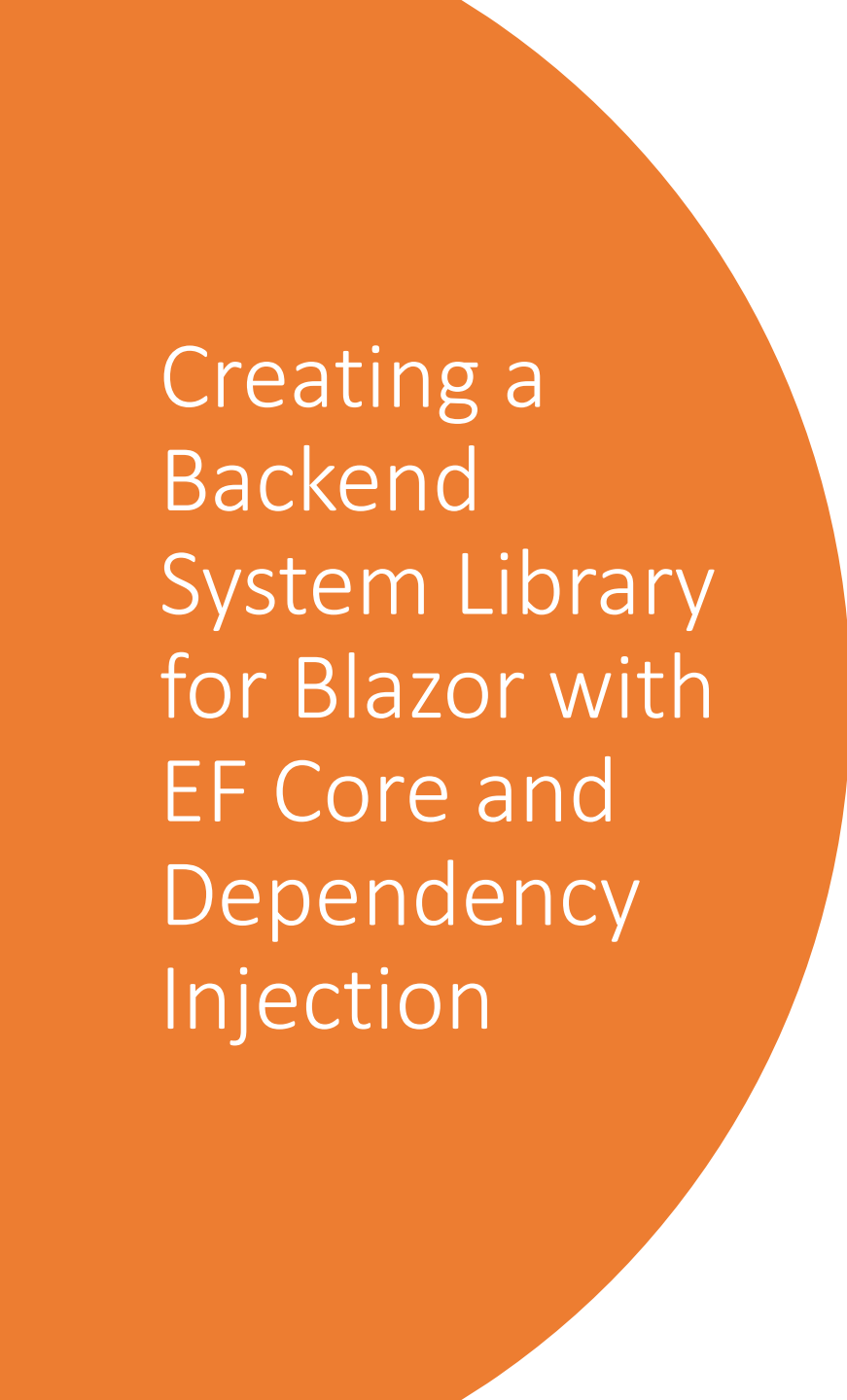


# Session 4 - Simple Table Query with Entity Framework


Name \_\_\_\_\_

Signature \_\_\_\_\_

Date \_\_\_\_\_

A large orange circle on the left side of the slide, partially cut off by the edge.

# Creating a Backend System Library for Blazor with EF Core and Dependency Injection

- In the world of web development, a robust backend is the foundation of every successful application.
  - Today, we're diving into a crucial aspect of web development – creating a backend system library for Blazor applications.
- 
- A yellow dashed line in the bottom right corner, consisting of several short, curved segments.

# Creating the System Library

---

## 1. Open the "HogWild" Solution:

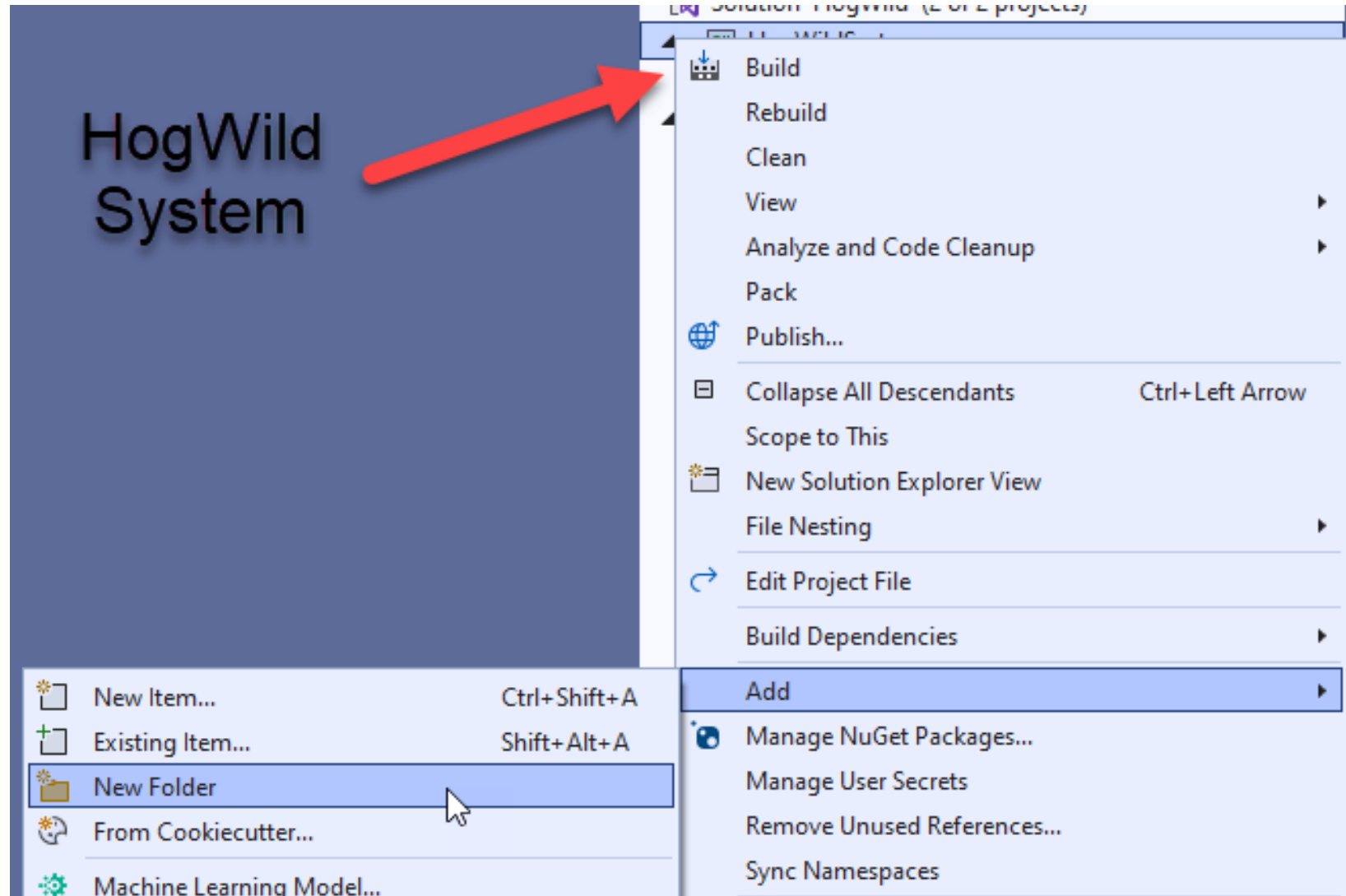
1. In Visual Studio, click on "HogWild" solution file that contains your Blazor application.

## 2. Solution Explorer:

1. In the Solution Explorer window, right-click on the solution (HogWild) where you want to add the Class Library.

## 3. Add > New Project:

1. From the context menu, select "Add" and then "New Project."



# Creating the System Library

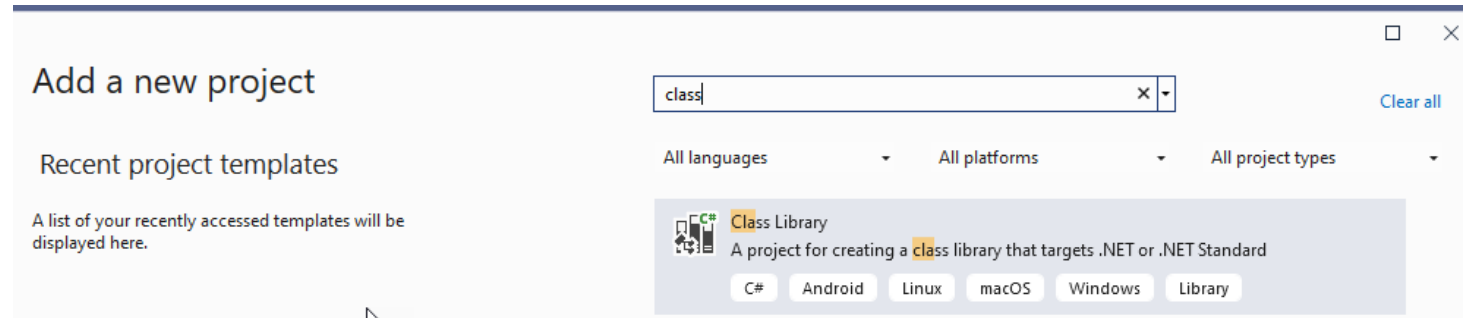
---

## 1. Create New Project Dialog:

1. In the "Create a new project" dialog, you'll see a list of project templates.
2. Use the search box in the upper right corner and type "Class Library."

## 2. Select Class Library Template:

1. Select the "Class Library (.NET Standard)" template from the list. This is suitable for creating a class library.



# Creating the System Library

## 1. Configure the New Project:

1. Provide a name for the project. In this case, name it "HogWildSystem."
2. Choose the location where you want to save the project files (Use the default location).
3. Ensure the target framework is .NET 9.0

## 2. Create:

1. Click the "Create" button to create the "HogWildSystem" Class Library project.

## Configure your new project

Class Library

C#

Android

Linux

macOS

Windows

Library

Project name

HogWildSystem

Location

E:\OneDrive - NAIT\DMIT 2018 - Course Files\Blazor Demo\HogWild

Project will be created in "E:\OneDrive - NAIT\DMIT 2018 - Course Files\Blazor Demo\HogWild\HogWildSystem\"

## Additional information

Class Library

C#

Android

Linux

macOS

Windows

Library

Framework ⓘ

.NET 7.0 (Standard Term Support)

# Creating the System Library (Continue)

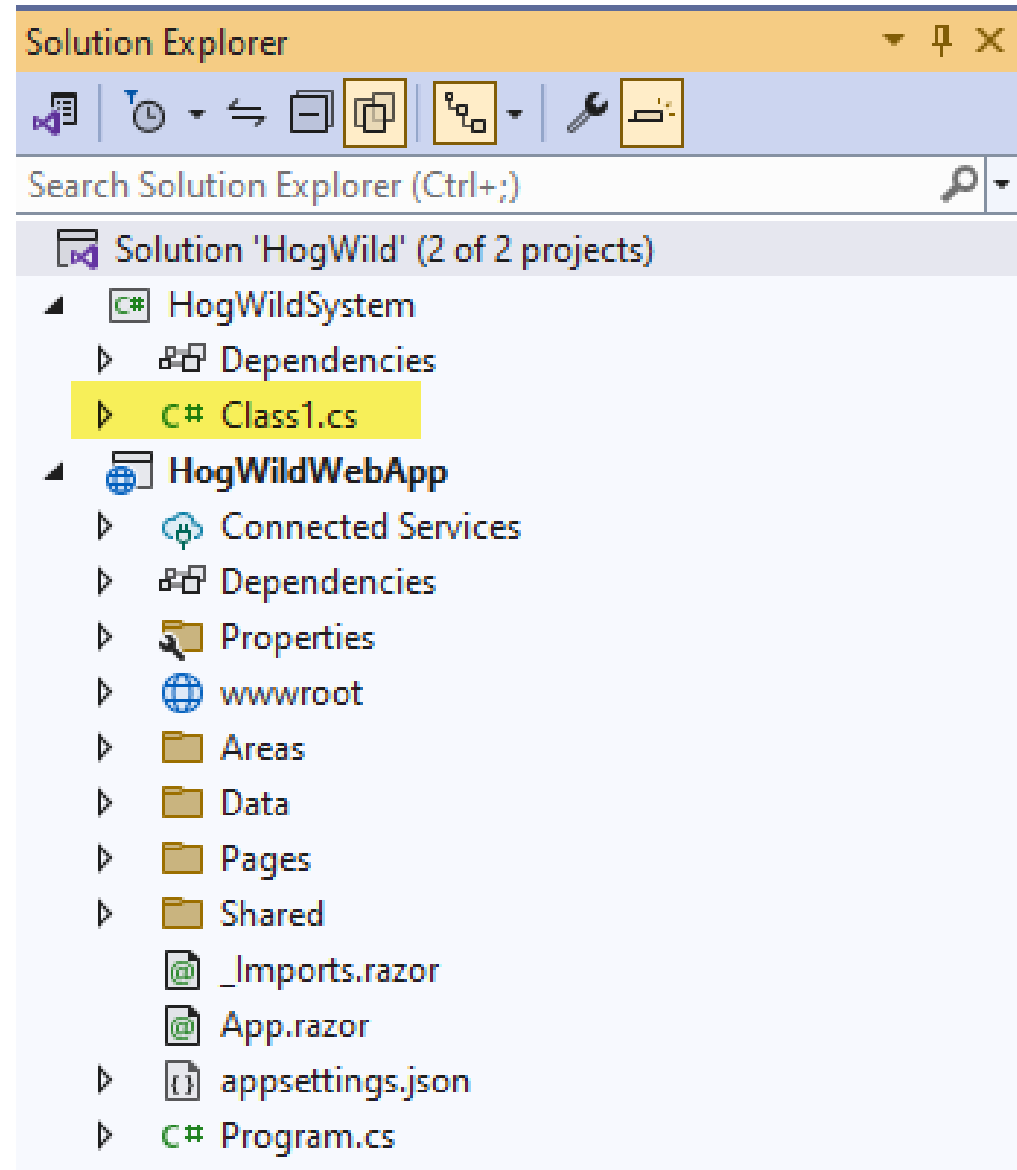
---

## 1. HogWildSystem Project:

1. Visual Studio will create and add the "HogWildSystem" project to your solution.

## 2. Class Library Structure:

1. Inside the "HogWildSystem" project, you'll see the default class file (Class1.cs). You can delete this file.



# Creating the System Library (Continue)

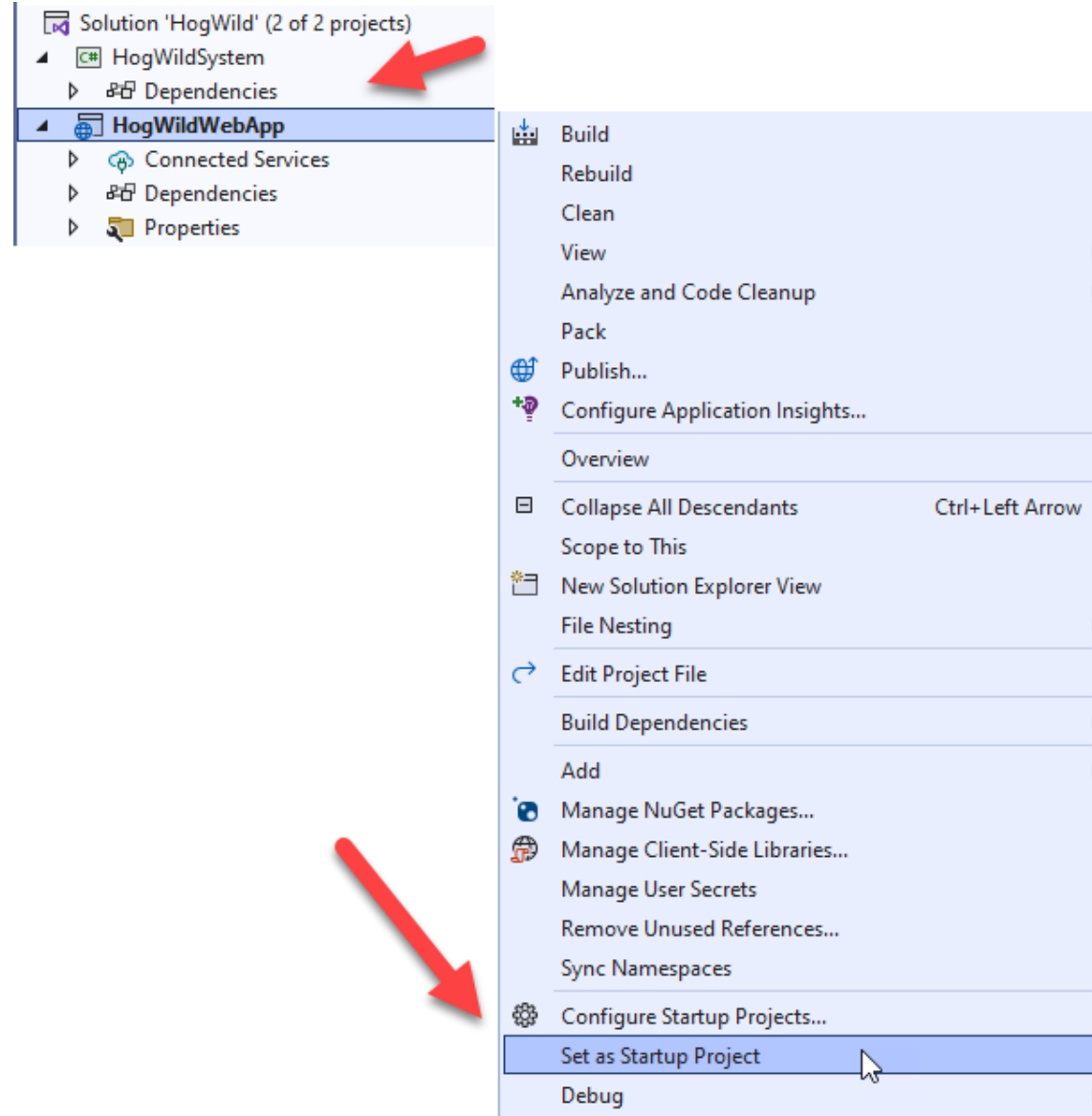
---

## 1. Set as Startup Project:

1. Right-click on the "HogWildWebApp" project in the Solution Explorer.

## 2. Set as Startup Project:

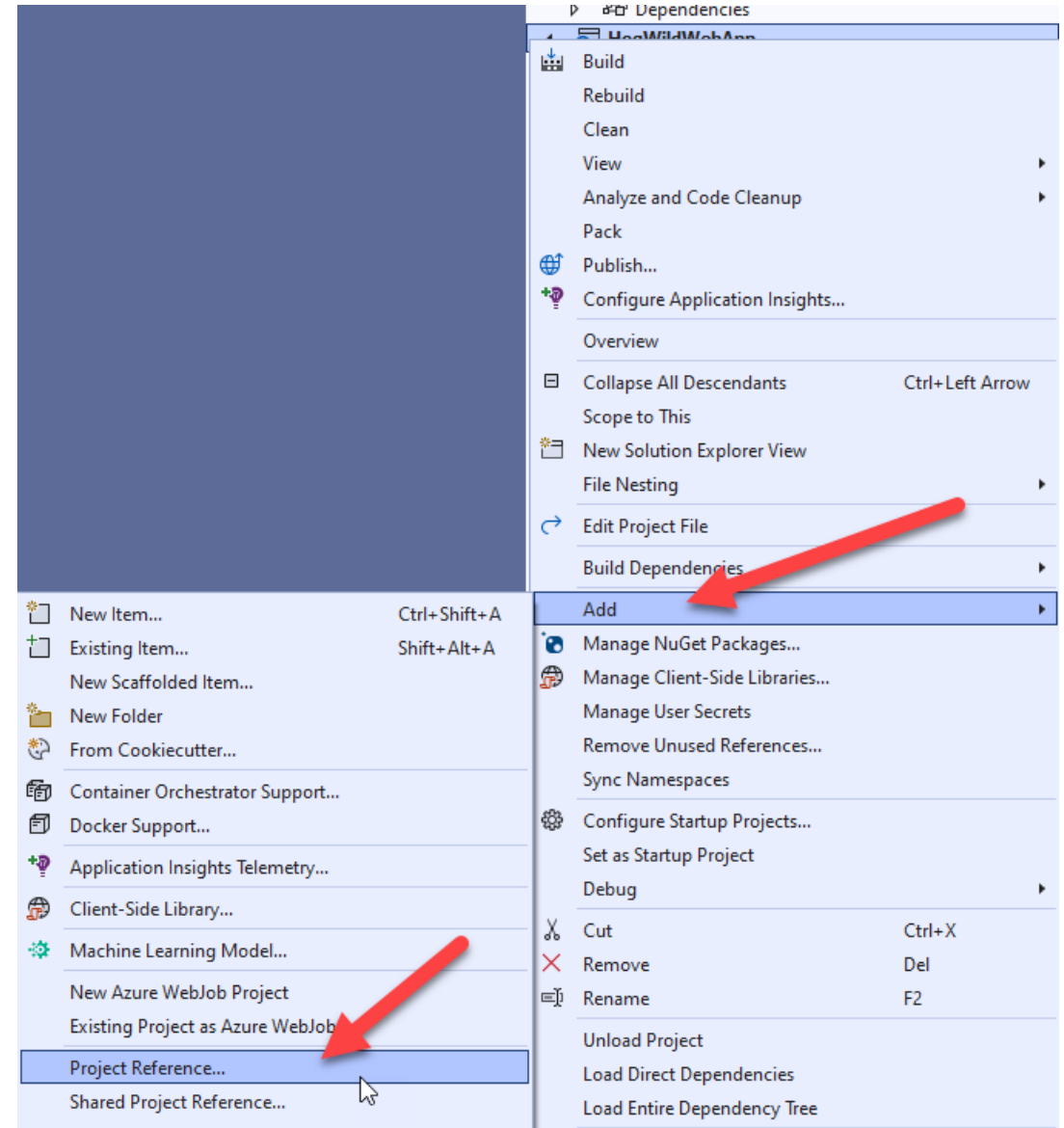
1. From the context menu, select "Set as Startup Project." This action designates "HogWildWebApp" as the startup project for your solution.



# Creating the System Library (Continue)

## 1. Usage in Blazor Application:

1. To use the "HogWildSystem" Class Library in your Blazor application, right-click on the "**HogWildWebApp**" project within the solution and select "Add Project Reference."

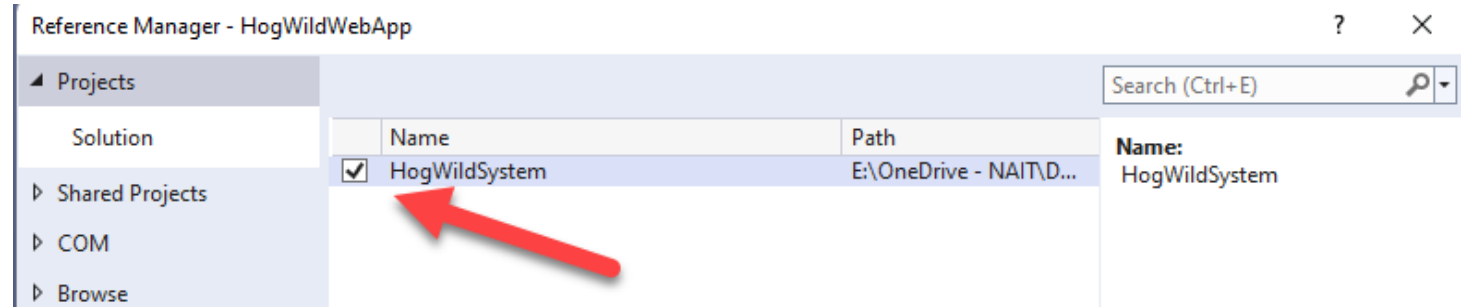




# Creating the System Library (Continue)

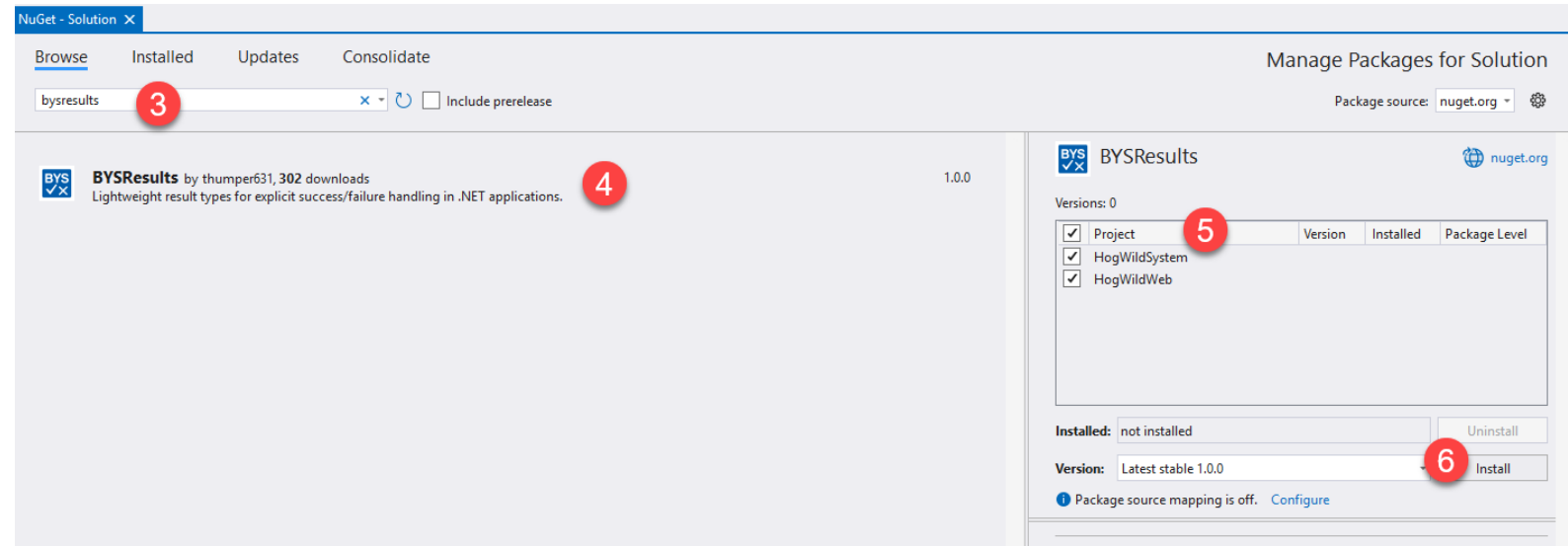
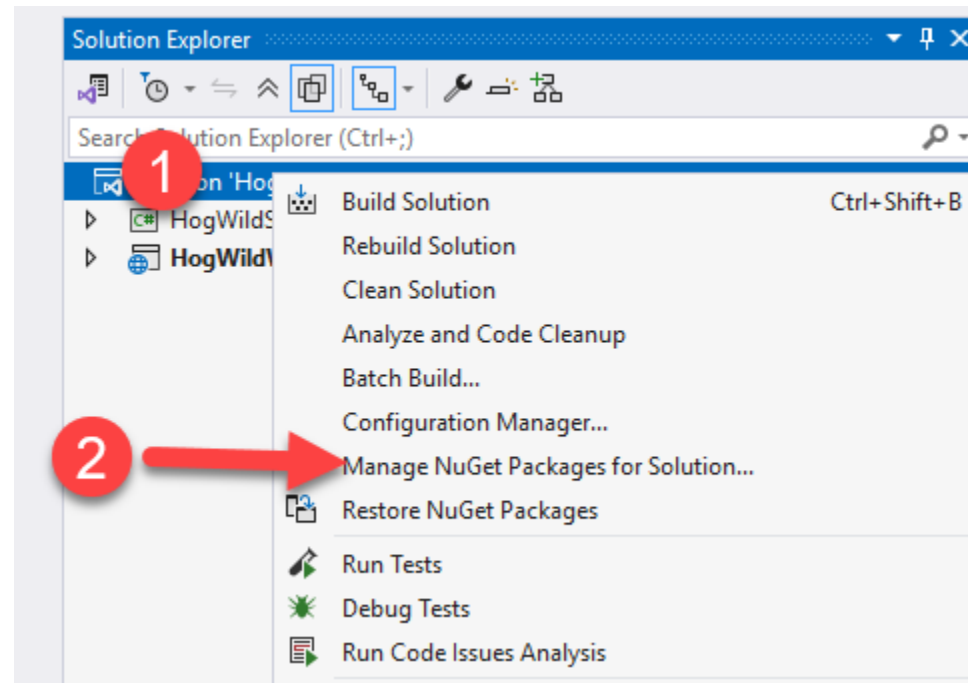
## 1.Usage in Blazor Application:

- Choose the "HogWildSystem" project as a reference.



# Add BYSResults Package

1. Right Click on the “Hogwild” Solution
2. Select “Manage NuGet Packages for Solutions”
3. Select “Browse” and type “BysResults”
4. Click on the BYSResults package
5. Select both the “HogWildSystem & HogWildWeb”
6. Click Install



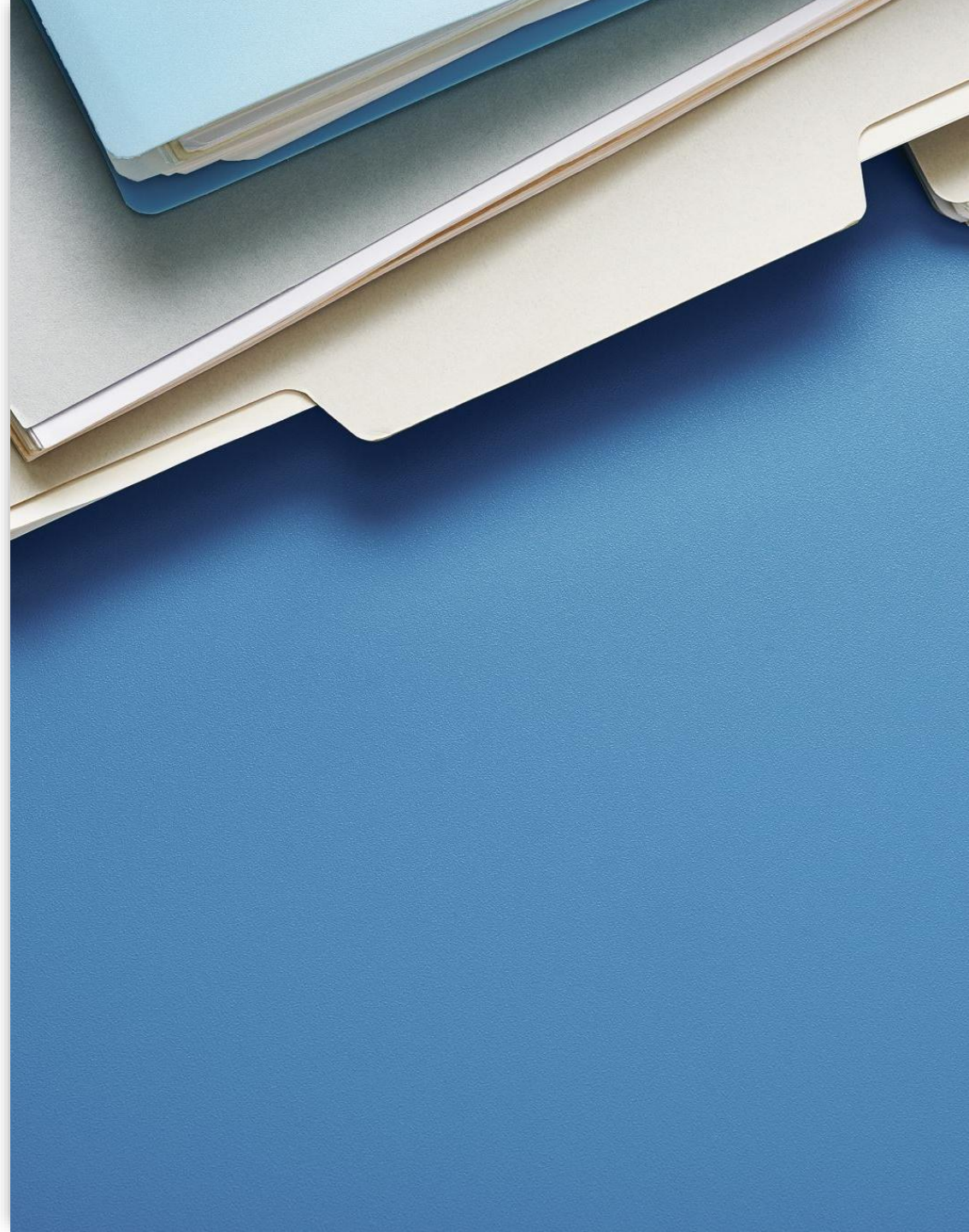
# The Backend Folder Structure

- **BLL (Business Logic Layer):**
- **Purpose:** The "BLL" folder stands for Business Logic Layer. This folder contains classes and components responsible for implementing the business logic of your application. It acts as an intermediary between the data access layer (DAL) and the presentation layer.
- **Contents:**
  - Business logic classes: These classes contain the core functionality of your application. They encapsulate the logic for processing and manipulating data.
  - Services: Often, the BLL folder includes service classes responsible for coordinating and managing various aspects of the application's functionality.
  - Validation: Any validation logic, such as input validation or business rule validation, can be placed in this folder.
  - Business workflows: Complex business processes or workflows can be organized and implemented within this folder.



# The Backend Folder Structure (Continue)

- **DAL (Data Access Layer):**
- **Purpose:** The "DAL" folder, or Data Access Layer, is responsible for all interactions with the database. It abstracts away the database-related details and provides a clean API for the BLL to access and manipulate data.
- **Contents:**
  - **DbContext:** This is where you define your Entity Framework DbContext. It represents your application's data model and manages the database connections.
  - **Entity Configuration:** This folder may contain classes that configure how your data entities map to database tables, defining relationships, keys, and constraints.





# The Backend Folder Structure (Continue)

- **Entities:**
- **Purpose:** The "Entities" folder contains the entity classes representing your application's data model. Each entity typically corresponds to a database table and defines the structure of the data.
- **Contents:**
  - Entity classes: These classes represent the tables or collections in your database. They include properties that map to database columns.
  - Relationships: Entity classes may define relationships with other entities, specifying how they are related in the database (e.g., one-to-many, many-to-many).



# The Backend Folder Structure (Continue)

- **ViewModels:**
- **Purpose:** The "ViewModels" folder contains classes representing the data your Blazor frontend needs. View models are used to shape and structure data for display and interaction in the user interface.
- **Contents:**
  - View model classes: Each view model class corresponds to a specific view or UI component in your Blazor application. These classes contain properties that match the data requirements of the front end.
  - Data transformation: View model classes may also include methods or logic for transforming data from backend entity models into a format suitable for presentation.
  - Validation: If client-side validation is needed, it can be implemented in these classes to ensure data integrity on the front end.

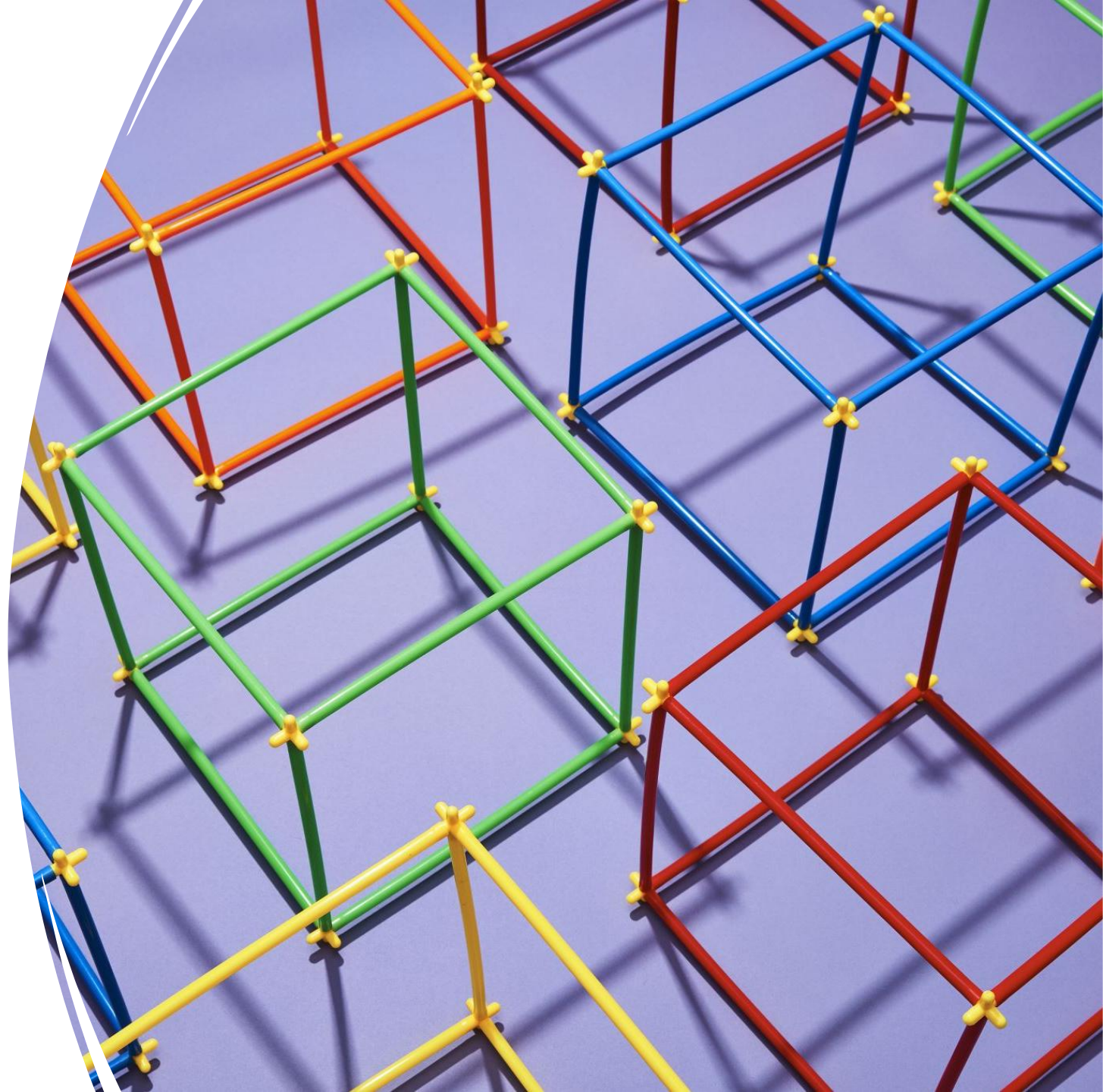




# Adding Folders and Files to the "HogWildSystem" Library

---

- 1. Organize Code:** Folders help categorize and structure code logically, making it easier to locate and manage specific components.
- 2. Enhance Collaboration:** A well-organized structure with meaningful folders and "dummy.txt" files improves collaboration by providing clarity and understanding, especially when working with a team or sharing code on platforms like GitHub.



# Adding Folders and Files to the "HogWildSystem" Library

---

## 1. Solution Explorer:

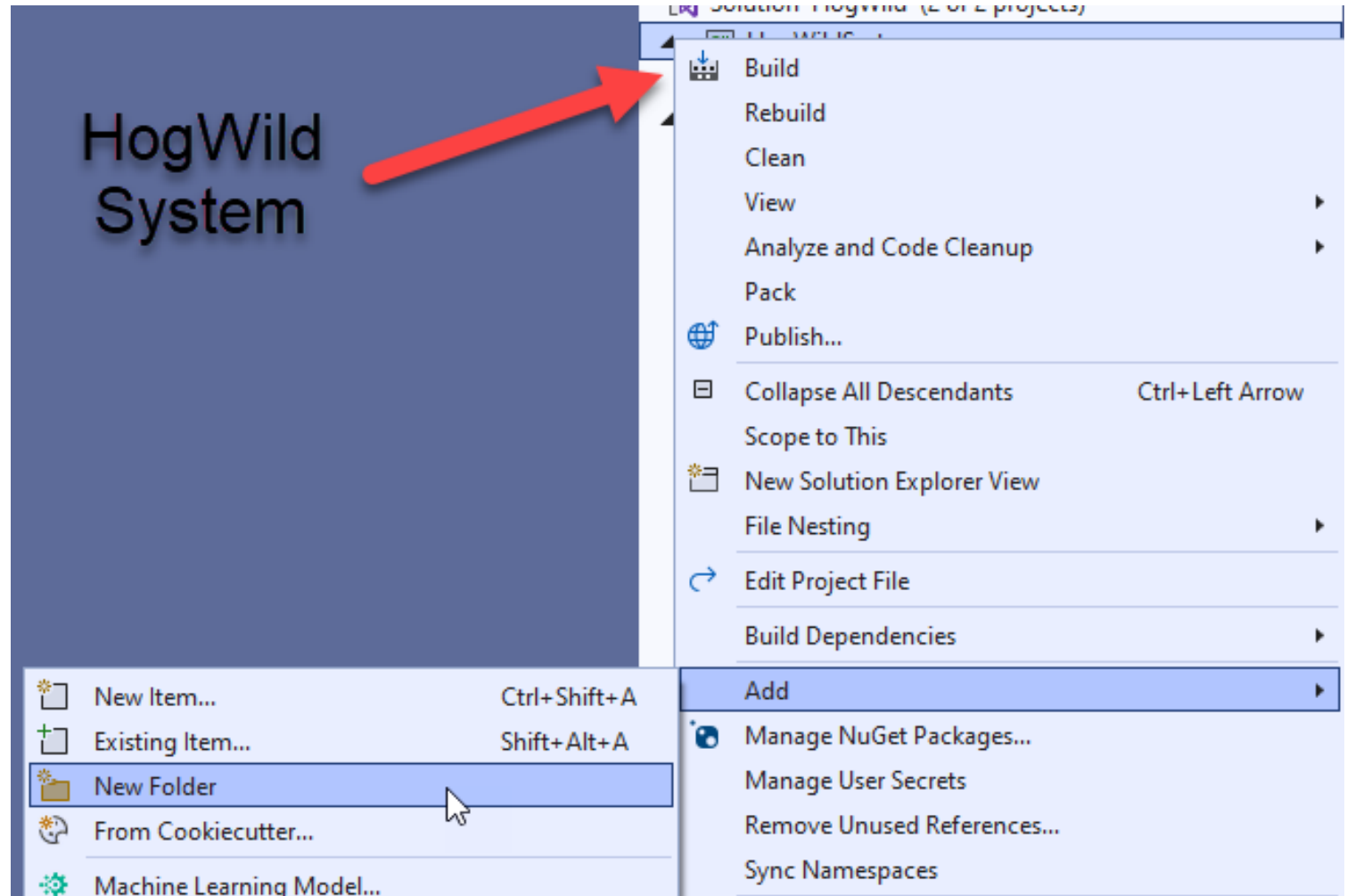
1. In the Solution Explorer window, locate and select the "HogWildSystem" project within your solution.

## 2. Adding Folders:

1. Right-click on the "HogWildSystem" project to open the context menu.

## 3. Add > New Folder:

1. From the context menu, select "Add" and then "New Folder."





Adding Folders and Files  
to the "HogWildSystem"  
Library

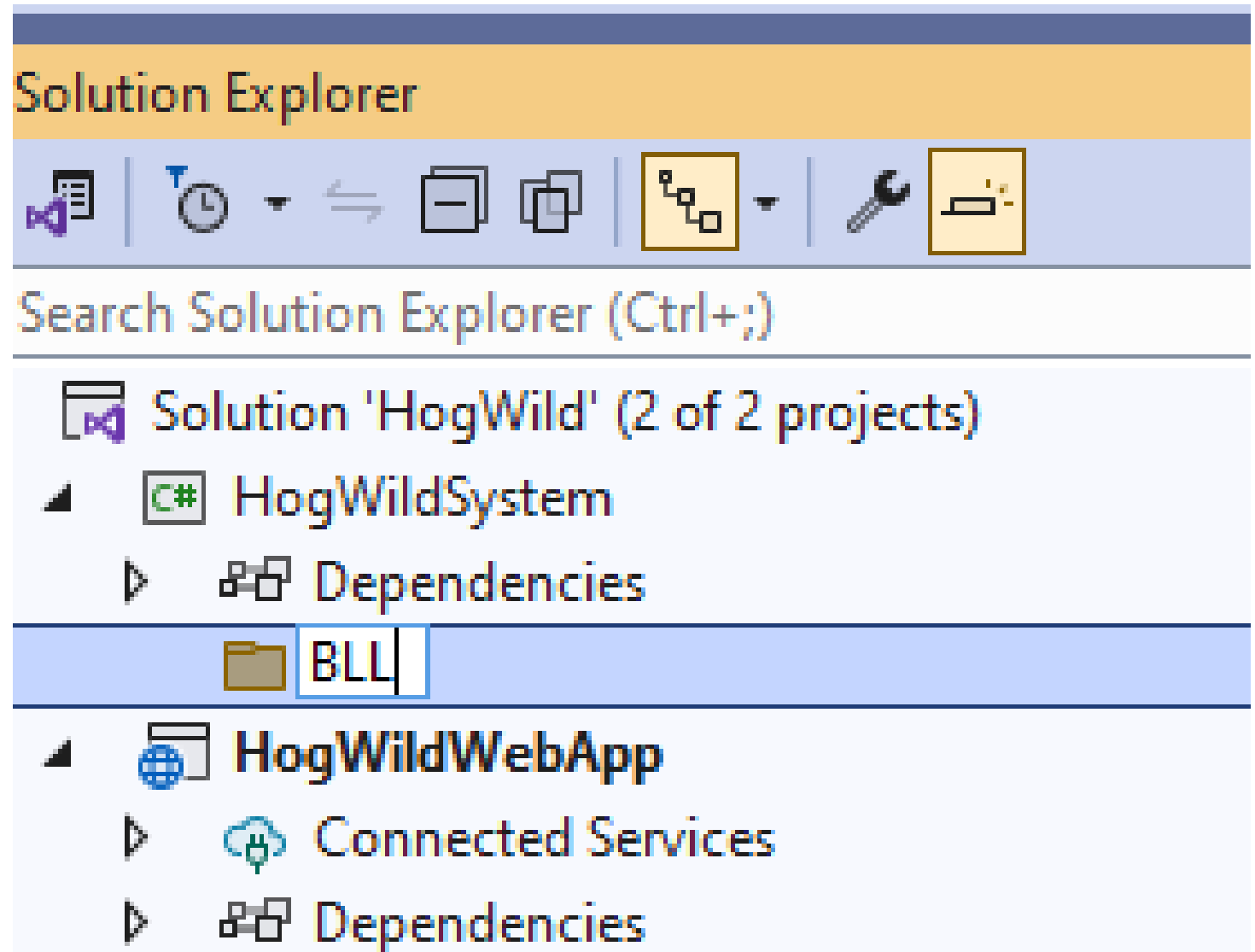
---

### 1.Folder Name:

1. Name the new folder "BLL" and press Enter to create it.

### 2.Repeat :

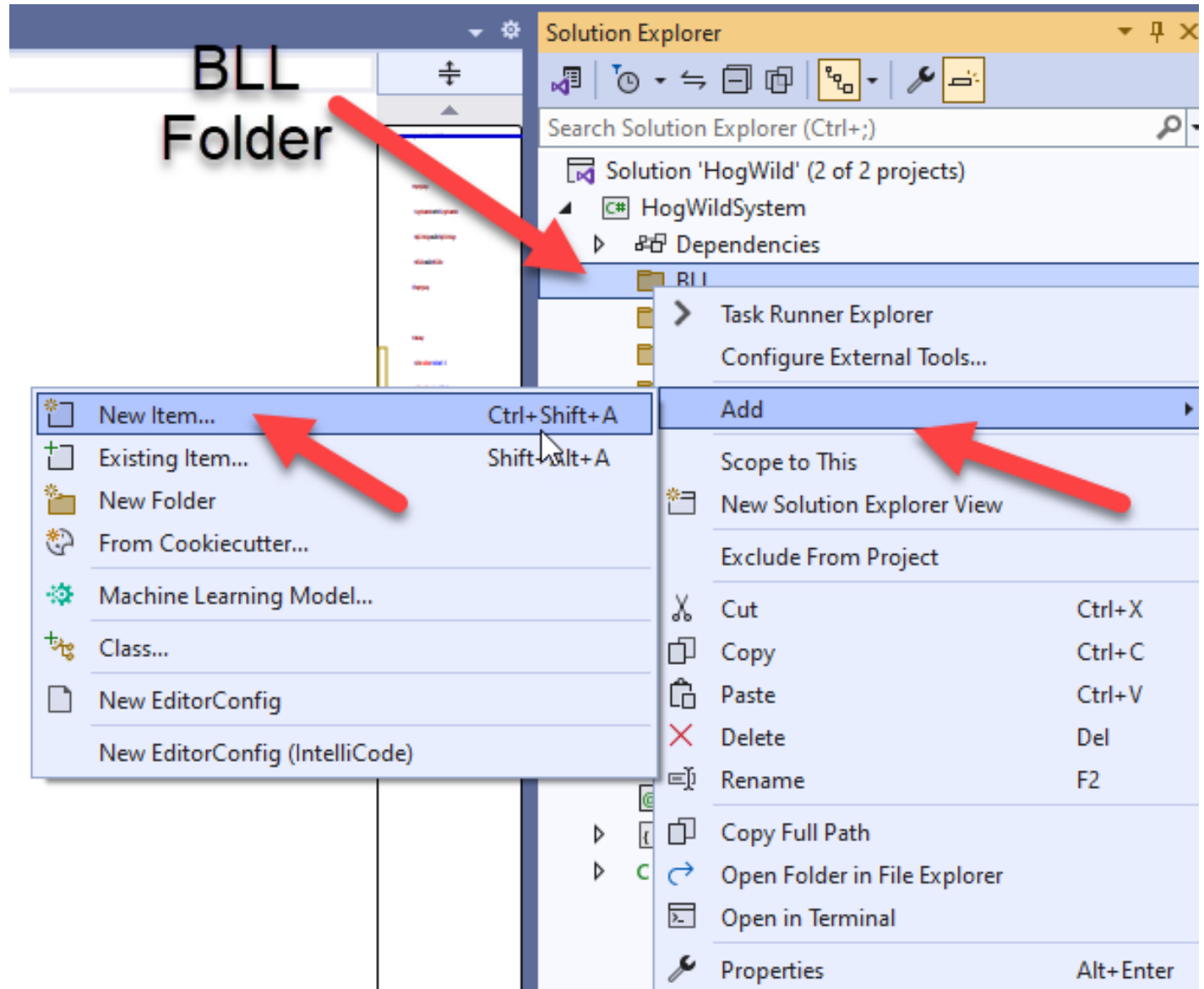
1. Repeat to create the following additional folders: "DAL," "Entities," and "ViewModels."



# Adding Folders and Files to the "HogWildSystem" Library

## 1. Adding "dummy.txt" Files:

1. Inside each of the created folders (BLL, DAL, Entities, and ViewModels), right-click on the folder, select "Add," and then "New Item."



# Adding Folders and Files to the "HogWildSystem" Library

## 1. Adding "dummy.txt" Files:

1. Inside each of the created folders (BLL, DAL, Entities, and ViewModels), right-click on the folder, select "Add," and then "New Item."

## 2. Add New Item Dialog:

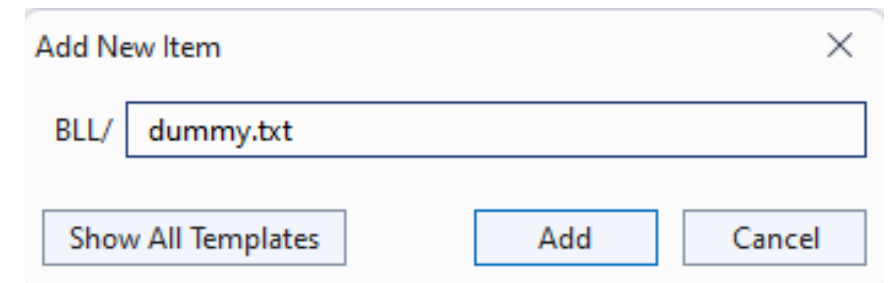
1. In the "Add New Item" dialog, rename "FileName.cs" to "dummy.txt."

## 3. Repeat for Each Folder:

1. Repeat steps 7 and step 8 for each of the four folders (BLL, DAL, Entities, and ViewModels).

## 4. Populate "dummy.txt" Files:

1. You leave it empty, as it is just a placeholder for pushing to GitHub.

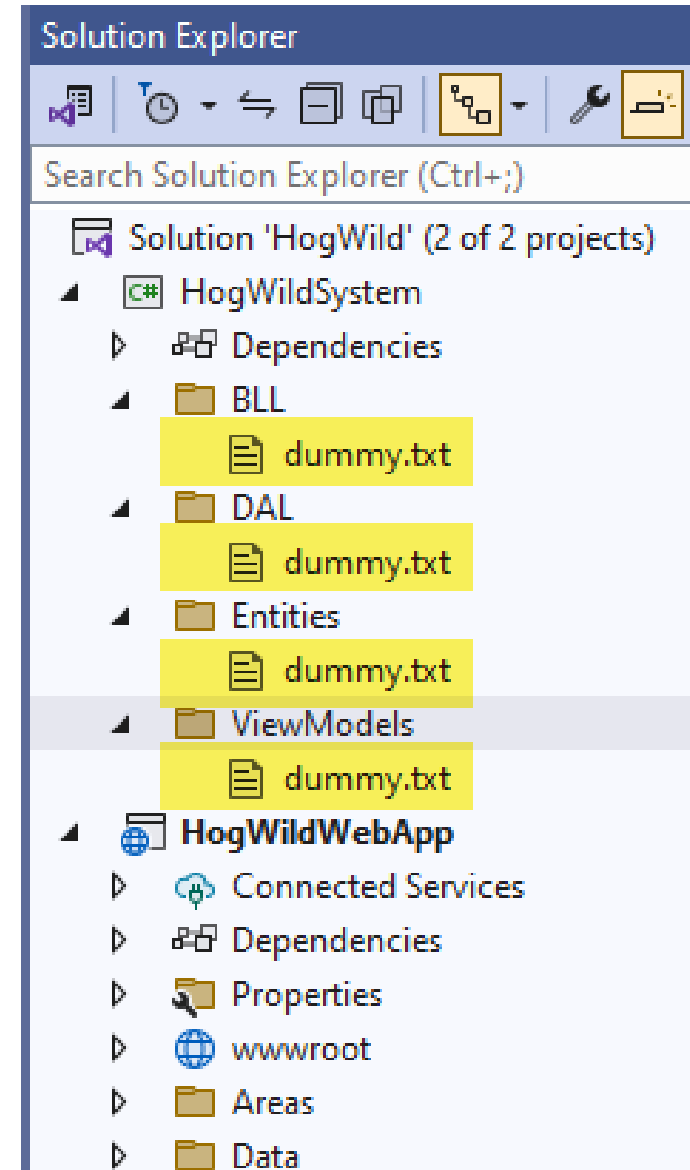


# Adding Folders and Files to the "HogWildSystem" Library

---

## 1. Populate "dummy.txt" Files:

1. You leave it empty, as it is just a placeholder for pushing to GitHub.



## Blazor Single Table Exercise – “Working Version”

- The exercise involves creating a Blazor page that interacts with a single table called "Working Version" to query and display a single record representing the database version. Here's a step-by-step overview of the exercise:
- **Objective:** Create a Blazor page with view models, entities, business logic (BLL), and dependency injection to retrieve and display the database version from the "Working Versions" table.
- **NOTE:** The table is listed as "WorkingVersions", however, the naming convention should be plural "WorkingVersion."







# Blazor Single Table Exercise – “Working Version”

## 1. Entity Model (Entities):

- Define an entity model for the "Working Version" table. This model should represent the structure of the table, including its fields and data types.

## 2. Database Context (DAL):

- Create a database context class that inherits from Entity Framework Core's `DbContext`. Configure it to include a `DbSet` for the "Working Version" entity.

## 3. Dependency Injection (HogWildExtension.cs):

- Set up dependency injection to inject your database context into the Blazor page.

# Blazor Single Table Exercise – “Working Version”

- **4. View Model (ViewModels):**
  - - Create a view model class that represents the data you want to display on the Blazor page. This view model should include a property to hold the database version.
- **5. Blazor Page (WorkingVersion.razor/  
WorkingVersion.razor.cs):**
  - - Create a Blazor page (Razor component) that will display the database version. Inject the database context and view model into this page.
- **6. Query Database Version (BLL):**
  - - In the Blazor page, write code to query the "Working Version" table using the injected database context. Retrieve the single record representing the database version and populate the view model property.
- **7. Display Version(WorkingVersion.razor/  
WorkingVersion.razor.cs):**
  - - Bind the view model property containing the database version to the page's UI elements (e.g., a label or text element) to display it to the user.



# Working Versions Table Structure

AMD-3900XT-A.OLT...o.WorkingVersions			
	Column Name	Data Type	Allow Nulls
▶ 🔑	VersionId	int	<input type="checkbox"/>
	Major	int	<input type="checkbox"/>
	Minor	int	<input type="checkbox"/>
	Build	int	<input type="checkbox"/>
	Revision	int	<input type="checkbox"/>
	AsOfDate	datetime	<input type="checkbox"/>
	Comments	nchar(50)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>



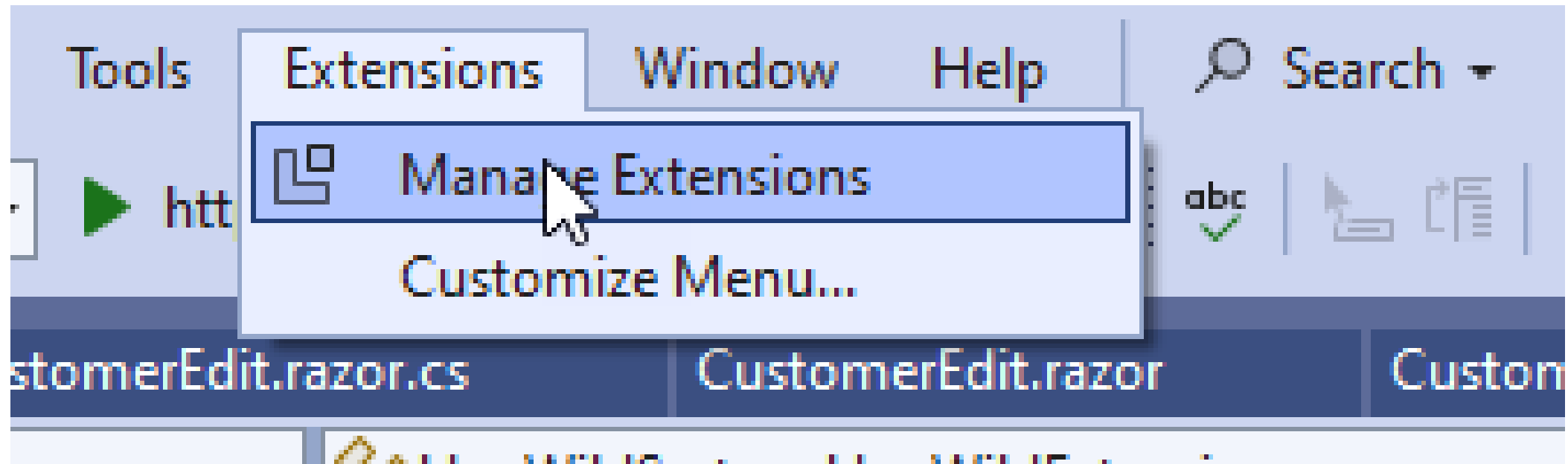
# Version Information

- Change the default values to the following values so that we can see the different values from the table.

[illegible]

# EF Power Tools

- Open Visual Studio:
  - Launch Visual Studio on your computer.
- Access Extensions and Updates:
  - Go to the "Extensions" menu in Visual Studio by clicking "Extensions" > "Manage Extensions."



# EF Power Tools

## 1. Browse for EF Power Tools:

1. In the "Extensions and Updates" window, click on the "Online" tab on the left sidebar.

## 2. Search for EF Power Tools:

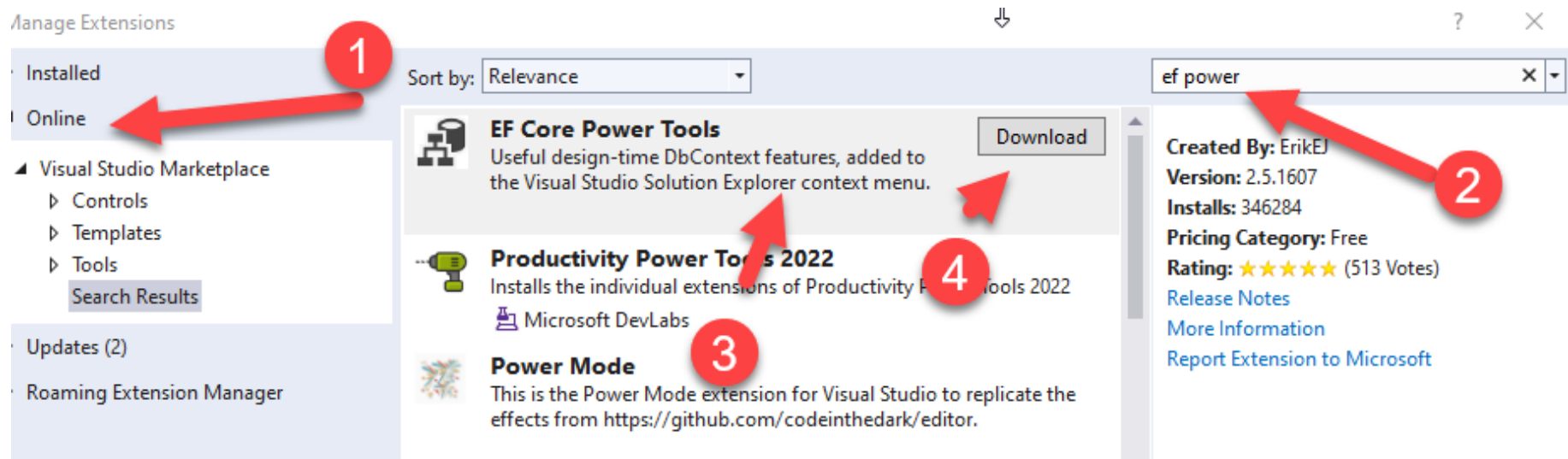
1. In the search bar on the upper-right corner, type "EF Power Tools" and press Enter.

## 3. Install EF Power Tools:

1. Locate "Entity Framework 6 Power Tools" in the search results. It should be an extension provided by Microsoft. Click on it.

## 4. Install Button:

1. Click the "Download" button to begin the installation process.



# EF Power Tools

---

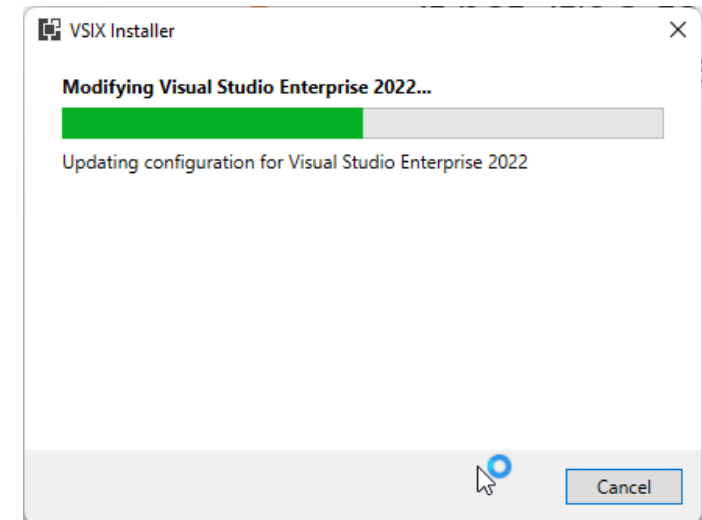
## 1.Restart Visual Studio:

- After installation, Visual Studio may prompt you to restart to complete the installation. If not, it's a good practice to restart Visual Studio manually to ensure the extension is properly loaded.

[Change your settings for Extensions](#)

⚠ Your changes will be scheduled. The modifications will begin when all Microsoft Visual Studio windows are closed.

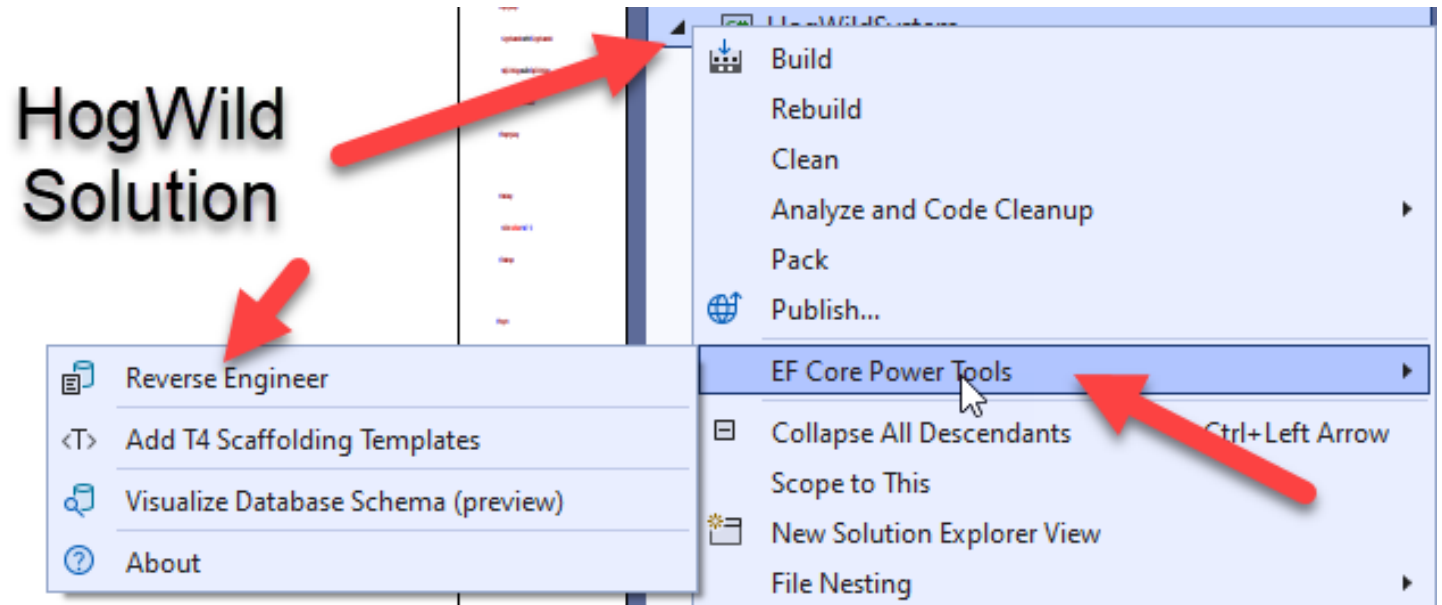
Close



# Reverse Engineering using EF Core Power Tools

---

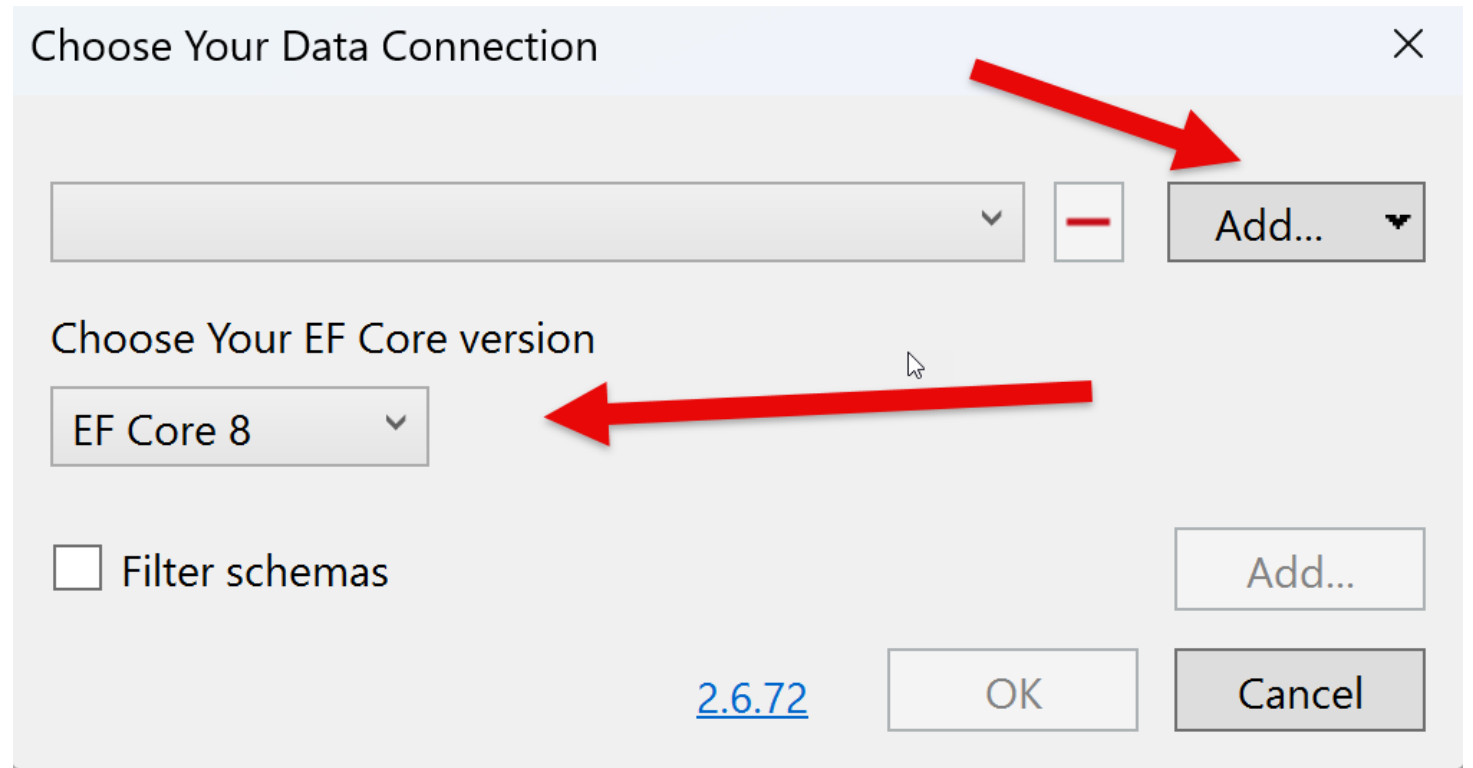
- Right-click on HogWildSystem and select “EF Core Power Tools” and then “Reverse Engineer”.



# Reverse Engineering using EF Core Power Tools

---

- Ensure that you are using EF Core 8 and click the “Add...” and select “Add Database Connection”



# Reverse Engineering using EF Core Power Tools

---

1. Select your server's name.
2. Select the OLTP-DMIT2018 Database
3. Test Connection

Connection Properties

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (Microsoft SqlClient) Change...

Server name: . Refresh

Log on to the server

Authentication: Windows Authentication

User name:

Password:

☐ Save my password

Connect to a database

☒ Select or enter a database name: OLTP-DMIT2018

☐ Attach a database file:  Browse...

Logical name:

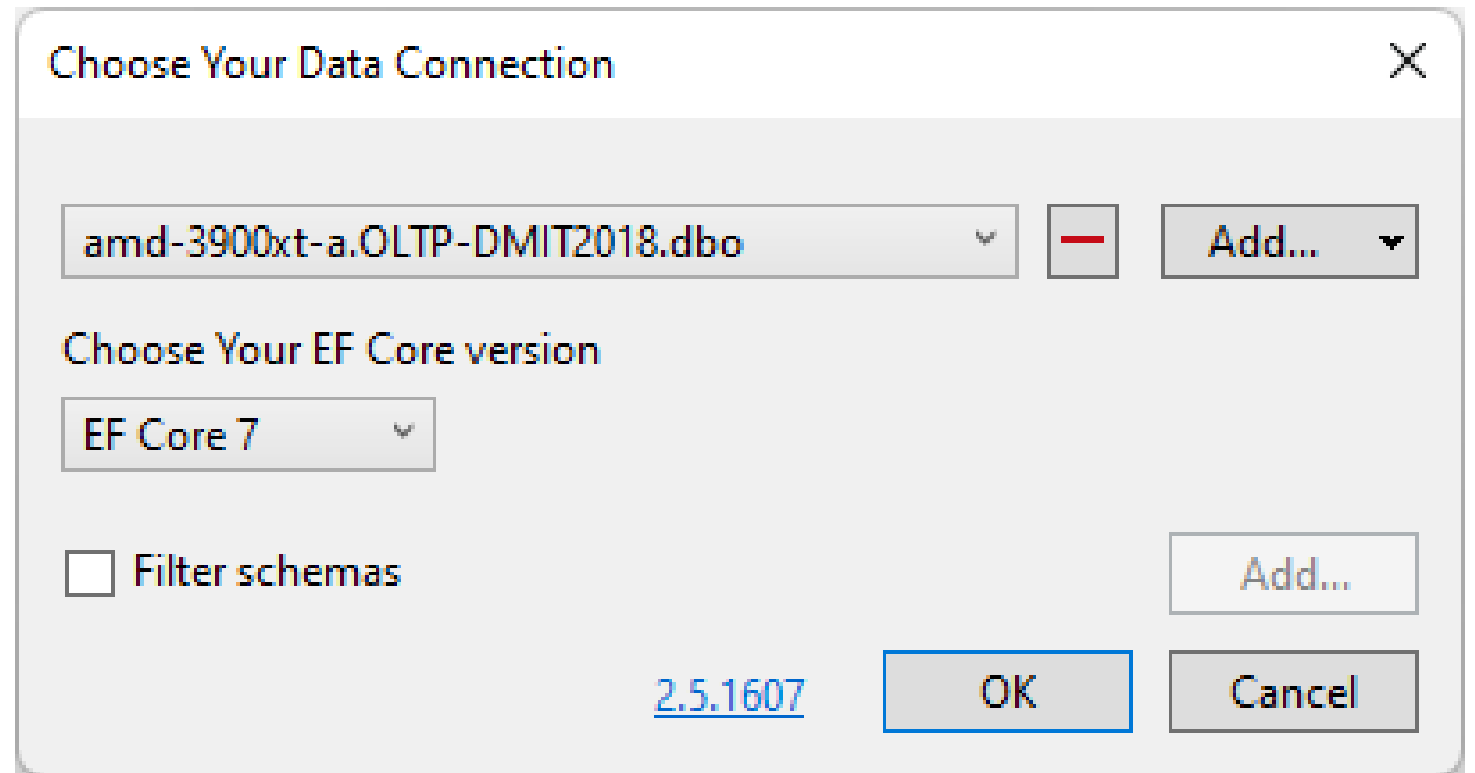
Advanced...

Test Connection OK Cancel

# Reverse Engineering using EF Core Power Tools

---

1. Press OK

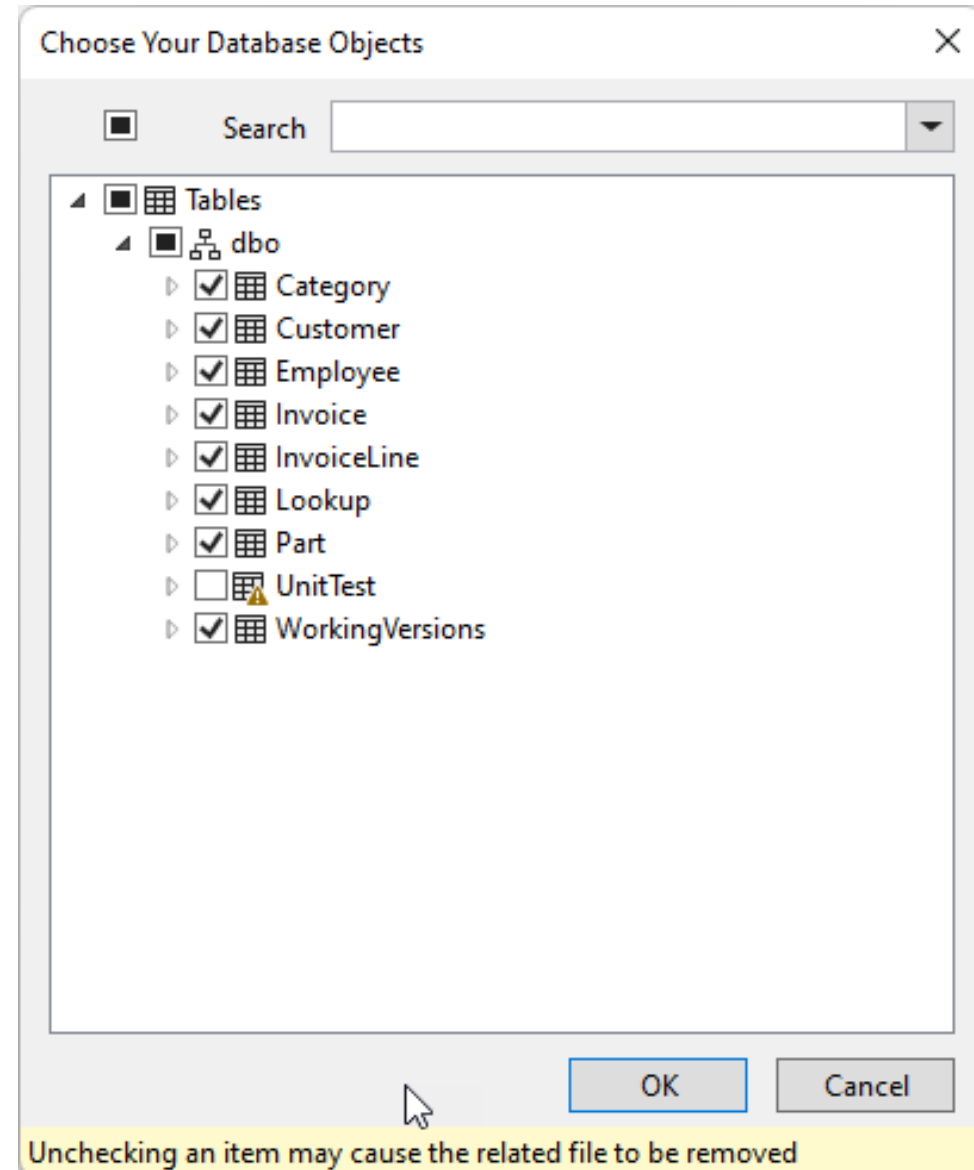




# Reverse Engineering using EF Core Power Tools

---

1. Select all tables except  
for “Unit Test”



# Reverse Engineering using EF Core Power Tools

---

## 1. Context Name:

- Set the "Context Name" property to "HogWildContext." This defines the name of the DbContext class that will be generated.

## 2. Namespace:

- Set the "Namespace" property to "HogWildSystem." This determines the namespace for the generated EF Core model classes.

## 3. Entity Types:

- Set the "Entity Types" property to "Entities." This specifies the folder or namespace where the generated entity classes will be placed.

The screenshot shows the 'Choose Your Settings for Project HogWildSystem' dialog box. It contains the following settings and callouts:

- Context name:** HogWildContext (Callout 1)
- Namespace:** HogWildSystem (Callout 2)
- EntityTypes path (f.ex. Models) - optional:** Entities (Callout 3)
- What to generate:** EntityTypes & DbContext (Callout 4)
- Naming:**
  - ☒ Pluralize generated object names (English) (Callout 5)
  - ☒ Use table and column names directly from the database (Callout 6)
- ☒ Use DataAnnotation attributes to configure the model (Callout 7)
- ☐ Customize code using templates: C# - Handlebars (Callout 8)
- ☐ Include connection string in generated code
- ☒ Install the provider package in the project

At the bottom, there are links for [Help](#), [★ Rate](#), and buttons for [Advanced](#), [OK](#), and [Cancel](#).

# Reverse Engineering using EF Core Power Tools

---

## 4. Pluralize:

- Set the "Pluralize" property to "True" if you want EF Core Power Tools to pluralize table names when generating entity classes automatically. For example, if you have a table named "User," it will generate an entity named "Users."

## 5. Use Table and Column Names:

- Set the "Use Table and Column Names" property to "True" if you want the generated entity classes and properties to match your database's exact names of tables and columns. This ensures that there is no name transformation.

The screenshot shows the 'Choose Your Settings for Project HogWildSystem' dialog box. It contains the following settings and callouts:

- Context name:** HogWildContext (Callout 1)
- Namespace:** HogWildSystem (Callout 2)
- EntityTypes path (f.ex. Models) - optional:** Entities (Callout 3)
- What to generate:** EntityTypes & DbContext (Callout 4)
- Naming:**
  - ☒ Pluralize generated object names (English) (Callout 5)
  - ☒ Use table and column names directly from the database (Callout 6)
- ☒ Use DataAnnotation attributes to configure the model (Callout 7)
- ☐ Customize code using templates: C# - Handlebars (Callout 8)
- ☐ Include connection string in generated code
- ☒ Install the provider package in the project

At the bottom, there are links for [Help](#), [★ Rate](#), and buttons for [Advanced](#), [OK](#), and [Cancel](#).

# Reverse Engineering using EF Core Power Tools

---

## 6. Use DataAnnotation Attributes:

- Set the "Use DataAnnotation Attributes" property to "True" if you want EF Core Power Tools to use data annotation attributes (such as [Key], [MaxLength], etc.) to apply data annotations to your generated entity classes based on the database schema.

## 7. Install the EF Core provider package

- EF Core Database Provider Packages: Entity Framework Core is designed to be extensible, allowing developers to use it with various databases. EF Core provides a set of database provider packages that you can install in your project to enable this compatibility. For example:
- Microsoft.EntityFrameworkCore.SqlServer: This is the EF Core provider package for SQL Server databases.

The screenshot shows the 'Choose Your Settings for Project HogWildSystem' dialog box. It contains the following fields and options:

- Context name:** HogWildContext (Annotation 1)
- Namespace:** HogWildSystem (Annotation 2)
- EntityTypes path (f.ex. Models) - optional:** Entities (Annotation 3)
- What to generate:** EntityTypes & DbContext (dropdown menu)
- Naming section:**
  - ☒ Pluralize generated object names (English) (Annotation 4)
  - ☒ Use table and column names directly from the database (Annotation 5)
- ☒ Use DataAnnotation attributes to configure the model (Annotation 6)
- ☐ Customize code using templates: C# - Handlebars (dropdown menu)
- ☐ Include connection string in generated code
- ☒ Install the provider package in the project (Annotation 7)

At the bottom, there are links for [Help](#), [★ Rate](#), and buttons for [Advanced](#) (Annotation 8), [OK](#), and [Cancel](#).

# Reverse Engineering using EF Core Power Tools (Advance...)

---

## 6. Advance:

- DbContext Path (Advanced):  
When you reverse engineer an existing database using EF Core Power Tools, it generates a DbContext class that represents your database and provides access to its tables and data. The "DbContext Location" setting allows you to specify where this generated DbContext class should be placed in your project.

Advanced...

Code generation

- ☐ Remove default DbContext constructor
- ☐ Remove SQL default from bool columns
- ☐ Use nullable reference types (experimental)
- ☐ Use many to many entity (EF Core 6 and later)
- ☐ Always include all database objects

File layout

- ☐ Split DbContext into Configuration classes (preview)
- ☐ Use schema folder separation (experimental)
- ☐ Use schema namespaces separation (experimental)

EntityTypes sub-namespace (overrides path) - optional

DbContext path (f.ex. Data) - optional

DAL

DbContext sub-namespace (overrides path) - optional

Mapping

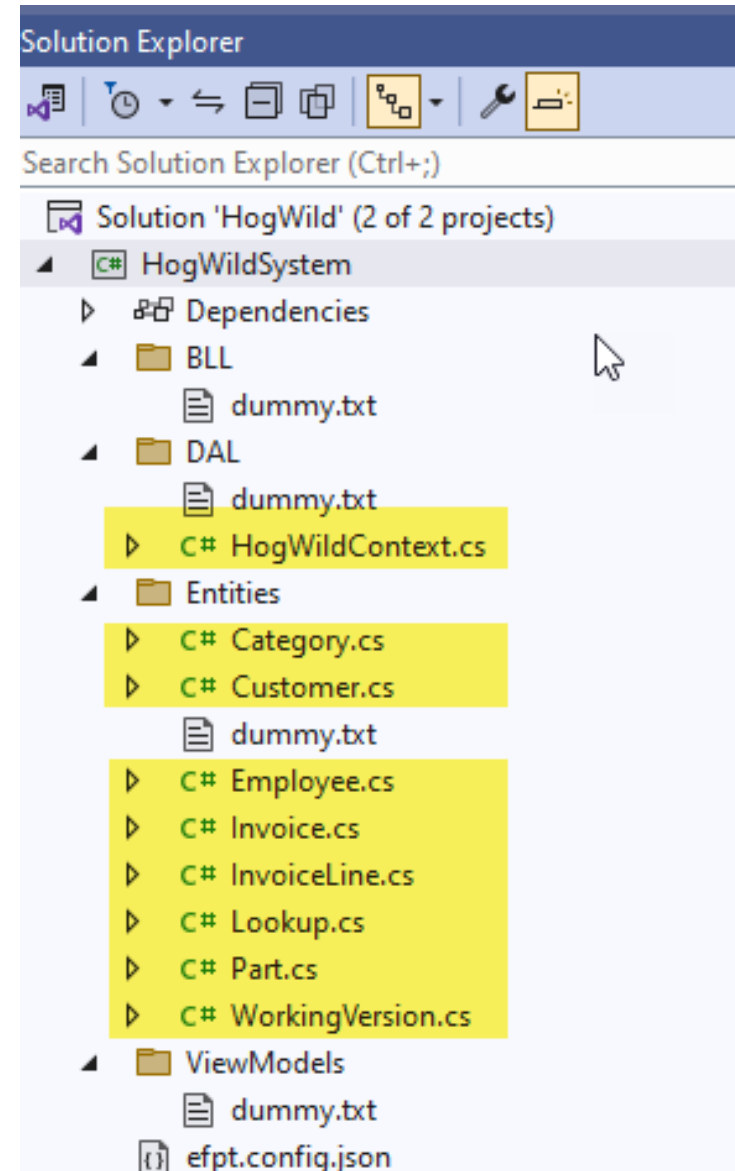
- ☐ Map spatial types
- ☐ Map hierarchyid
- ☐ Map DateOnly and TimeOnly
- ☐ Map Noda Time types
- ☐ Use EF6 pluralizer

OK Cancel

# Reverse Engineering using EF Core Power Tools - Final

---

- HogWildContext.cs has been added to the DAL folders.
- Entities have been created for the database tables and they have been added to the Entities folder.



# HogWild Context File

- This file represents the essential components of a DbContext class in EF Core, providing a connection to the database, defining DbSet properties for entities, configuring the database model, and setting up relationships and constraints between entities for data access and manipulation.
- **File Purpose:** This file is auto-generated by EF Core Power Tools and serves as the DbContext (Database Context) class for an Entity Framework Core (EF Core) data access layer within the "HogWildSystem.DAL" namespace.
- **Database Context:** The primary purpose of this file is to define the "HogWildContext" class, which inherits from the EF Core "DbContext" class. This class acts as the bridge between the application and the underlying database.

# Refactoring HogWild Context

---

## HogWildContext.cs

- **Separation of Concerns and Enhanced Security:**
  - By making the DbContext internal, you encourage a clear separation of concerns, ensuring that components responsible for data access are encapsulated within the data access layer. This separation promotes maintainability and testability and enhances security by limiting the exposure of sensitive database operations to external code or unauthorized users.

```
2 references
internal partial class HogWildContext : DbContext
{
    0 references
    public HogWildContext(DbContextOptions<HogWildContext> options)
        : base(options)
    {
    }

    0 references
    public virtual DbSet<Category> Categories { get; set; }
}
```



# Refactoring Entity Classes

---

- **Entity Classes:**
  - While entity classes themselves don't necessarily need to be marked as internal (they are often entity framework-generated and represent the database schema), their accessibility should be considered based on your design and architectural choices
  - **Repeat this for all classes in the Entities folder.**

```
[Table("Category")]
```

2 references

```
internal partial class Category
```

```
{
```

```
    [Key]
```

0 references

```
    public int CategoryID { get; set; }
```

```
[Table("Customer")]
```

9 references

```
internal partial class Customer
```

```
{
```

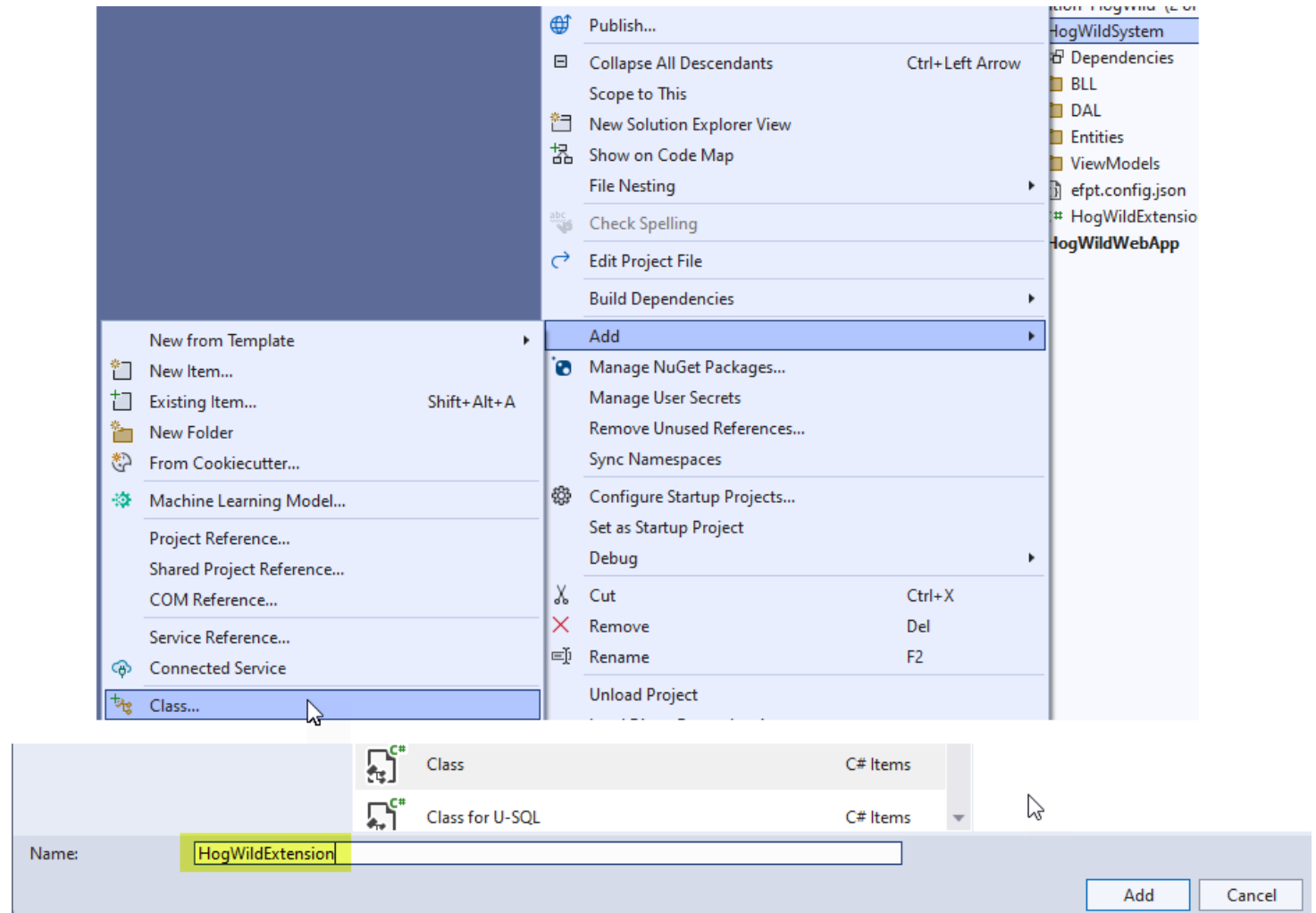
```
    [Key]
```

0 references

```
    public int CustomerID { get; set; }
```

# HogWildExtension

- This HogWildExtension class simplifies the registration of backend-related services and dependencies in an ASP.NET Core application. It is designed to be used in the **Program.cs** file of the web application to configure the services required for data access and business logic, including the DbContext and various service classes.



## HogWild Extensions (Code)

# HogWildExtension.cs

```
namespace HogWildSystem
{
    0 references
    public static class HogWildExtension
    {
        // This is an extension method that extends the IServiceCollection interface.
        // It is typically used in ASP.NET Core applications to configure and register services.

        // The method name can be anything, but it must match the name used when calling it in
        // your Program.cs file using builder.Services.XXXX(options => ...).
        0 references
        public static void AddBackendDependencies(this IServiceCollection services,
            Action<DbContextOptionsBuilder> options)
        {
            // Register the HogWildContext class, which is the DbContext for your application,
            // with the service collection. This allows the DbContext to be injected into other
            // parts of your application as a dependency.

            // The 'options' parameter is an Action<DbContextOptionsBuilder> that typically
            // configures the options for the DbContext, including specifying the database
            // connection string.

            services.AddDbContext<HogWildContext>(options);
        }
    }
}
```

# Adding Connection String

- **Open appsettings.json:** In your HogWildWebApp, locate the **appsettings.json** file. This file is found in the project's root directory. Open it for editing.
- **Add the Connection String:** Within the **appsettings.json** file, you can add your connection string under the **"ConnectionStrings"** section.

## appsettings.json

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=.;Database=OLTP-DMIT2018;Trusted_Connection=true;TrustServerCertificate=True;MultipleActiveResultSets=true"
  },
  "OLTP-DMIT2018": "Server=.;Database=OLTP-DMIT2018;Trusted_Connection=true;TrustServerCertificate=True;MultipleActiveResultSets=true"
}
```

# Program.cs

---

- Sets up the web application, configures services, middleware, and routing, and launches the application. It also handles database connections and configuration related to Identity and Entity Framework



# Program.cs - Connection Strings (Given)

- `// Add services to the container.`
- `// :given (This is code that is provided when we create our application)`
- `// supplied database connection due to the fact that we created this`
- `// web app to use Individual accounts`
- `// Core retrieves the connection string from appsettings.json`
- `var connectionString = builder.Configuration.GetConnectionString("DefaultConnection") ?? throw new InvalidOperationException("Connection string 'DefaultConnection' not found.");`

Program.cs

# Program.cs - Connection Strings (Added)

- `var connectionString = builder.Configuration.GetConnectionString("DefaultConnection") ?? throw new InvalidOperationException("Connection string 'DefaultConnection' not found.");`
- `// :added`
- `// code retrieves the HogWild connection string`
- `var connectionStringHogWild = builder.Configuration.GetConnectionString("OLTP-DMIT2018");`

Program.cs

# Program.cs – Services (Given)

- `var connectionStringHogWild = builder.Configuration.GetConnectionString("OLTP-DMIT2018");`
- `// :given`
- `// register the supplied connections string with the IServiceCollection (.Services)`
- `// Register the connection string for individual accounts`
- `builder.Services.AddDbContext<ApplicationDbContext>(options =>`
- `options.UseSqlServer(connectionString));`

Program.cs



# Program.cs – Services (Added) Program.cs

- `builder.Services.AddDbContext<ApplicationDbContext>(options =>`
- `options.UseSqlServer(connectionString));`
- `// added:`
- `// Code the logic to add our class library services to IServiceCollection`
- `// One could do the registration code here in Program.cs`
- `// HOWEVER, every time a service class is added, you would be changing this file`
- `// The implementation of the DBContent and AddTransient(...) code in this`
- `// example will be done in an extension method to IServiceCollection`
- `// The extension method will be code inside the HogWildSystem class library`
- `// The extension method will have a paramater: options.UseSqlServer()`
- `builder.Services.AddBackendDependencies(options =>`
- `options.UseSqlServer(connectionStringHogWild));`

# Program.cs – Final Code

## Program.cs

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// :given (This is code that is provided when we create our application)
// supplied database connection due to the fact that we created this
// web app to use Individual accounts
// Core retrieves the connection string from appsettings.json
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection") ?? t

// :added
// code retrieves the HogWild connection string
var connectionStringHogWild = builder.Configuration.GetConnectionString("OLTP-DMIT2018");

// :given
// register the supplied connections string with the IServiceCollection (.Services)
// Register the connection string for individual accounts
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));

// added:
// Code the logic to add our class library services to IServiceCollection
// One could do the registration code here in Program.cs
// HOWEVER, every time a service class is added, you would be changing this file
// The implementation of the DbContext and AddTransient(...) code in this example
// will be done in an extension method to IServiceCollection
// The extension method will be code inside the HogWildSystem class library
// The extension method will have a paramater: options.UseSqlServer()
builder.Services.AddBackendDependencies(options =>
    options.UseSqlServer(connectionStringHogWild));
```

# Business Logic Layer

## 1. Purpose of BLL:

- The Business Logic Layer (BLL) in Blazor is responsible for encapsulating the application's business logic and data access operations.
- It separates the presentation layer (Blazor components) from direct database access, promoting a clean and maintainable architecture.

## 2. Naming Convention:

- In accordance with your naming scheme, each service in the BLL should be named with an "Area" prefix followed by "Service," e.g., **WorkingVersionsService**.

## 3. DbContext Usage:

- The BLL interacts with the database through Entity Framework's DbContext. Each service should have access to the appropriate DbContext for its designated area of responsibility.
- DbContext facilitates CRUD (Create, Read, Update, Delete) operations on the database.

# Business Logic Layer

## **4.Data Validation and Transformation:**

- The BLL is responsible for validating incoming data, ensuring it adheres to business rules and constraints.
- It may also involve transforming data between the format used in the database and the format needed by the Blazor components.

## **5.Abstraction of Data Access:**

- BLL services abstract the details of data access, shielding the components from the complexities of database interactions.
- This abstraction allows for easier unit testing of business logic without the need for an actual database connection.

## **6.Dependency Injection:**

- BLL services are typically registered with the Dependency Injection (DI) container in your Blazor application's **Program.cs**.
- This enables the components to request and use the services via constructor injection, promoting loose coupling and maintainability.

# Business Logic Layer

## 7. Encapsulation of Operations:

- Each service should provide methods that encapsulate specific business operations. For example, **WorkingVersionsService** might have methods like **CreateVersion**, **GetVersionById**, **EditVersion**, and **DeleteVersion**.

## 8. Error Handling and Logging:

- The BLL should handle exceptions and errors gracefully, providing meaningful feedback to the user when an operation fails.
- Logging can be incorporated to track and diagnose issues within the business logic.

## 9. Security Considerations:

- The BLL can enforce security measures, such as user authorization and authentication, to ensure that only authorized users can perform certain operations.

## 10. Maintenance and Scalability:

- By organizing business logic into separate BLL services, the application becomes more modular and easier to maintain and scale. Changes to one area of the application are less likely to impact others.

# Working Version Service – Initial Setup

- Services are commonly used to interact with a database or other data sources to retrieve and provide data to other parts of the application.
- The pattern for our services will be:
  - `// This field holds a reference to an instance of the HogWildContext`
  - `private readonly HogWildContext? _hogWildContext;`
  - `// The constructor is used for dependency injection, specifically to inject an instance of HogWildContext into the service`
  - `internal WorkingVersionsService(HogWildContext hogWildContext)`
    - `{`
    - `_hogWildContext = hogWildContext;`
    - `}`
- NOTE: Your access modifier must be set to “**internal**” so that only the HogWildExtension has access to it.





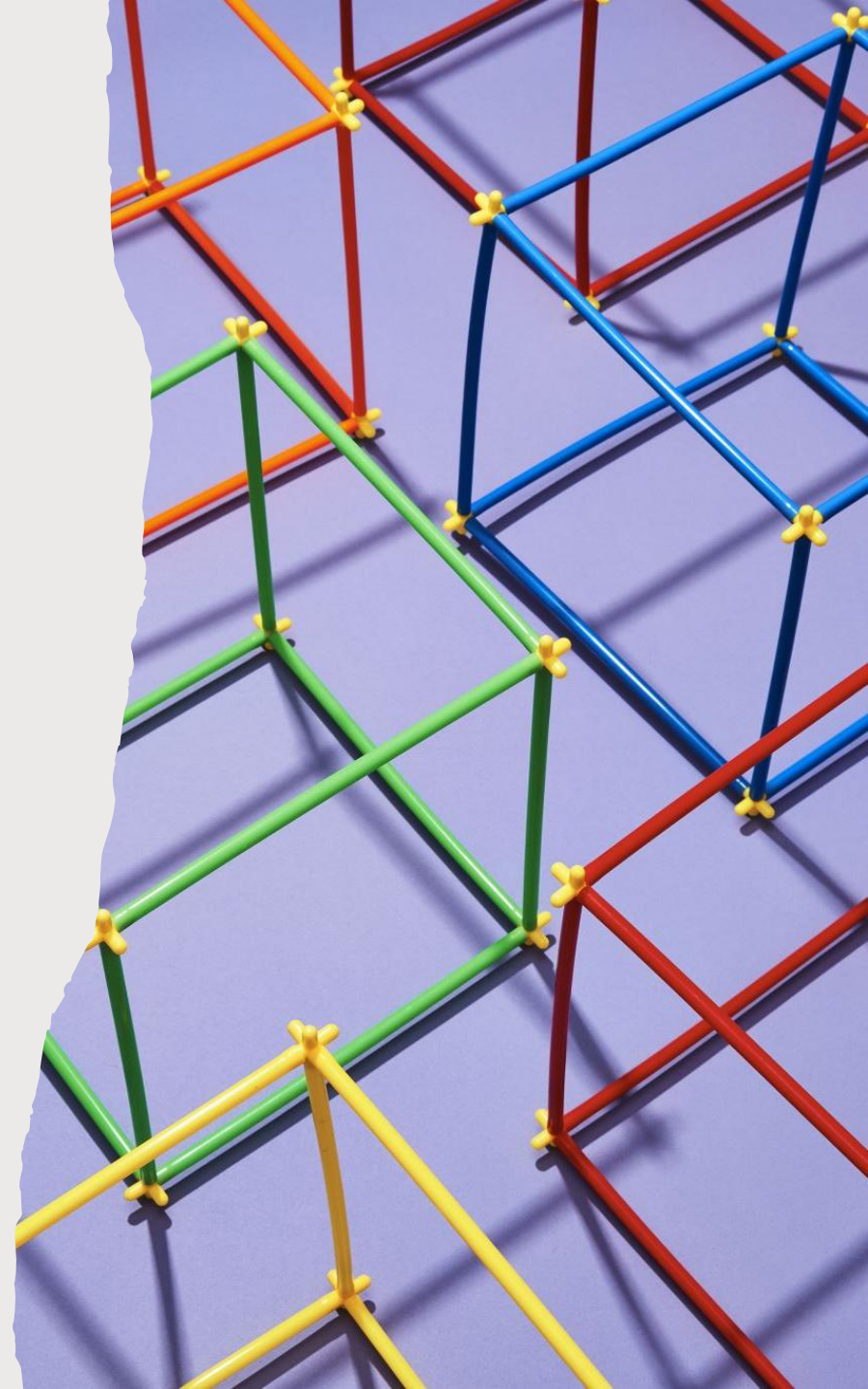
# Working Versions Service – Initial Setup (Continue)

## 1. Dependency Injection:

- The **WorkingVersionsService** relies on dependency injection to receive an instance of the **HogWildContext**. This is typically done to abstract data access and promote testability.
- The DbContext (**HogWildContext**) represents the database context for Entity Framework, which is used to interact with the database.

## 2. Initialization:

- Inside the constructor, the **\_hogWildContext** field is initialized with the **hogWildContext** parameter passed to the constructor. This allows the service to use this DbContext instance to interact with the database.



---



# Initial Setup of Working Versions

- You must follow this pattern for any services you set up.
- In C#, readonly is used for fields that can only be assigned a value during declaration or within the constructor. Once assigned, their values cannot be changed, making them effectively constant and safe for multi-threaded access.

## WorkingVersionsService.cs

```
#nullable disable
using HogWildSystem.DAL;

namespace HogWildSystem.BLL
{
    3 references
    public class WorkingVersionsService
    {
        #region Fields
        /// <summary>
        /// The hog wild context
        /// </summary>
        private readonly HogWildContext _hogWildContext;

        #endregion

        // Constructor for the WorkingVersionsService class.
        1 reference
        internal WorkingVersionsService(HogWildContext hogWildContext)
        {
            // Initialize the _hogWildContext field with the provided HogWildContext instance.
            _hogWildContext = hogWildContext;
        }
    }
}
```

Your constructor modifier must be set to "internal"

# HogWildExtension.cs

## Adding Working Version Service to HogWildExtension.cs

---

- Registers WorkingVersionService with dependency injection and specifies how to create instances of it when requested. The service will be created each time requested, making it a transient service.

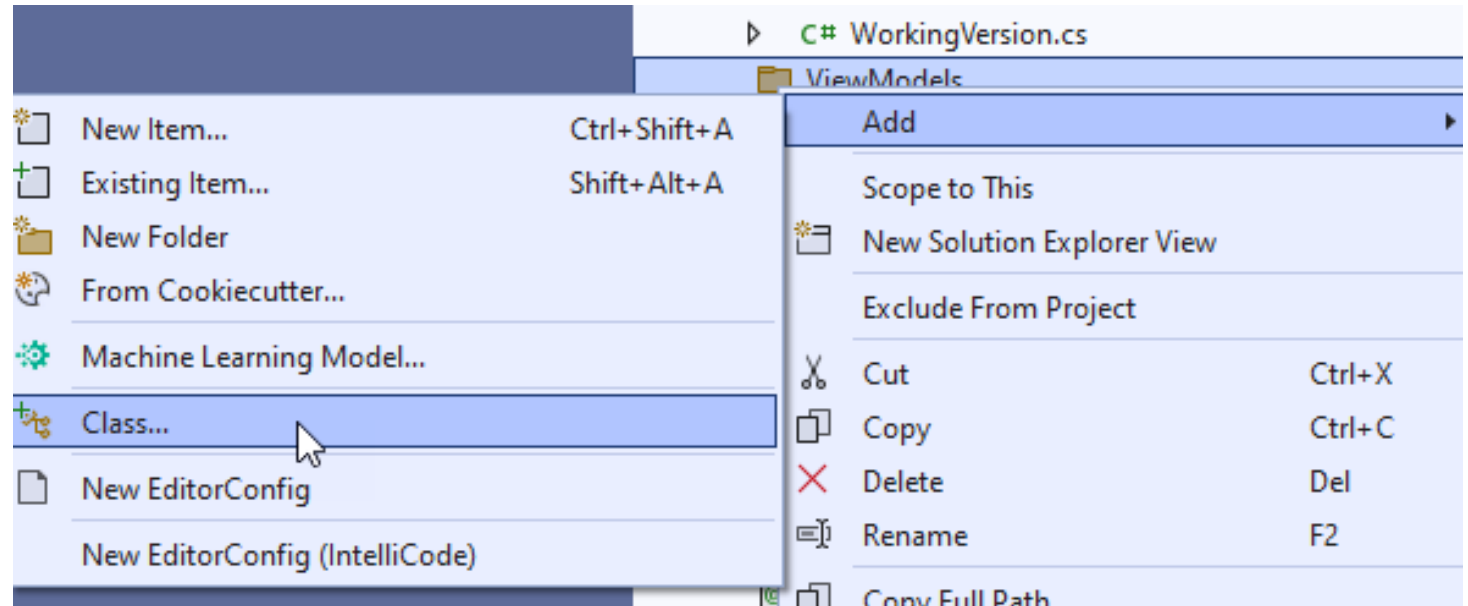
```
services.AddDbContext<HogWildContext>(options);
```

```
// adding any services that you create in the class library (BLL)
// using .AddTransient<t>(...)
// working versions
services.AddTransient<WorkingVersionsService>((ServiceProvider) =>
{
    // Retrieve an instance of HogWildContext from the service provider.
    var context = ServiceProvider.GetService<HogWildContext>();

    // Create a new instance of WorkingVersionsService,
    // passing the HogWildContext instance as a parameter.
    return new WorkingVersionsService(context);
});
```

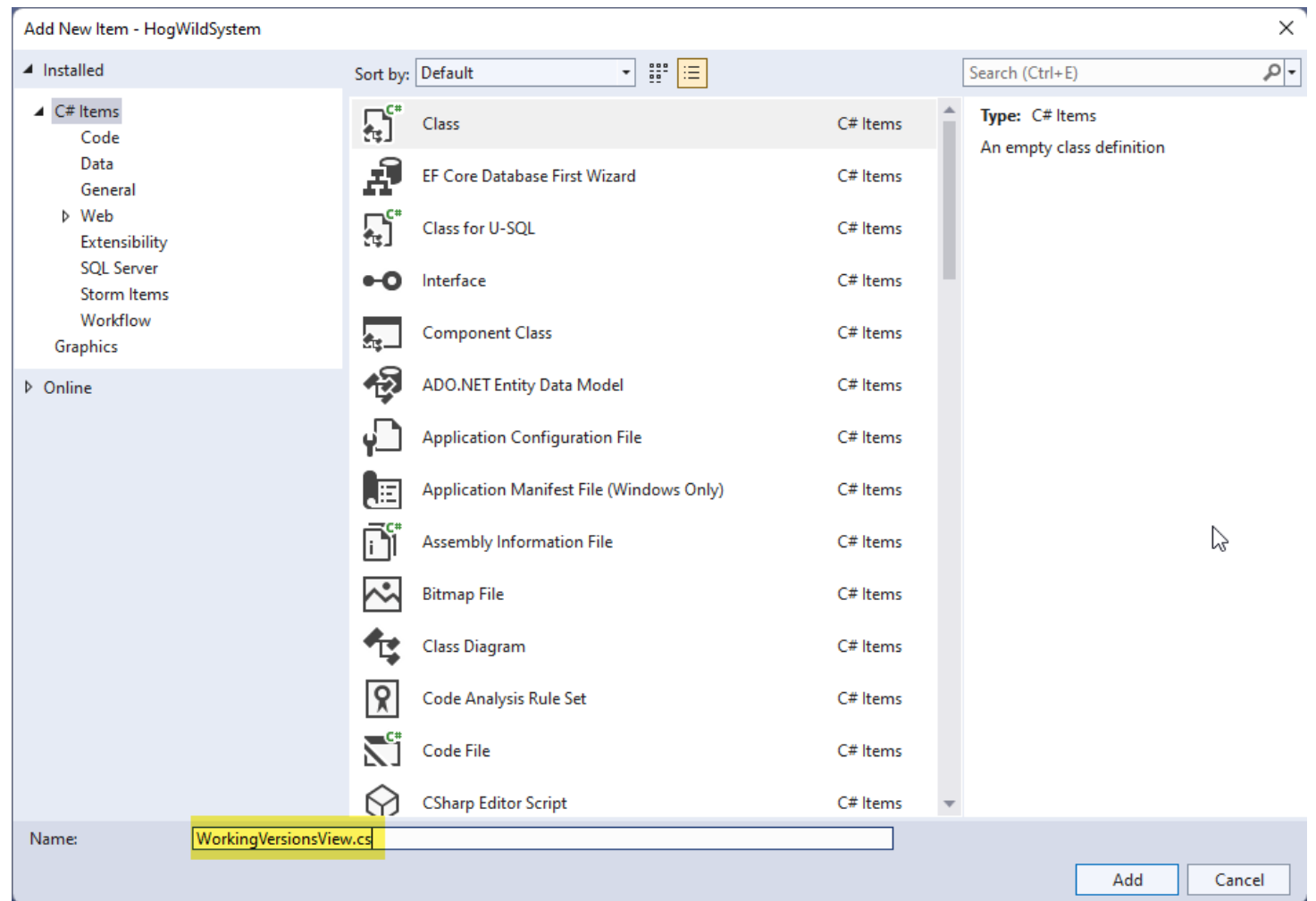
# Adding View Model

- We must create a view model for the “Working Versions” table by adding a new class call WorkingVersionsView



# Adding View Model

- Provide a name of “WorkingVersionsView.cs”



# Adding View Model

- Added the properties to reflect the data that we are capturing.

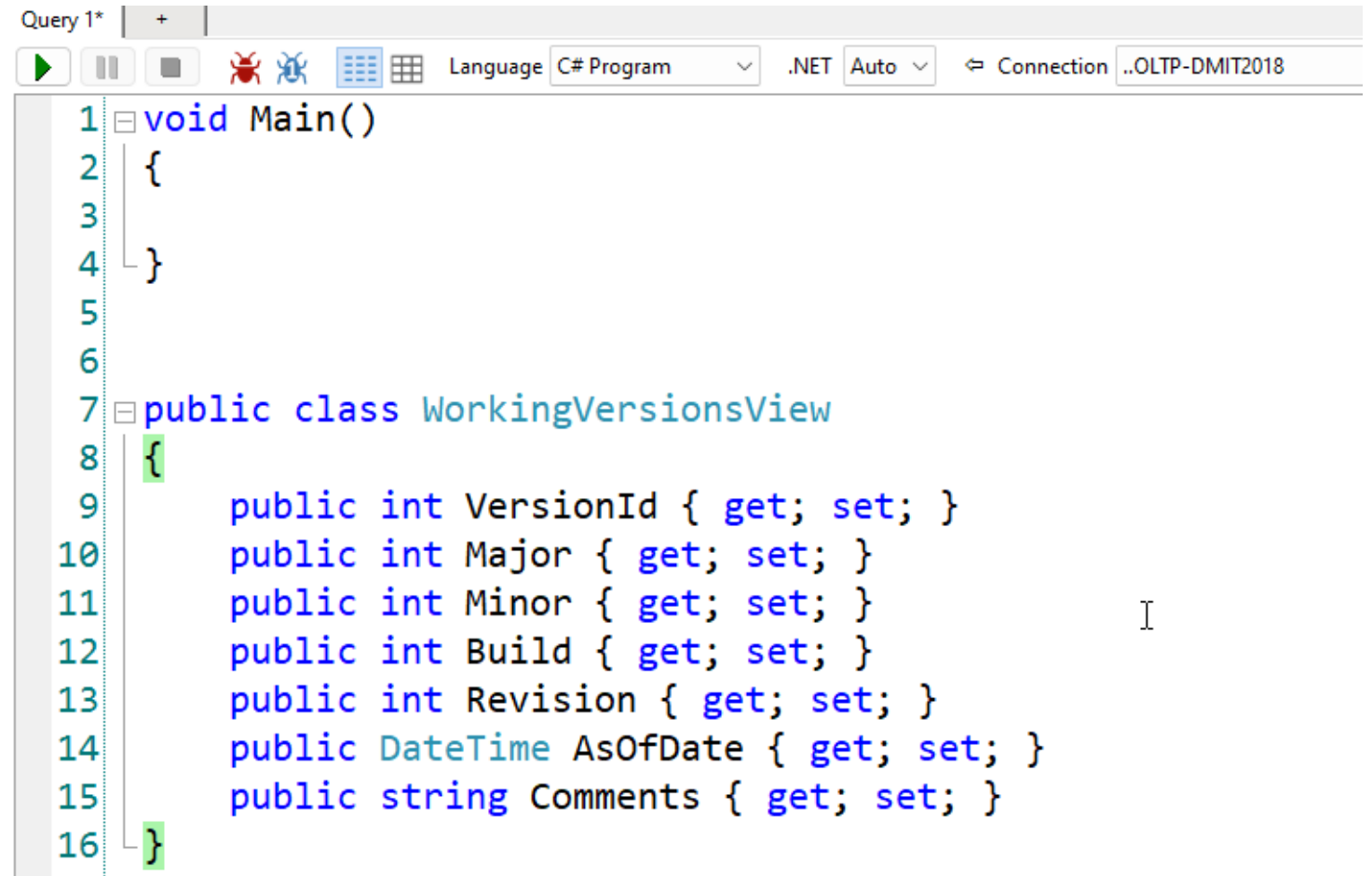
WorkingVersionViews.cs

```
namespace HogWildSystem.ViewModels
{
    0 references
    public class WorkingVersionsView
    {
        0 references
        public int VersionId { get; set; }
        0 references
        public int Major { get; set; }
        0 references
        public int Minor { get; set; }
        0 references
        public int Build { get; set; }
        0 references
        public int Revision { get; set; }
        0 references
        public DateTime AsOfDate { get; set; }
        0 references
        public string Comments { get; set; }
    }
}
```

# Creating GetWorkingVersion Method

---

- In LINQPad, we must create a “C# Program” and code up our method to return the last versioning information.
1. Add your WorkingVersionsView class to the LINQPad
  2. Set Connection to “OLTP-DMIT2018”



The screenshot shows the LINQPad interface. At the top, there's a toolbar with icons for running, debugging, and other functions. Below the toolbar, the 'Language' is set to 'C# Program' and the 'Connection' is set to '..OLTP-DMIT2018'. The main area displays the following C# code:

```
1 void Main()
2 {
3
4 }
5
6
7 public class WorkingVersionsView
8 {
9     public int VersionId { get; set; }
10    public int Major { get; set; }
11    public int Minor { get; set; }
12    public int Build { get; set; }
13    public int Revision { get; set; }
14    public DateTime AsOfDate { get; set; }
15    public string Comments { get; set; }
16 }
```

# Creating GetWorkingVersion Method

---

- Code your method

```
Language C# Program .NET Auto Connection ..OLTP-DMIT2018

void Main()
{
}

public WorkingVersionsView GetWorkingVersion()
{
    return WorkingVersions
        .Select(x => new WorkingVersionsView
        {
            VersionId = x.VersionId,
            Major = x.Major,
            Minor = x.Minor,
            Build = x.Build,
            Revision = x.Revision,
            AsOfDate = x.AsOfDate,
            Comments = x.Comments
        }).FirstOrDefault();
}

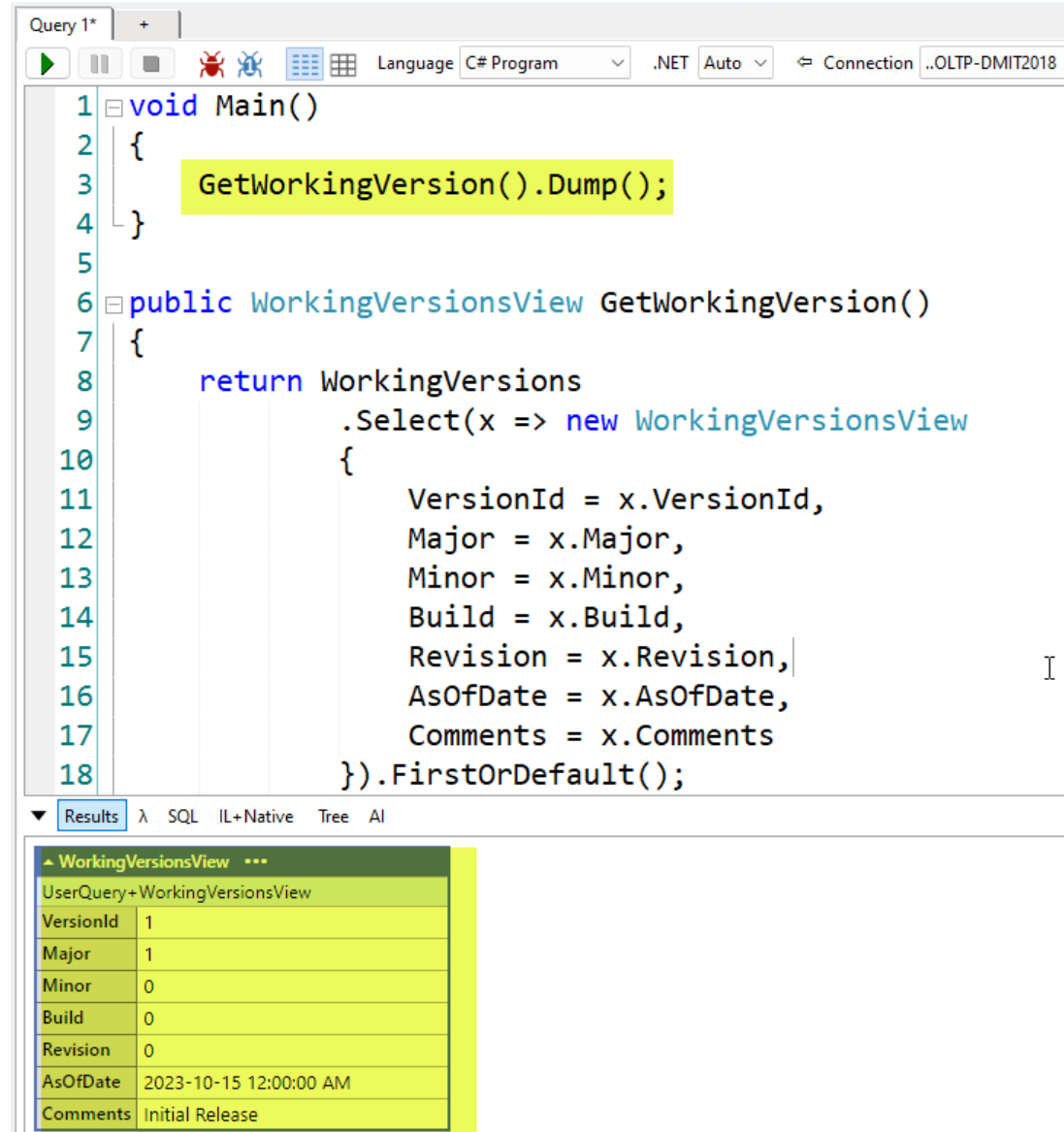
public class WorkingVersionsView
{
    public int VersionId { get; set; }
    public int Major { get; set; }
    public int Minor { get; set; }
    public int Build { get; set; }
    public int Revision { get; set; }
    public DateTime AsOfDate { get; set; }
    public string Comments { get; set; }
}
```



# Creating GetWorkingVersion Method

---

- Ensure that your method run correctly.



The screenshot shows a C# program in Visual Studio. The code defines a `void Main()` method that calls `GetWorkingVersion().Dump();`. The `GetWorkingVersion()` method is a `public WorkingVersionsView` that returns a `WorkingVersionsView` object. This object is created by selecting the first record from a query, mapping its properties (`VersionId`, `Major`, `Minor`, `Build`, `Revision`, `AsOfDate`, and `Comments`) to the corresponding properties of the `WorkingVersionsView` object.

```
1 void Main()
2 {
3     GetWorkingVersion().Dump();
4 }
5
6 public WorkingVersionsView GetWorkingVersion()
7 {
8     return WorkingVersions
9         .Select(x => new WorkingVersionsView
10             {
11                 VersionId = x.VersionId,
12                 Major = x.Major,
13                 Minor = x.Minor,
14                 Build = x.Build,
15                 Revision = x.Revision,
16                 AsOfDate = x.AsOfDate,
17                 Comments = x.Comments
18             }).FirstOrDefault();
```

The Results window at the bottom shows the output of the `Dump()` method, displaying the properties of the `WorkingVersionsView` object:

WorkingVersionsView ***	
UserQuery+WorkingVersionsView	
VersionId	1
Major	1
Minor	0
Build	0
Revision	0
AsOfDate	2023-10-15 12:00:00 AM
Comments	Initial Release

## Updating Working Versions Service with GetWorkingVersion()

- Copy the method into your WorkingVersionsService class.

## WorkingVersionsService.cs

```
internal WorkingVersionsService(HogWildContext hogWildContext)
{
    // Initialize the _hogWildContext field with the provided HogWildContext instance.
    _hogWildContext = hogWildContext;
}

// This method retrieves the working version of a resource.
0 references
public WorkingVersionsView GetWorkingVersion()
{
    return WorkingVersions
        .Select(x => new WorkingVersionsView
        {
            VersionId = x.VersionId,
            Major = x.Major,
            Minor = x.Minor,
            Build = x.Build,
            Revision = x.Revision,
            AsOfDate = x.AsOfDate,
            Comments = x.Comments
        }).FirstOrDefault();
}
```

# Handling Issues with Current Context

- This error typically occurs when trying to access a variable or class named 'WorkingVersions,' but the compiler can't find it in the current context.
- We need to add a reference to \_hogWildContext, which we define at the beginning of the file.
- Add “#nullable disable” to the top of the file to disable nullable warning message.

## WorkingVersionsService.cs

// This method retrieves the working version of a resource.

0 references

```
public WorkingVersionsView GetWorkingVersion()  
{  
    return WorkingVersions  
        .Select(x => new WorkingVersionsView  
        {  
            VersionId = x.VersionId,  

```

// This method retrieves the working version of a resource.

0 references

```
public WorkingVersionsView GetWorkingVersion()  
{  
    return _hogWildContext.WorkingVersions // DbSet<WorkingVersion>  
        .Select(x:WorkingVersion => new WorkingVersionsView  
        {  
            VersionId = x.VersionId,  
            Major = x.Major,  

```

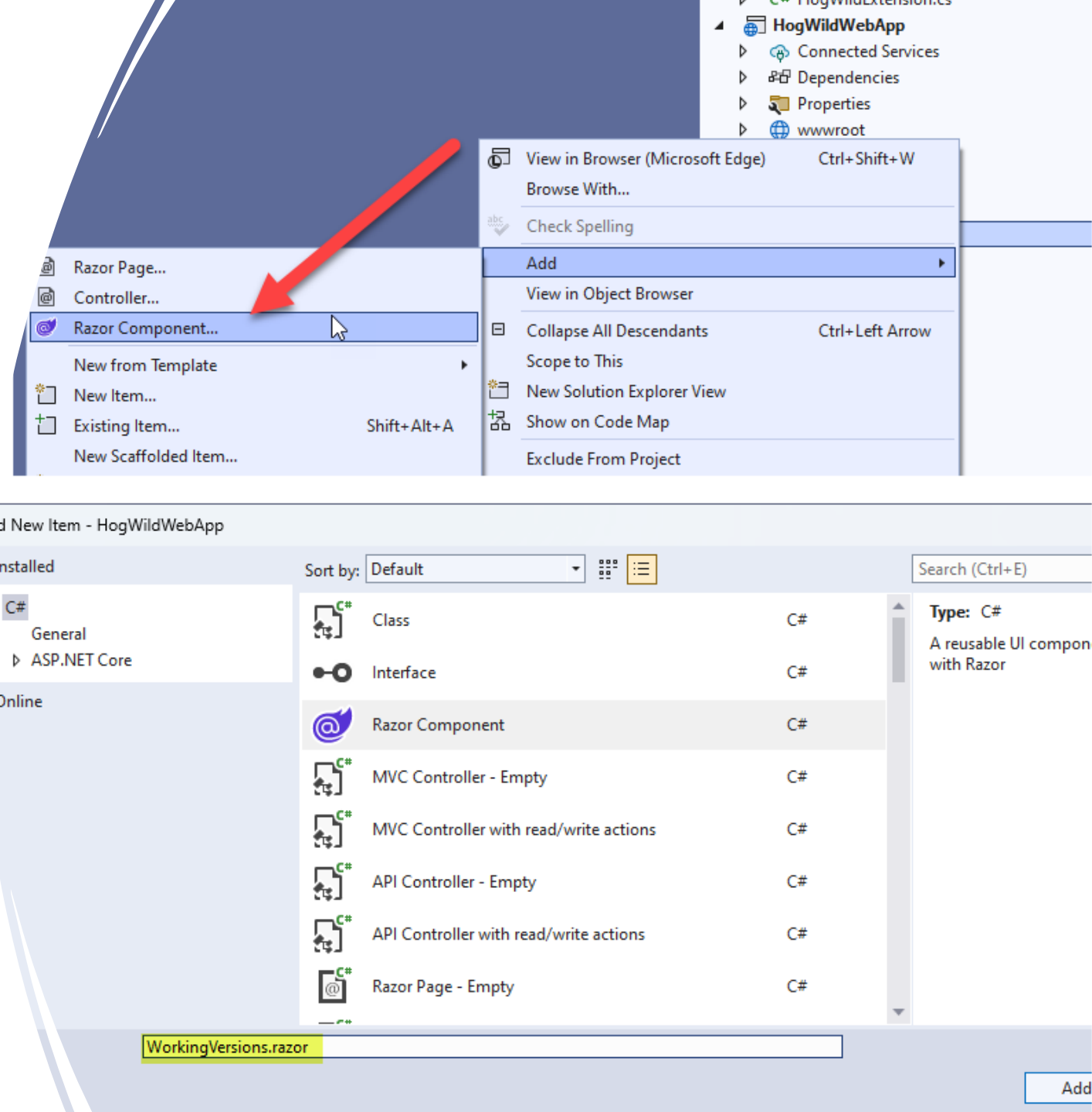
# Working Versions User Interface

1. Create a new Razor component page called "WorkingVersions."
2. Implement a navigation menu on the Razor Component Page.
3. Update the @Page reference.
4. In the code behind of the "WorkingVersions" page, add the necessary code to fetch data.
5. Refactor the Razor Component page to display the retrieved data.

# Add Razor Component

---

- Right-click on the sample page and select “Add -> Razor Component”
- Set the page name to “WorkingVersions.razor”



# Add Navigation Menu

NavMenu.razor

```
<MudNavMenu>
  <MudNavLink Href="/" Match="NavLinkMatch.All" Icon="@Icons.Material.Filled.Home">Home</MudNavLink>
  <MudNavLink Href="/counter" Match="NavLinkMatch.Prefix" Icon="@Icons.Material.Filled.Counter">Counter</MudNavLink>
  <MudNavLink Href="/weather" Match="NavLinkMatch.Prefix" Icon="@Icons.Material.Filled.Weather">Weather</MudNavLink>

  <MudNavGroup Title="Sample Pages">
    <MudNavLink Href="/SamplePages/Basics" Match="NavLinkMatch.Prefix" Icon="@Icons.Material.Filled.Abc">Basics</MudNavLink>
    <MudNavLink Href="/SamplePages/WorkingVersion" Match="NavLinkMatch.Prefix" Icon="@Icons.Material.Filled.Album">Working Version</MudNavLink>
  </MudNavGroup>

  <MudNavLink Href="/auth" Match="NavLinkMatch.Prefix" Icon="@Icons.Material.Filled.Login">Auth</MudNavLink>
  <AuthorizeView>
```

# Update the @Page Reference – WorkingVersions.razor

- Add the @page reference. Ensure that you include the “Sample Page” subfolder.
- Remove the “@code” coding area
- Update the name “WorkingVersions” -> “Working Versions”.
- Run the code to ensure that your page does load.

## WorkingVersion.razor

```
@page "/SamplePages/WorkingVersion"
```

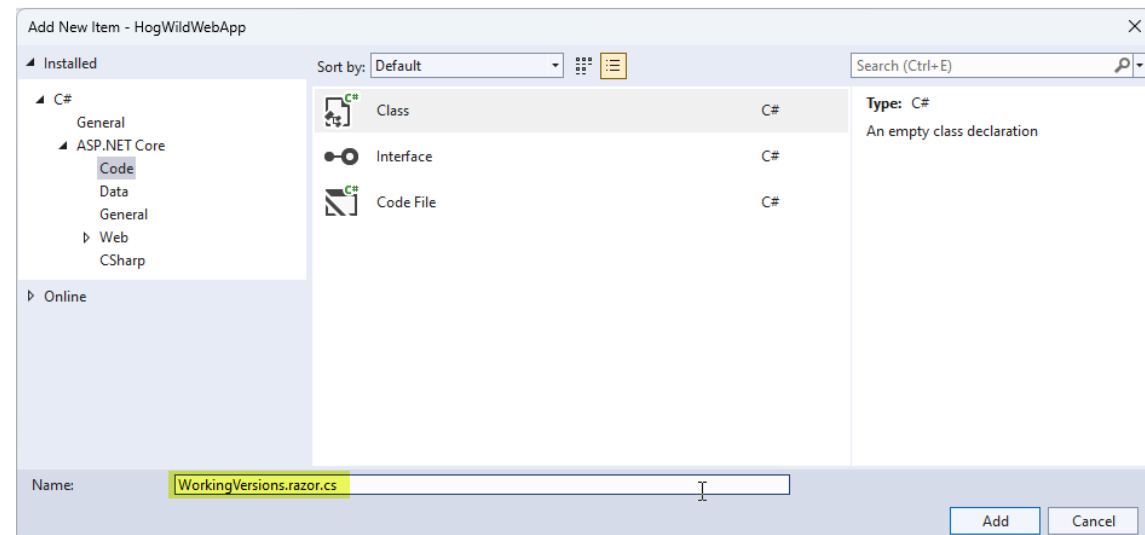
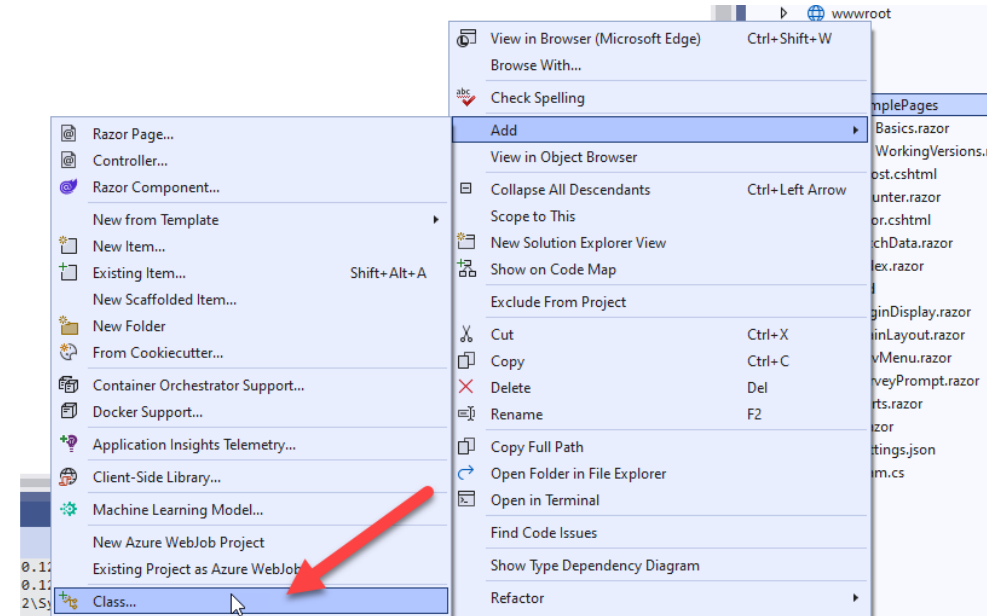
```
<PageTitle>Working Version</PageTitle>
```

```
<!-- Displays a heading using MudBlazor's MudText component with "h3" typography. -->  
<MudText Typo="Typo.h3">Current Working Version Info</MudText>
```



# Create the Code Behind File

- Make sure that your new file name is “WorkingVersions.razor.cs”



# Refactor the Code Behind

WorkingVersion.razor.cs

- Add “partial” to the class definition.
- Add “Working Versions” services
- - `[Inject]` is a special attribute in ASP.NET Core and Blazor that lets you request and use services (like databases or custom classes) within your web components.
- - When you apply `[Inject]` to a property, the framework automatically provides the requested service, making it available for your component. This promotes code organization, testability and reduces the need for components to create their own dependencies.

```
#nullable disable
using HogWildSystem.BLL;
using Microsoft.AspNetCore.Components;

namespace HogWildWebApp.Pages.SamplePages
{
    1 reference
    public partial class WorkingVersions
    {
        #region Properties

        // This attribute marks the property for dependency injection.
        [Inject]
        // This property provides access to the 'WorkingVersionsService' service.
        0 references
        protected WorkingVersionsService WorkingVersionsService { get; set; }
        #endregion
    }
}
```

# Add Working Versions View Model

---

- We must create a new WorkingVersionsView so that we do not get a null exception when the page loads.

## WorkingVersion.razor.cs

```
1 reference
public partial class WorkingVersions
{
    #region Fields
    // This private field holds a reference to the WorkingVersionsView instance.
    private WorkingVersionsView workingVersionsView = new WorkingVersionsView();
    #endregion

    #region Properties
    // This attribute marks the property for dependency injection.
    [Inject]
    // This property provides access to the 'WorkingVersionsService' service.
    1 reference
    protected WorkingVersionsService WorkingVersionsService { get; set; }
    #endregion
}
```

# Add Code to Retrieve Working Versions Data

---

1. Add feedback property to store any error messages.

## WorkingVersion.razor.cs

```
#region Fields
// Property for holding any feedback messages
private string feedback;
// This private field holds a reference to the WorkingVersionsView instance.
private WorkingVersionsView workingVersionsView = new WorkingVersionsView();
#endregion

#region Properties
// This attribute marks the property for dependency injection.
[Inject]
```

# Add Code to Retrieve Working Versions Data

---

1. Add code to retrieve a Working Versions View Model.
2. Add try/catch for any errors.

## WorkingVersion.razor.cs

```
#region Methods
1 reference | James Thompson, 90 days ago | 1 author, 1 change
private void GetWorkingVersion()
{
    try
    {
        workingVersionView = WorkingVersionService.GetWorkingVersion();
    }
    catch (Exception ex)
    {
        // capture any exception message for display
        feedback = ex.Message;
    }
}
#endregion
```

# Working Versions Page

---

## WorkingVersion.razor

- NOTE: If the “Version ID” equal to zero, we have not retrieved the record yet, we do not show any of the labels for the View Model.

```
@if(workingVersionView?.VersionID > 0)
{
    <MudText Typo="Typo.h5"><strong>Version ID:</strong> @workingVersionView.VersionID</MudText>
    <MudText Typo="Typo.h5"><strong>Major:</strong> @workingVersionView.Major</MudText>
    <MudText Typo="Typo.h5"><strong>Minor:</strong> @workingVersionView.Minor</MudText>
    <MudText Typo="Typo.h5"><strong>Build:</strong> @workingVersionView.Build</MudText>
    <MudText Typo="Typo.h5"><strong>Revision:</strong> @workingVersionView.Revision</MudText>
    <MudText Typo="Typo.h5"><strong>Date:</strong> @workingVersionView.AsOfDate</MudText>
    <MudText Typo="Typo.h5"><strong>Comments:</strong> @workingVersionView.Comments</MudText>
}
```

# Working Versions Page

- Add the button to retrieve the working version view.

## WorkingVersion.razor

```
<MudButton OnClick="GetWorkingVersion"|
           Variant="Variant.Filled"
           Color="Color.Primary">
    Get Working Version
</MudButton>
@if(!string.IsNullOrEmpty(feedback))
{
    <div class="mt-4">
        <MudAlert Severity="Severity.Error">
            <MudText Typo="Typo.h4">@feedback</MudText>
        </MudAlert>
    </div>
}
```



# Blazor Page Output

---

## Current Working Version Info

GET WORKING VERSION

Before Button  
Push

## Current Working Version Info

Version ID: 1

Major: 1

Minor: 2

Build: 12

Revision: 5

Date: 10/15/2023 12:00:00 AM

Comments: Initial Release

GET WORKING VERSION

After Button  
Push

## Statement Regarding Slide Accuracy and Potential Revisions

- Please note that the content of these PowerPoint slides is accurate to the best of my knowledge at the time of presentation. However, as new research and developments emerge, or to enhance the learning experience, these slides may be subject to updates or revisions in the future. I encourage you to stay engaged with the course materials and any announcements for the most current information