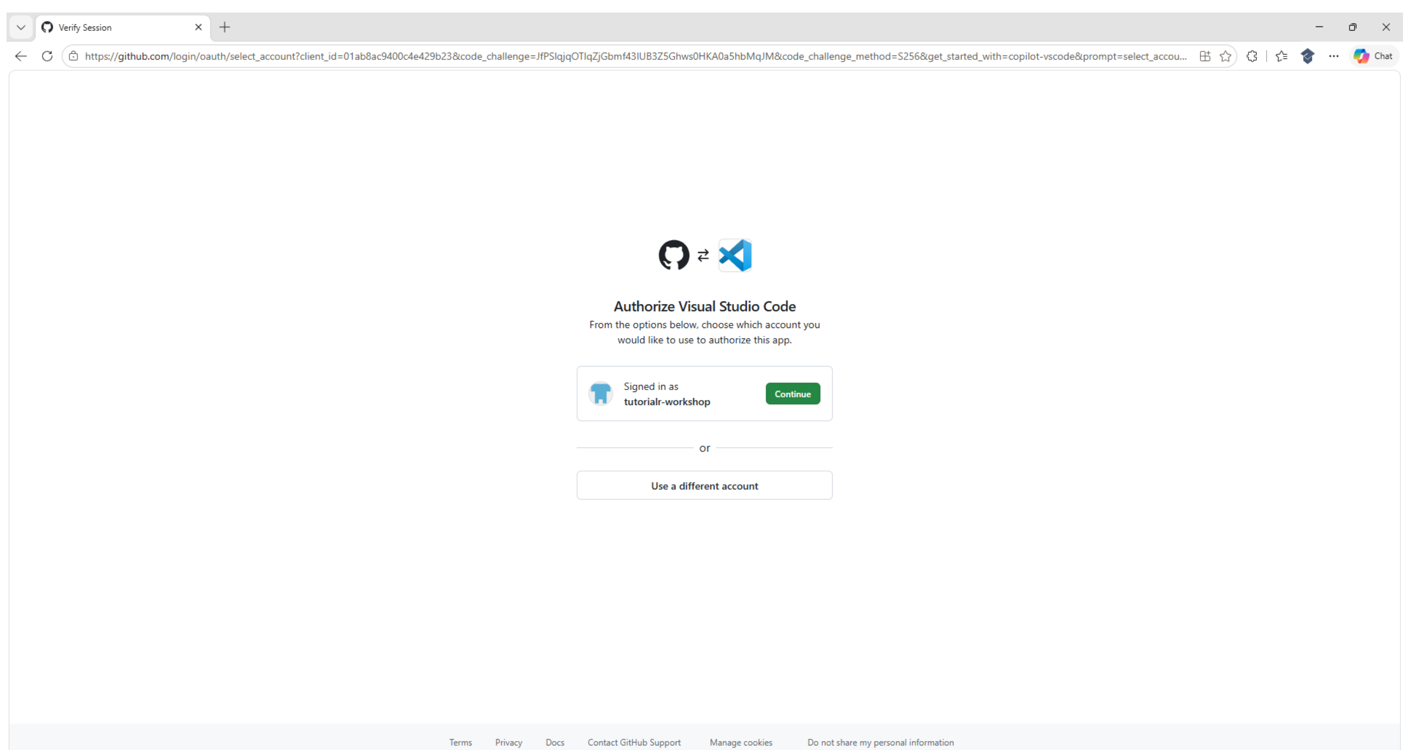
Login GitHub Copilot

If you have not already signed in for **GitHub Copilot** in **Visual Studio Code**, from the **Menu** select **Sign In**:

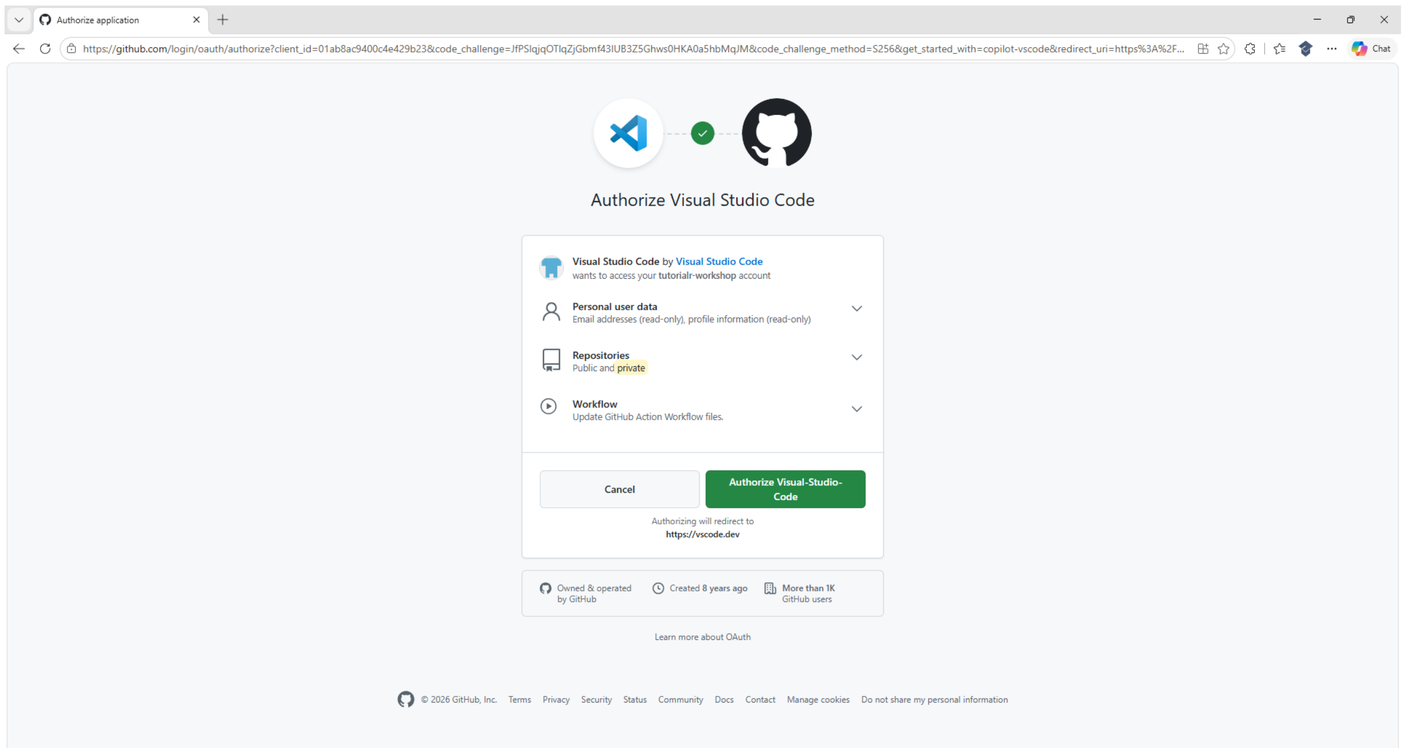


Information – If have already signed in for **GitHub Copilot** then the **Sign In** option will not be shown.

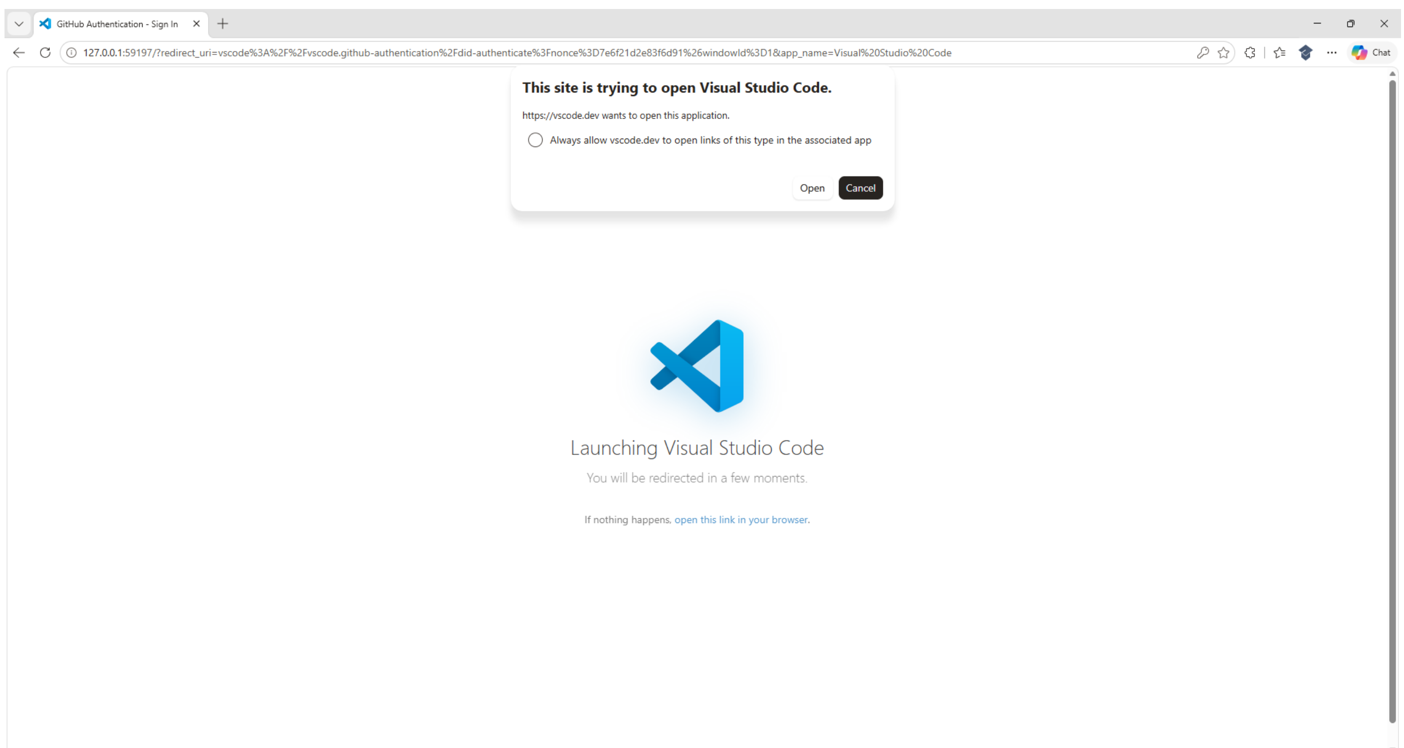
Then in **Visual Studio Code** from **Sign in to use AI features** choose **Continue with GitHub**, which will launch a **Browser** with **Authorize Visual Studio Code** where you can either provide a **Username or Email Address** and **Password** for a **GitHub Account** or already signed with a new or existing **GitHub Account**:



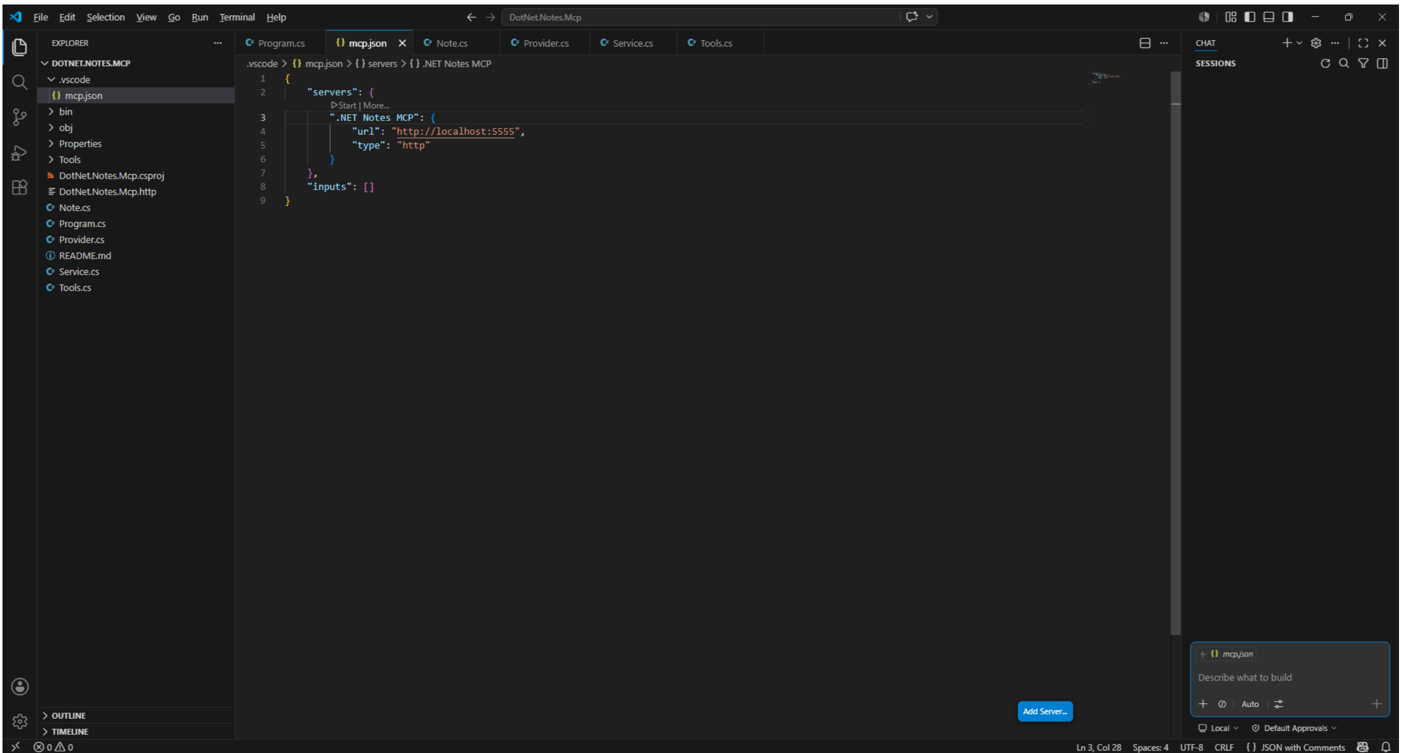
In the **Browser** for **Authorize Visual Studio Code** select **Continue** or **Sign In** if provided a **Username or Email Address** and **Password** and then the following should appear:



While still in the **Browser**, select **Authorize Visual Studio Code**, you may be asked to **Confirm** your **Password**, then be redirected to a **Launching Visual Studio Code** page where you will see a message similar to **This site it trying to open Visual Studio Code,** where you need to select **Open**, as follows:



Once you have selected **Open** in the **Browser** you will be returned to **Visual Studio Code** as follows:

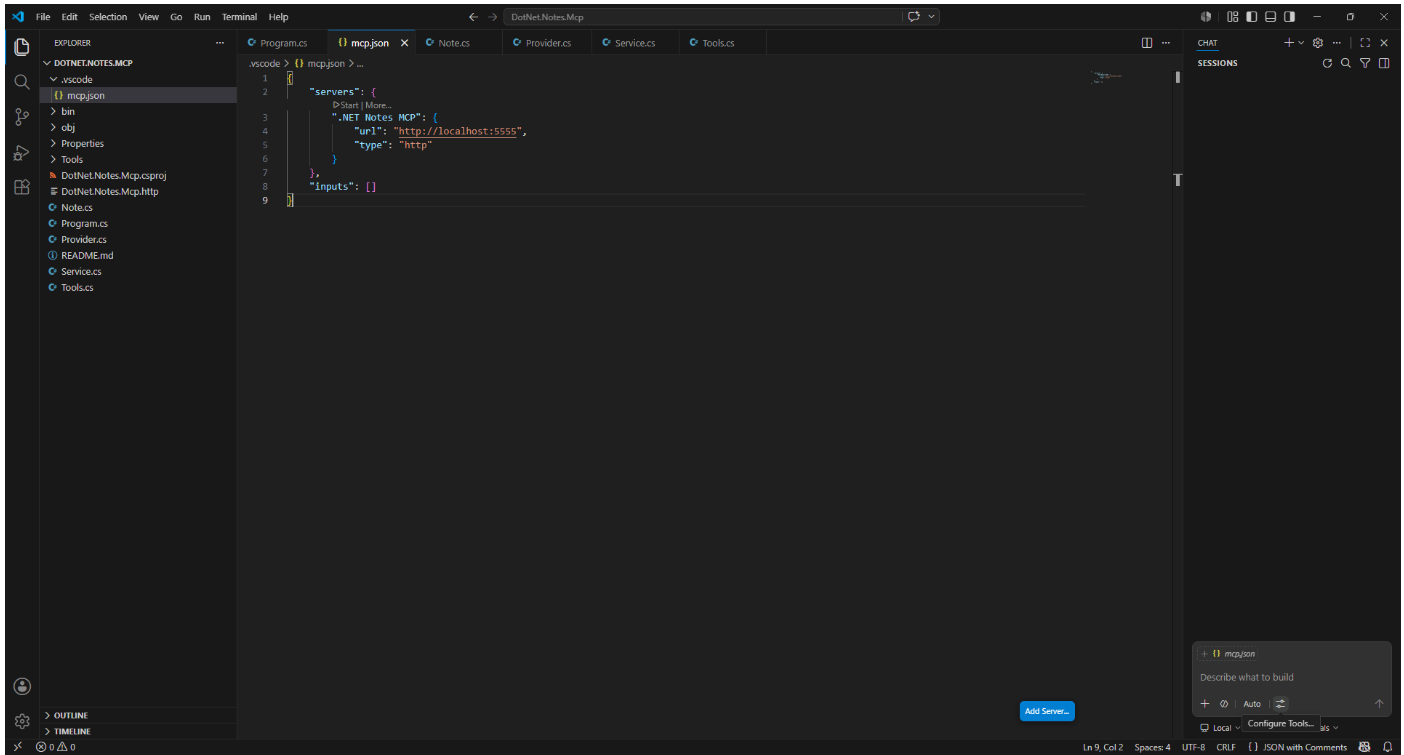


Information – **Visual Studio Code** will display the **Pane** for **Chat** when signed in for **GitHub Copilot** and the **Sign In** option will no longer appear on the **Menu**.

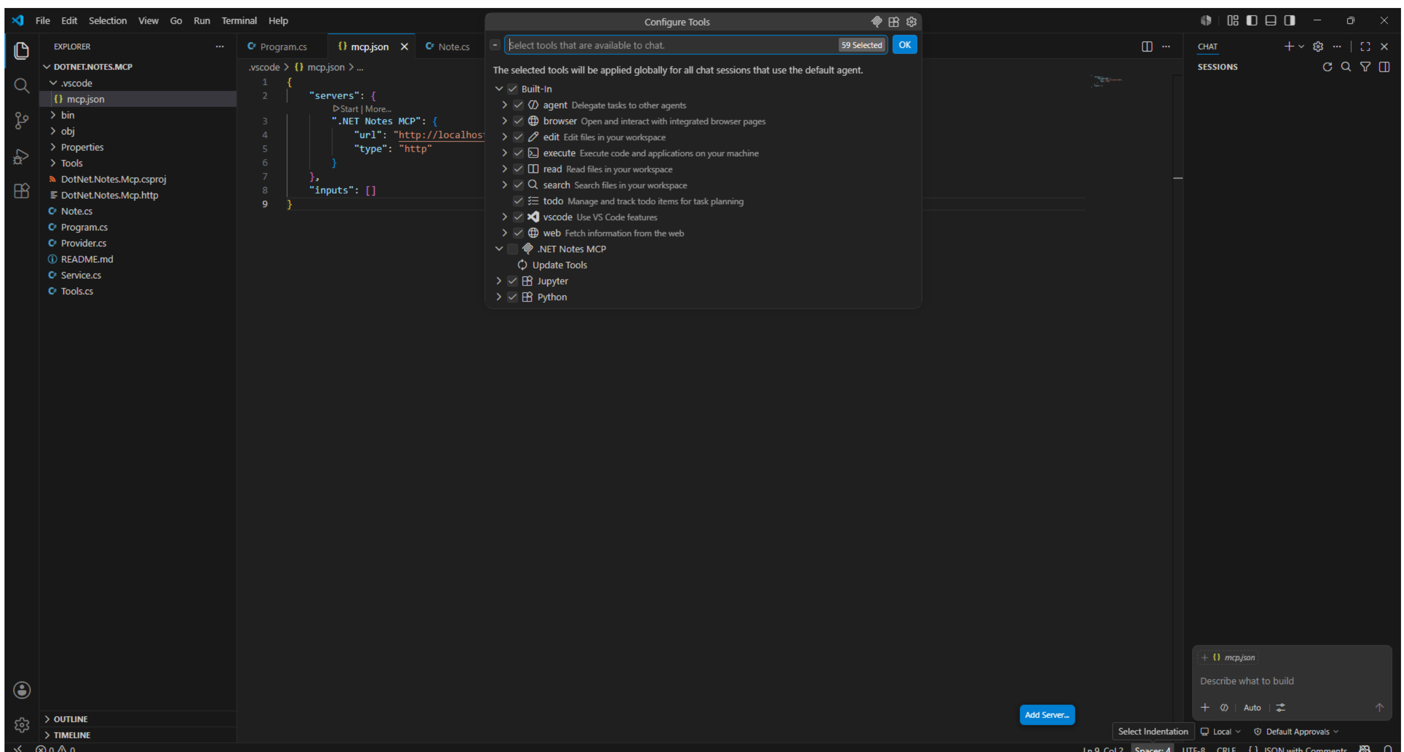
That completes the process of signing in to **GitHub Copilot** in **Visual Studio Code** for the **Workshop**.

Configure GitHub Copilot

While in **Visual Studio Code** from the **Pane** for **Chat** select **Configure Tools…** below the **Chatbox**, which displays **Describe what to build**, as follows:

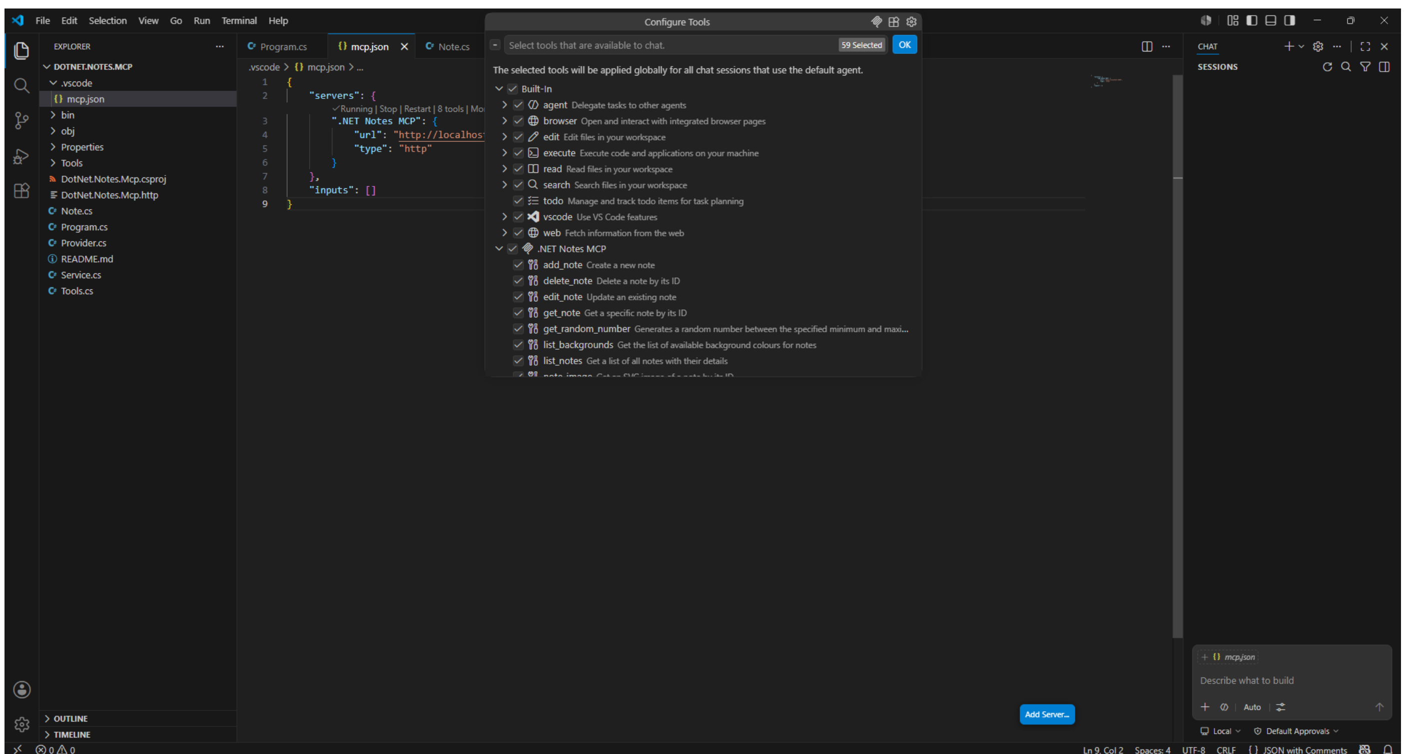


Once selected the **Configure Tools** will be displayed in **Visual Studio Code** as follows:



CC BY SA

Then in **Configure Tools** in **Visual Studio Code** within the for **.NET Notes MCP** select **Update Tools**, which should then display a list of the **Tools** as follows:



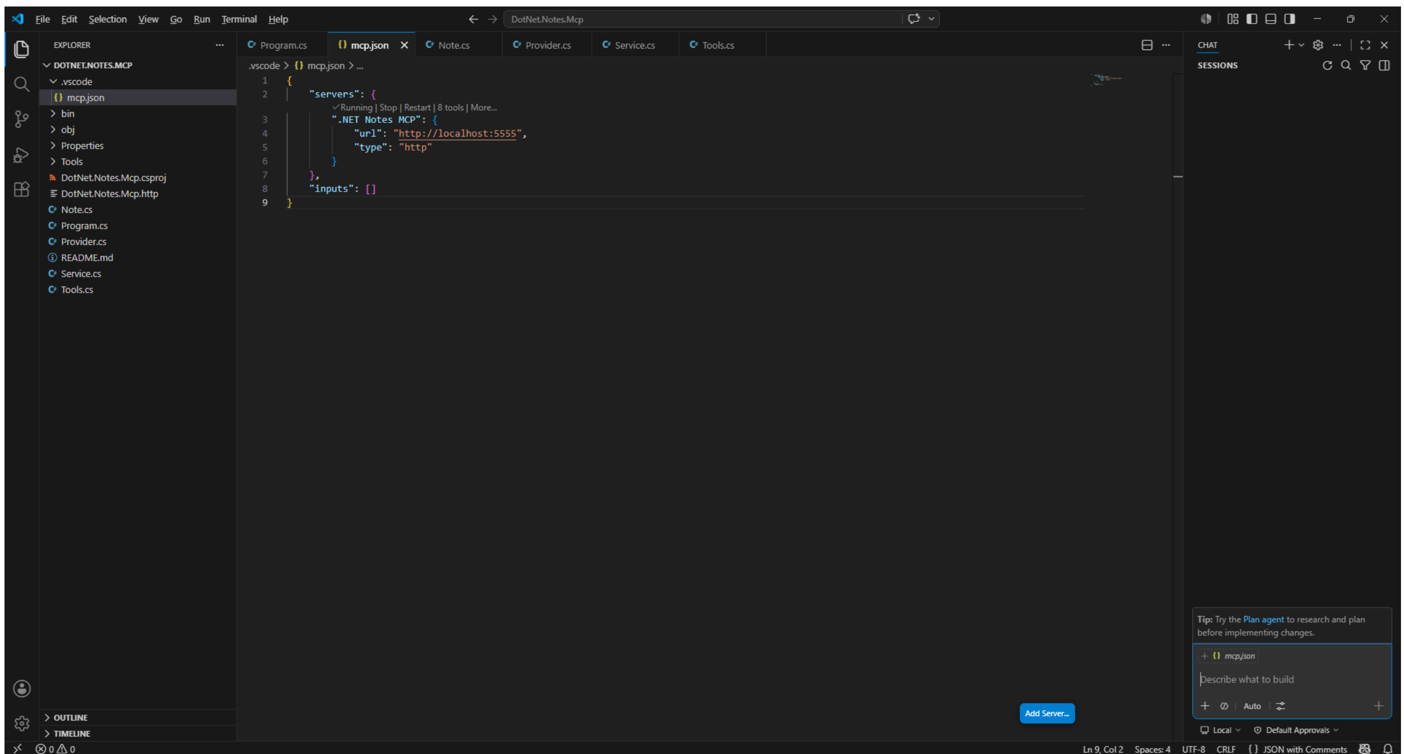
Information – If within **Configure Tools** for **Visual Studio Code** you do not see **.NET Notes MCP**, then check you have not missed any previous **Steps** of the **Workshop** or if you have **Closed** the **Command Prompt** on **Windows** or **Terminal** on **Mac**, where if you close the **Terminal** on **Mac** then you need to go to **Finder**, search for **Terminal** and then select it to **Open** it, or if you **Closed** the **Command Prompt** on **Windows**, go to **Start**, search for **Command Prompt** and then select it to **Open** it. Then once opened you need to change directory using **cd** to the location for your **Project**, for example *cd DotNet.Notes.Mcp* and then you need to type **`dotnet watch --urls http://localhost:5555`** followed by **Enter**. Then return to **Visual Studio Code** and then from the **Pane** for **Chat**, select **Configure Tools…** and within **Configure Tools** you should see **.NET Notes MCP** and select **Update Tools** to access to the **Tools** in **GitHub Copilot**.

Don't **Close** the **Visual Studio Code** for your **Project** of **DotNet.Notes.Mcp** but if **Visual Studio Code** is **Closed**, then if using a **Mac** you need to go to **Finder**, search for **Visual Studio Code** and then select it to **Open** it again, or if using **Windows** you need to go to **Start**, search for **Visual Studio Code** and select it so it is **Open** again. Then from **Welcome** in **Visual Studio Code** select **DotNet.Notes.Mcp** from **Recent**.

That completes the process of configuring **GitHub Copilot** in **Visual Studio Code** for the **Workshop**.

Demonstrate GitHub Copilot

Within **Visual Studio Code** the **Pane** for **Chat** should be displayed, if it is not then from the **Menu** select **View** then **Chat**, so it is displayed as follows:

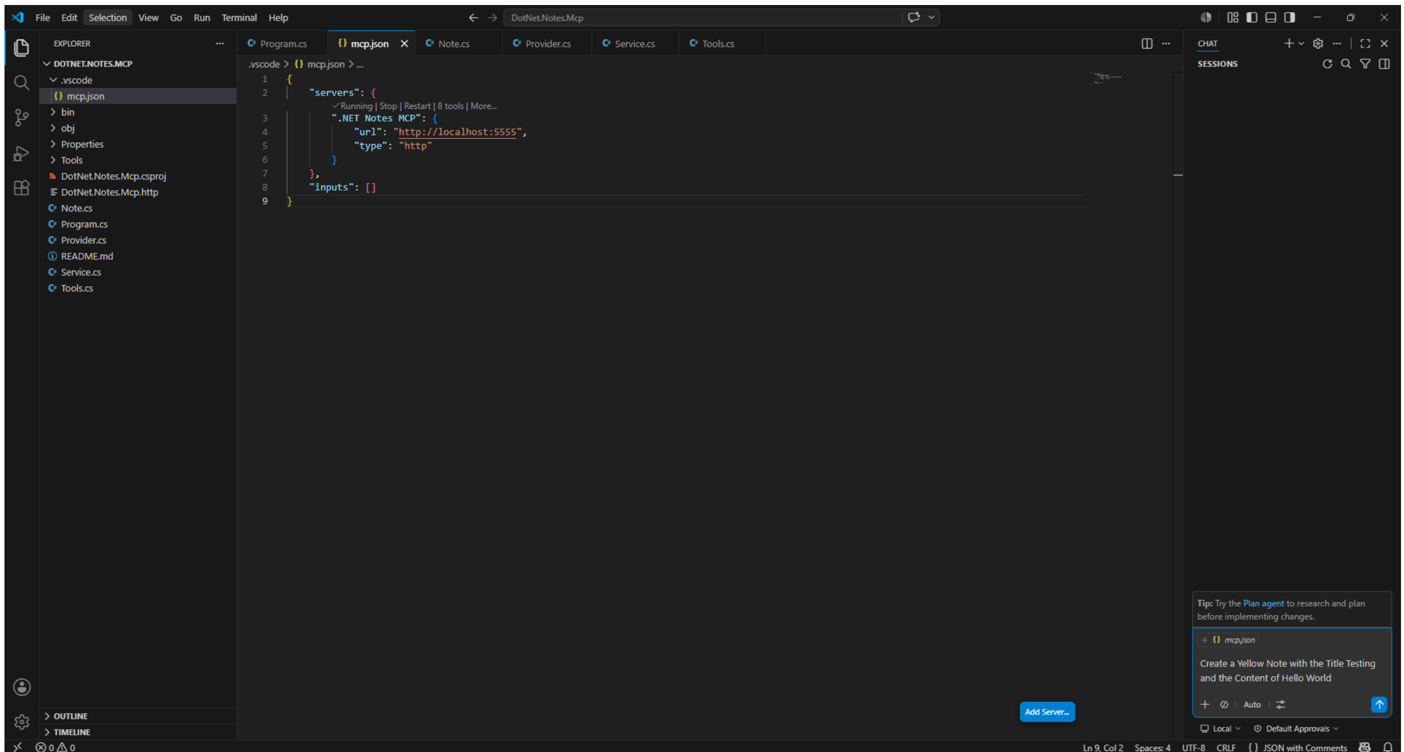


Information – **Visual Studio Code** can support many modes of interacting with **GitHub Copilot** including **Chat**, which makes it possible to demonstrate the **Tools** supported by **.NET Notes MCP** of **`add_note`**, **`delete_note`**, **`edit_note`**, **`get_note`**, **`get_random_number`**, **`list_backgrounds`**, **`list_notes`** and **`note_image`**. The **Tool** of **`get_random_number`** was included by default in the **Project.**

Within **Visual Studio Code** the **Pane** for **Chat** you can *Copy* and *Paste* into the **Chatbox** this **Prompt**:

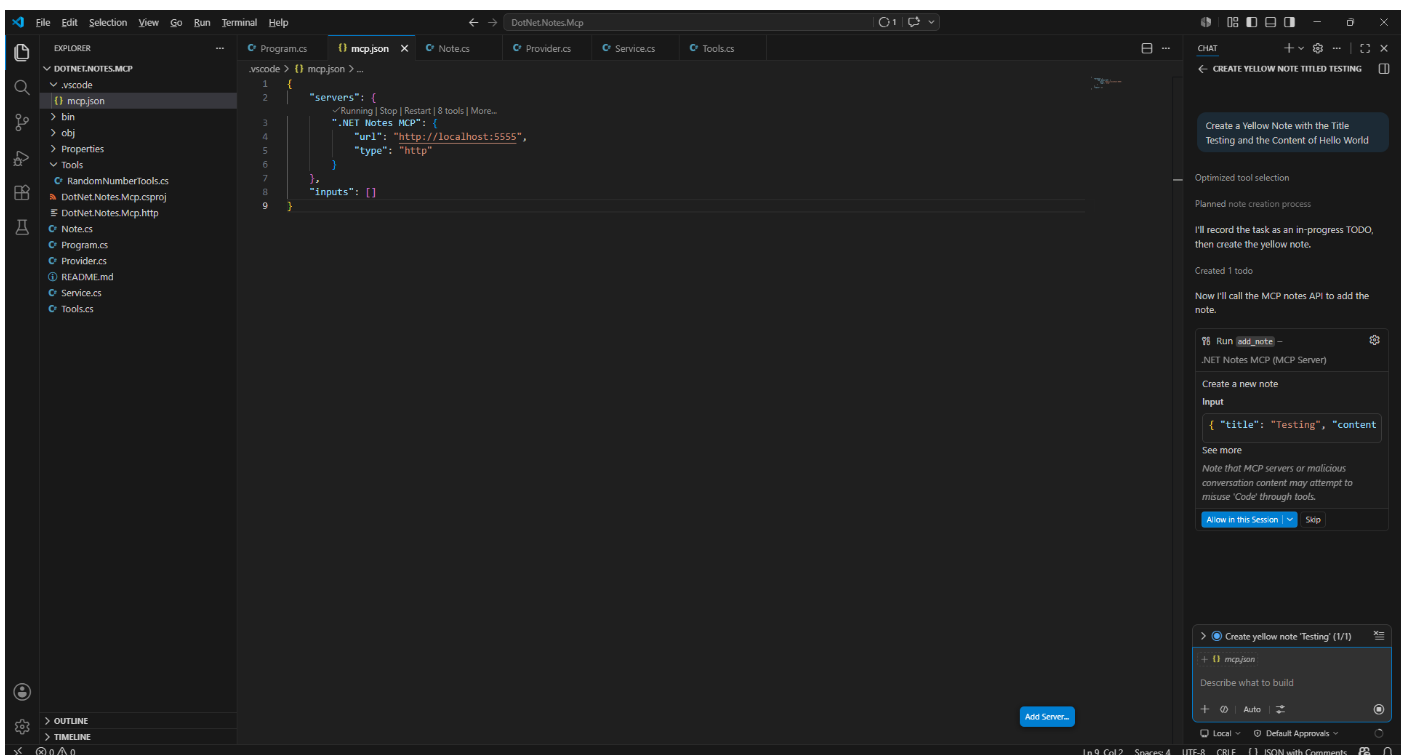
```
Create a Yellow Note with the Title Testing and the Content of Hello World
```

The **Chatbox** in the **Pane** for **Chat** within **Visual Studio Code** should contain the **Prompt** as follows:

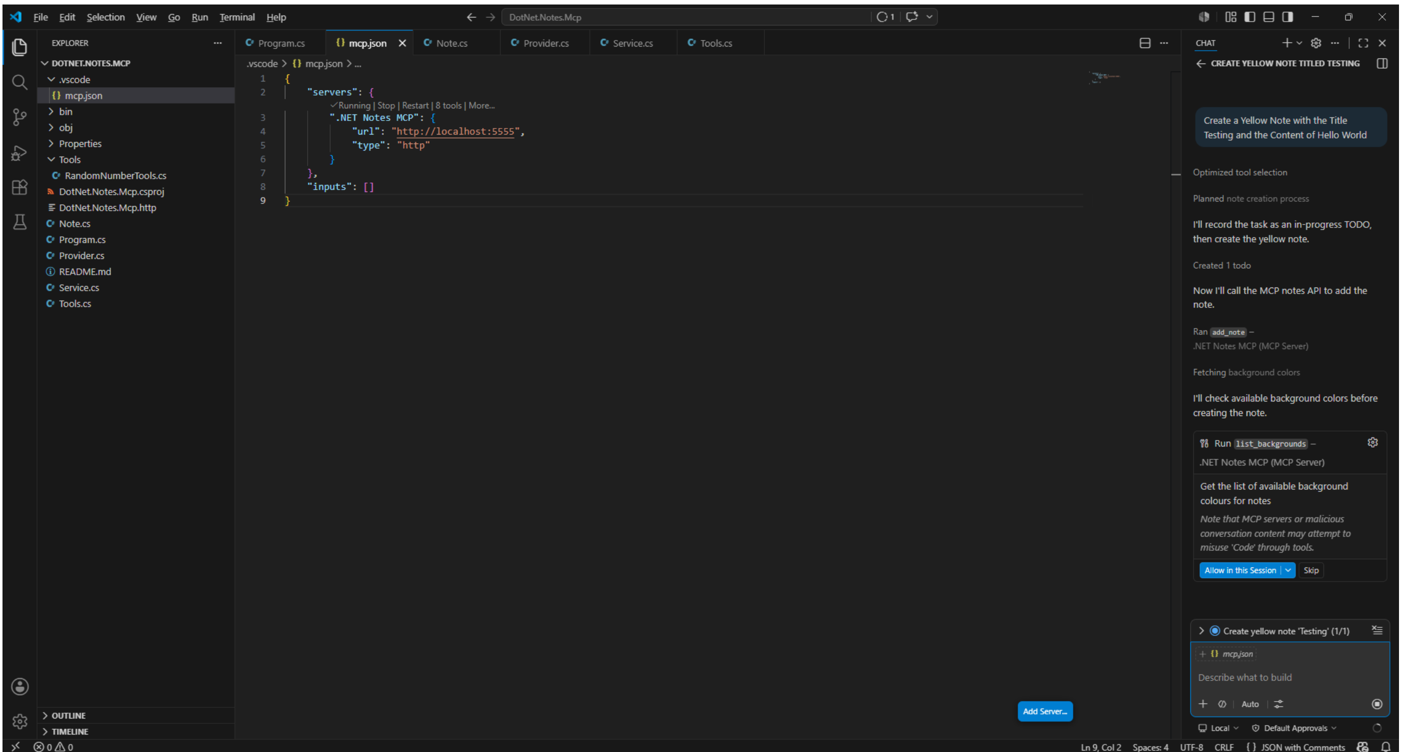


Information – **Background** could be any of those supported as well as any **Title** or **Content** for the **Note**.

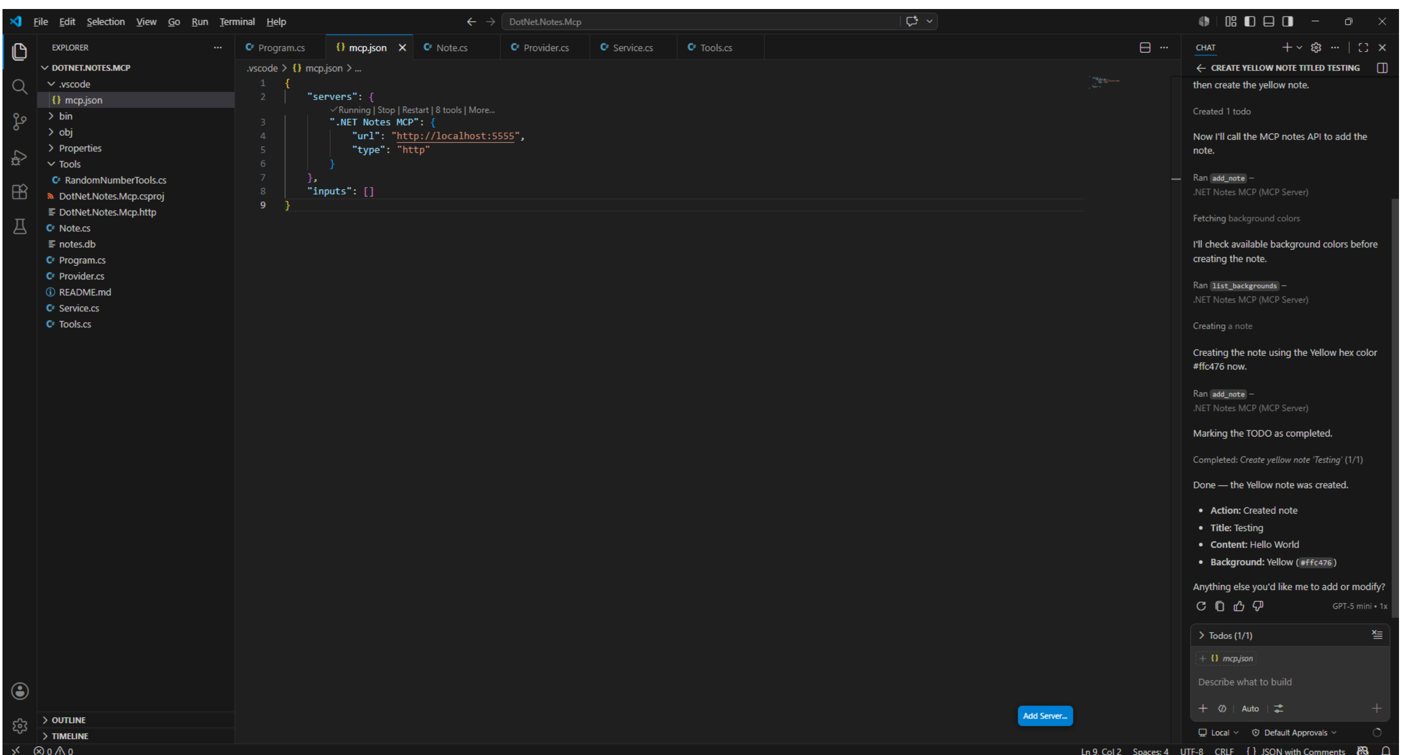
Then select **Send** from the **Chatbox** in the **Pane** for **Chat** within **Visual Studio Code** to submit the **Prompt**:



What happens next will vary depending on **GitHub Copilot**, but it should attempt to **Run** the **Tool** of **`add_note`**, by asking to **Allow in this Session**, which you can then select to proceed, where it may ask to **Run** the **Tool** of **`list_backgrounds`** to determine the available colours which you can also select the option to **Allow in this Session** to proceed as follows:



GitHub Copilot should then have created a new **Note**, the output of doing so may vary, as follows:



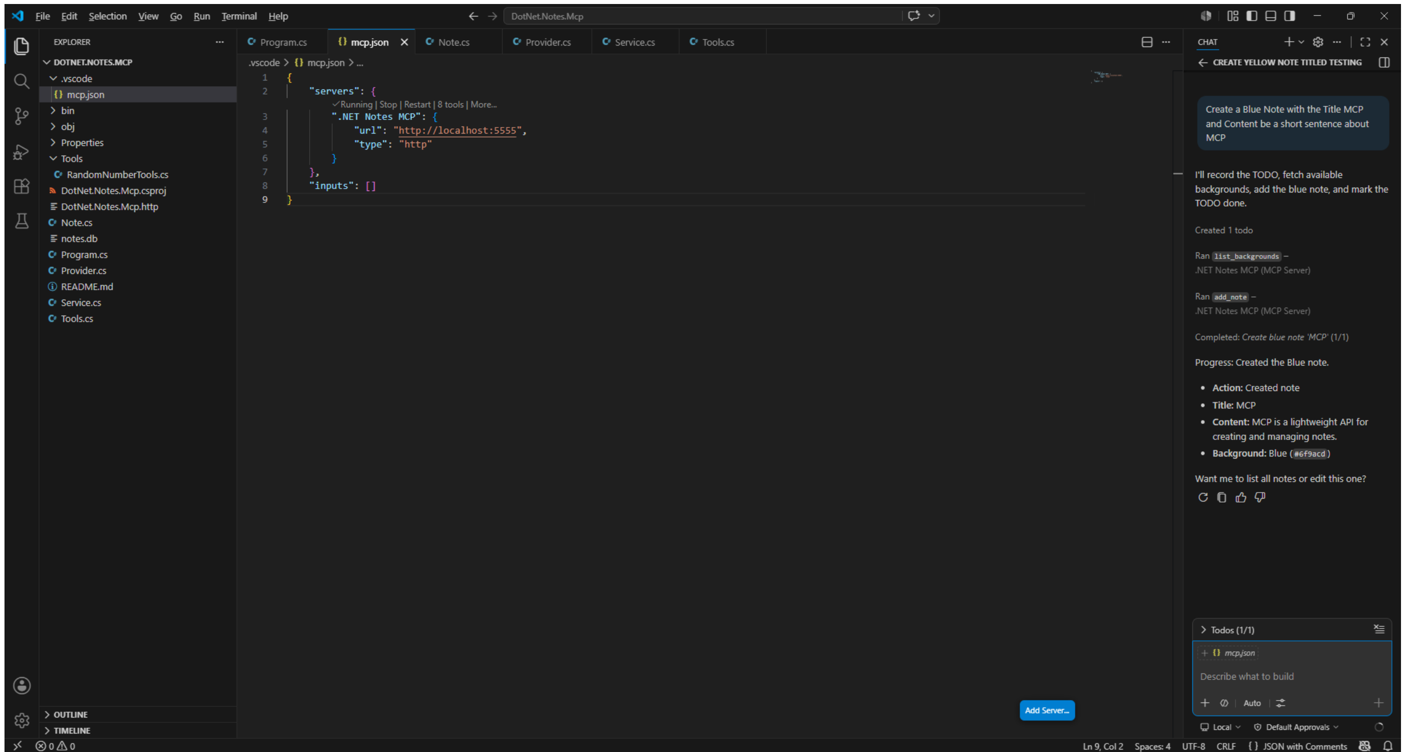
Information – The output and exact behaviour of **GitHub Copilot** is non-deterministic, which is the nature of **AI** using **Large Language Models** or **LLMs**. However, the outcome of the created **Note** will be deterministic as it will always create a **Note** with the **Background**, **Title** and **Content** required.

Within **Visual Studio Code** the **Pane** for **Chat** you can *Copy* and *Paste* into the **Chatbox** this **Prompt**:

```
Create a Blue Note with the Title MCP and Content be a short sentence about MCP
```

Information – **Background** could be any other colour with a **Title** and **Content** to be generated.

Select **Send** then **GitHub Copilot** should **Run** the **Tools** of `add_note` and `list_backgrounds`:

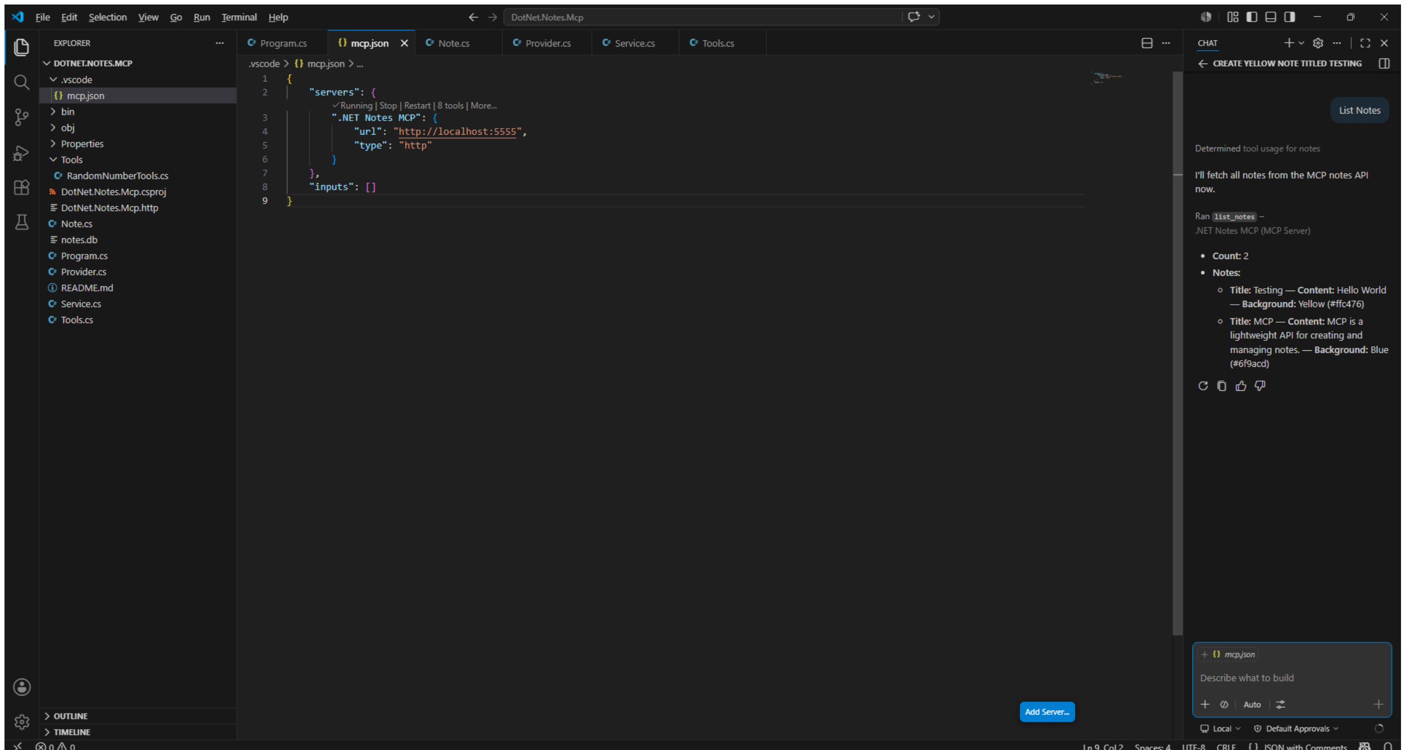


Information – The **Note** created by **GitHub Copilot** should contain the **Title** as specified but the **Content** won't literally be what was in the **Prompt** but be a short, and varying, sentence about **MCP**.

Within **Visual Studio Code** the **Pane** for **Chat** you can *Copy* and *Paste* into the **Chatbox** this **Prompt**:

```
List Notes
```

Select **Send** then **GitHub Copilot** should ask **Run** the **Tool** of **`list_notes`**, select the option to **Allow in this Session** to output the **Notes**, which may vary in style, but should be as expected as follows:



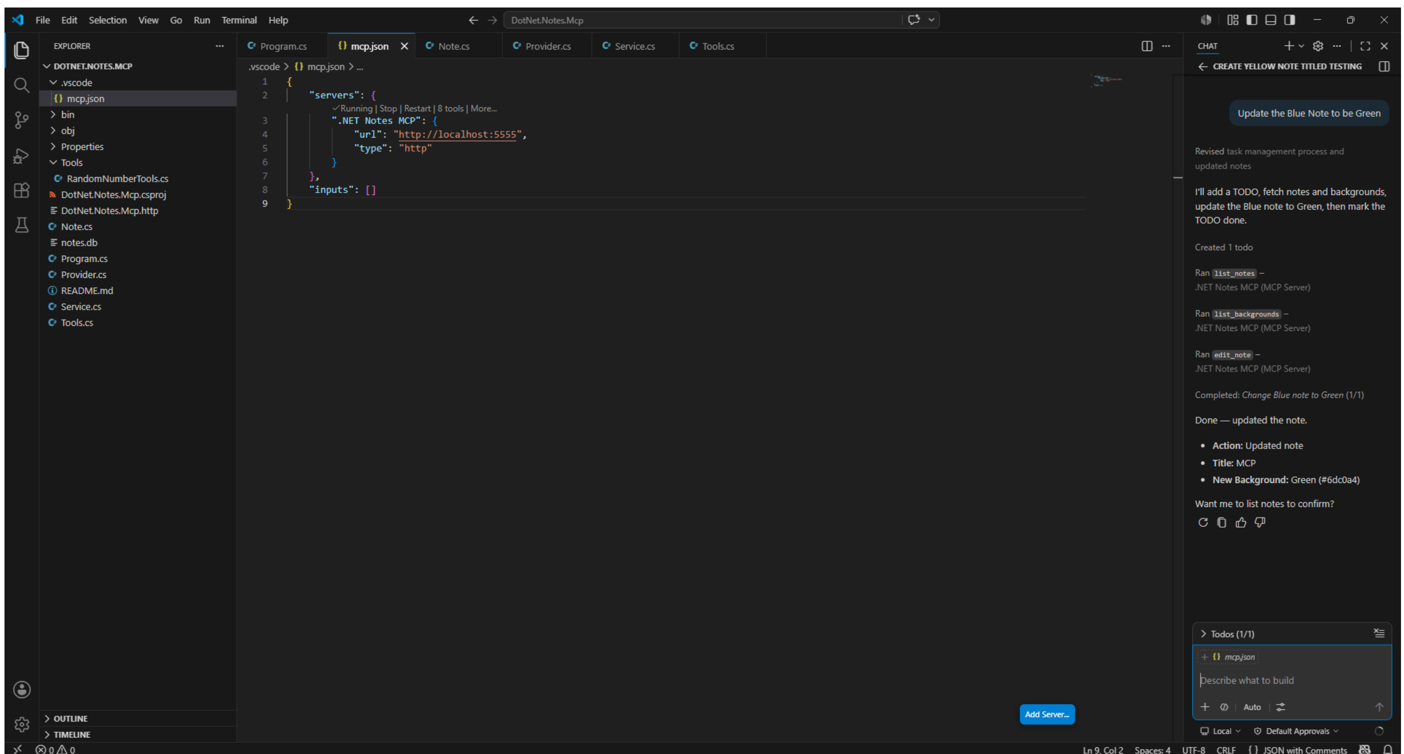
Information – **GitHub Copilot** will choose whatever way in the moment makes sense to output **Notes** as there is nothing in the **Tools** to control or describe how this should be done as the output style is not important, but if **`Description`** could be amended to specify how this should be displayed.

Within **Visual Studio Code** the **Pane** for **Chat** you can *Copy* and *Paste* into the **Chatbox** this **Prompt**:

```
Update the Blue Note to be Green
```

Information – **Colour** could be any other colour other than **Yellow** so both **Notes** have different colours.

Select **Send** then **GitHub Copilot** should ask **Run** the **Tool** of `edit_notes`, select the option to **Allow in this Session** to edit the **Note**, the output of which may vary in style, as follows:



Information – **GitHub Copilot** will choose whatever way in the moment makes sense for it to output the updated **Note** as there is nothing in the **Tool** to control the style, as it just returns the values of the **Note**.

You can try out a variety of different **Prompts** in **GitHub Copilot** to use the **Tools** for **.NET Notes MCP** just remember to select **Allow in this Session** when needed remaining **Tools** such as **`get_note`**, **`note_image`**, **`delete_note`** or **`get_random_number`** here are some **Prompts** you can *Copy* and *Paste* into the **Chatbox**:

```
Delete the Green Note
```

```
Get Images for all Notes
```

```
Create Seven Notes with different Colours, Titles and Content
```

```
Using the Random Number Generator Get a Random Note
```

Information – **GitHub Copilot** will run the appropriate **Tools** depending on what is being asked for in each **Prompt**, if you get any **Ids** back you can try to ask for a **Note** by **Id** to trigger the **Tool** of **`get_note`**, if not already triggered by **GitHub Copilot**, but there's plenty of combinations to try to see what **Tools** are used.

You can also try creating **Notes** where the **Background**, **Title** and **Content** are not acceptable by the **Validation** such as the following **Prompts** you can *Copy* and *Paste* into the **Chatbox**:

```
Create a Brown Note
```

```
Create a Note with Title over 50 characters and Content over 255 characters
```

Information – **GitHub Copilot** should reject these **Prompts**, although it is possible it will either ask or perform the correct action, but it likely that **GitHub Copilot** will not perform these actions.

This concludes the **Workshop** so you can close **Visual Studio Code**, any **Browser** tabs or windows and **Terminal** on **Mac** or **Command Prompt** on **Windows**.