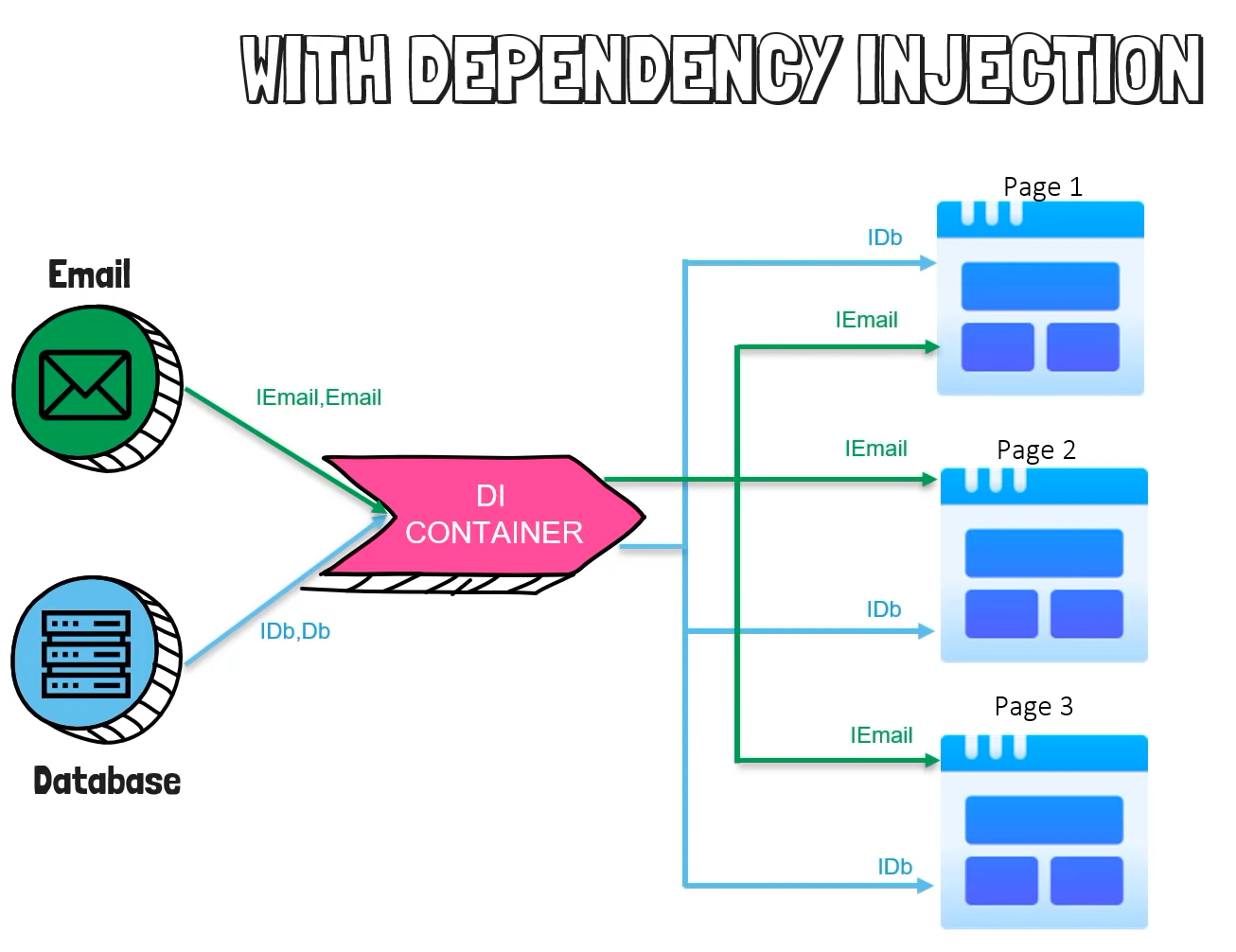


Without Dependency Injection: Let say we have 3 pages and in all 3 pages we’ve to use send the email and access database. So in this case what generally we were doing was initiating an object of both the classes Email & Database.

Disadvantages:

* So create the object open the connection and close the connection in all three pages.
* Which is basically the duplication of the code.
* In case if you made some changes in the main class i.e email and database then you have to reconfigure the use in all the 3 pages.



interface IEmail

{

void OpenConnection();

}

class Email : IEmail

{

public void OpenConnection ()

{

Database db = new Database();

return db;

}

}

class Page1

{

Email E = new Email();

Var Connection = E.OpenConnection();

Connection.NewDb();  
}

With Dependency Injection: We have a Dependency Injection Container inside which we’ll be having the implementation of the Email and Database interfaces. When any of the pages will require the functionality it’ll just ask the DI container to create an object of this functionality. So inside the page we will be using an interface, and then DI passes the object when the website needs it.

**Default Project Structure**

**Properties**:

launchSettings.json: It contains a different profiles needed to run the project such as iisSettings, profiles.

**wwwroot**: Containes the static files of a project, the project may contain css, js, lib (images), favicon.ico. So all these static files are included under wwwroot folder.

**appsettings.json**: Connection strings and keys will be added inside this json file.

**Program.cs**: The main file responsible for running the application.

**Controllers**: Contains business logic of the project pages i.e .cs file.

**Views**: Contains the web code of the pages.

**Model**: Contains the property/attributes for storing & accessing data.

**Program.cs:**

**Builder:**

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews(); //This is the service added to container

var app = builder.Build();

builder is the DI Container. When we want to register anything with our dependency injection, let say if we want to register our database or email that we have to do between the *builder* and *app* variables.

**Pipeline**:

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

Diagram

Description automatically generated

1. The pipeline specifies how application should respond to a web request, so when your application receives a request from the browser that request goes back and forth through the pipeline.
2. In pipelines you can add items that you want, it’s made up of different middlewares, and MVC is a type of middleware itself.
3. So if we want an application to be built using MVC, we have to add that middleware.
4. Other example could be Authentication, Static files, etc.
5. The order of the middlewares is important.

**MVC Architecture:**

Diagram

Description automatically generated

**Model**: It represents all the data in your application, if there’s product page, product database then the product data/properties is a model. So if you have 10 tables in your database then we’ll be having 10 models that correspons to them.

**View**: User Interface create in web languages such as html, css, js, etc.

**Controller**: It acts as an interface between the model and the view to process all the business logic and incoming request. It manipulates the data using model and interacts with the view to render the final output.

*So if there’s any request for the information from view it’ll take the data from the model and gives back to the view.*

**Routing:**

***URL Pattern:***

[https://localhost:54321/{controller}/{action}/{id}](https://localhost:54321/%7bcontroller%7d/%7baction%7d/%7bid%7d)

Ex: <https://localhost:54321/Product/Details/1>

Table

Description automatically generated

**Controller:**

public class ProductController : Controller

{

public IActionResult Details()

{

return View();

}

}

Product is the controller name and Details is the view name which is returning it’s view present in View Folder.

[https://localhost:54321/Product/Details](https://localhost:54321/Product/Details/1)

*Example 2:*

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

public IActionResult Privacy()

{

return View();

}

}

Graphical user interface, text, application

Description automatically generated

To visit Index: <https://localhost:54321/Home/Index>

To visit Privacy: <https://localhost:54321/Home/Privacy>

**Views:**

**Shared:** A view within a view that can be used in any other views.

Let say you have created a default color code for your website which makes a page background color as blue now whenever you want to have a background color as blue for your new page you can simply import this default layout of the shared folder.

*\_Layout*: A partial view which is a default master page of your application.

\_ValidationScriptsPartial: Contains a set of scripts for validations.

Error: Contains a set of error to display different errors.

*Example:*

*Index.cshtml*

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

***Output:***

Graphical user interface, application

Description automatically generated

*Where’s that header and footer came from apart from the text?*

* *From Shared \_Layout:*

Shared > \_Layout.cshtml

Text

Description automatically generated

Views > **\_ViewImports.cshtml**: Let say you have to include some namespaces in all of your project classes, then at here you can mention those namespace to be included.

@using BulkyBookWeb

@using BulkyBookWeb.Models

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

Views > **\_ViewStart.cshtml:** It specifies the default master page for the application which is a \_Layout.cshtml.

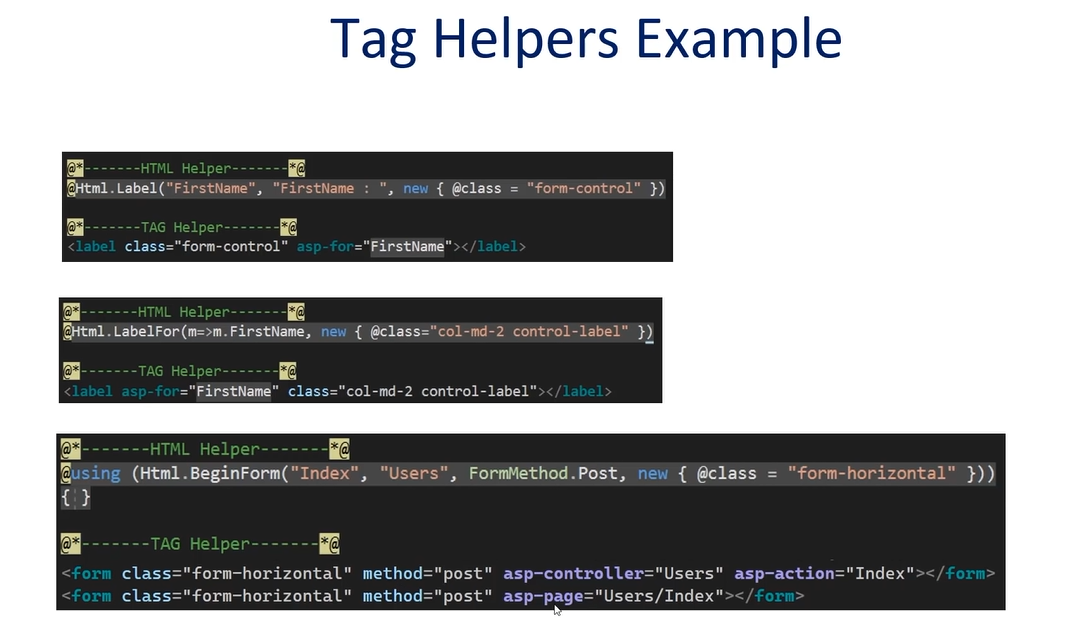
@{

Layout = "\_Layout";

}

**Tag Helpers:**

* Tag Helpers enable server side code to participate in creating and rendering html elements in Razor files.
* Tag helpers are very focused around the html elements and much more natural to use.



**ActionResult:** IActionResult is a generic type that implements all of the other return types.

Ex:

So Instead of writing return type explicitly such as:

public ViewResult Index()

{

return View();

}

public PageResult Index()

{

return Page();

}

We can write a generic return type which is IActionResult

public IActionResult Index()

{

return View();

OR

return Page();

}

**Starting With The Project:**

* Create a new class called Category.cs inside model folder

using System.ComponentModel.DataAnnotations;

namespace BulkyBookWeb.Models

{

public class Category

{

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

public int DisplayOrder { get; set; }

public DateTime CreatedDateTime { get; set; } = DateTime.Now;

}

}

**Data Annotations:** [Key] & [Required] are the data annotations which specifies the constraints for the attributes. Here [Key] annotation states that the Id attribute is a primary key and [Required] states that the attribute is mandatory to fill and it can’t be null.

// type prop and double tap the ‘tab’ button to create the property.

We can set the default value for the property such as:

public DateTime CreatedDateTime { get; set; } = DateTime.Now;

**Database Connection:**

* **ConnectionStrings:**

To connect a database with our application, we have to connect to a local database through MSSMS (SQL Server Management) and then have to specify a connection string in our appsettings.json file to establish the connection.

"ConnectionStrings": {

"DefaultConnection": "Server:ILPT4055; Database=Bulky;Trusted\_Connection:True;"

}

Apart from this we have to use the DB Context of Entity Framework for the connection of the database.

* **DbContext:**

Create the Data folder inside it create a class as *ApplicationDbContext*

The general configuration/syntax of DbContext:

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)

{

}

}

In the contructor parameter we will pass some options and those option we just have to pass to the base class which is db context.

* **DbSet:**

Then, we have to create a DB Set inside the DB Context class:

Syntax: public DbSet<ModelName> DatabaseTableName { get; set; }

Ex: public DbSet<Category> Categories { get; set; }

Final Steps For Configuration:

1. We need to tell our application that it has to use the DbContext which is inside the ApplicationDbContext.
2. Then it has to use sql server using the connection string that we defined inside the appsettings.json file.
3. We have to add the DbContext service inside the program.cs class.
4. After that we have to do migrations using entity framework to push the changes to database.

***Program.cs***

builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(

builder.Configuration.GetConnectionString("DefaultConnection")

));

**Migration**: It keeps a track of all DB Changes that are needed, once that migration is done you push that migration to the database to actually create the database or make changes to your table.

1. Go to Tools > nuget package manager > Package Manager Console.
2. Run the command: *add-migration AddCategoryToDatabase*
3. After the migration is done, we have to create the table, for that run the command: *update-database.* It’ll create the table inside the database.

**MVC Setups:**

***Category > Controller:***

public class CategoryController : Controller

{

private readonly ApplicationDbContext \_db;

public CategoryController(ApplicationDbContext db)

{

\_db = db;

}

public IActionResult Index()

{

IEnumerable<Category> objCategoryList = \_db.Categories;

return View(objCategoryList);

}

}

***View > Category > Index.cshtml:***

@model IEnumerable<Category>

<table class="table table-bordered table-striped" style="width:100%">

<thead>

<tr>

<th>

Category Name

</th>

<th>

Display Order

</th>

</tr>

</thead>

<tbody>

@foreach(var obj in Model)

{

<tr>

<td width="50%">

@obj.Name

</td>

<td width="30%">

@obj.DisplayOrder

</td>

</tr>

}

</tbody>

</table>