Ch10 – Dependency Injections

Dependancy Injections is a design pattern found everywhere. The idea is simple; Instead of creating a handler/model inside my function to use once and destroy, I feed this object to my function as a parameter (or feed it to the class itself through the constructor).

Ex:

Before injection, the EmailSender is cotnantly created, used once, and destroyed evey time I get to the page which is inefficient

```
Public class myController : Controller{

    Public IActionResult doSomething(…, …, )
       …
       …
       EmailSender em = new EmailSender();
       em.doStuff();
    return View(…);
}
```

Injection through constructor:

```
Public class myController : Controller{

    Public readonly EmailSender _em;

    Public myController(EmailSender em){
        This._em = em;
    }

    Public IActionResult doSomething(…, …, )
       …
       …
       _em.doStuff();
    return View(…);
}
```

Further loosing up the code: (loosely connected code means any alterations and additions will be easy to manage)

Replace EmailSender with an interface IEmailSender that way we can give our class models that extend this interface further loosening up the code

Ex: MockEmailSender, HTMLEmailSender, BatchEmailSender all extending IEmailSender. In our main constructor we can create any of these 3 and send it to myController and it will do it's job. We can also later create a fourth sender and use that as well.

Injections made by ASP itself: ASP Core has built in injection providers. For this course we will use them for two components, our identity provider (See chapter 14) and database handler (see chapter 12)

Have all controllers extend from this main one:

```
namespace ProjectName.Controllers
{
    public class MainController : Controller
    {
        protected readonly ApplicationDbContext _context;
        protected readonly UserManager<IdentityUser> _userManager;
        public MainController(ApplicationDbContext context, UserManager<IdentityUser> userManager)
        {
            _context = context;
            _userManager = userManager;
        }
    }
}
```

Ex:

```
namespace LebHealth.Controllers
{
    public class AdminController : MainController
    {
        public AdminController(ApplicationDbContext context, UserManager<IdentityUser> userManager)
        : base(context, userManager) //These can now be used alll over the page
        {}
    }
}
```

Ch12 – Entity Framework

EF Core is a library that provides object-oriented access to databases. It acts as the mediator between what the user sees, models and classes, and what is stored in the database.

An entity is a .NET class to be mapped into a database using EF Core.

Adding EF to a project:

- Tools -> NuGet Package Manager -> Manage NuGet Packages For Solution
- Search "Entity Framework"
- Select the first result
- Click install on the right

Note: Do not use the NuGet console unless you know what you're doing. The console will install only the latest version which might cause erros later. Using the GUI will automatically add the best suited version for your project.

Using EF:

- Create the models you want
- In Data-> ApplicationDBContext.cs :

```
namespace Something.Data{
    public class ApplicationDbContext : IdentityDbContext{

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options){  }
        public ApplicationDbContext() {  }
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder){
            if (!optionsBuilder.IsConfigured){
              optionsBuilder.UseSqlServer("COPY_THIS_FROM_DefaultConnection_IN_appsettings.json");
            }
        }      //Add your models here:
        public DbSet<MyFirstModel> MyFirstModel { get; set; }
        public DbSet<MySecondModel> MySecondModel { get; set; }
        public DbSet<MyThirdModel> MyThirdModel { get; set; }
    }
}
```

- Tools -> NuGet Package Manager -> Project Manager Console:
  - Cd "C:/path/to/your/project/file"
  - `dotnet ef migrations add MyMigrationName`
  - **dotnet ef database update**
- The command is pretty smart and will atuomatically handle foreign keys, it will also set attributes with "ID" in them as primary keys though there might be some errors yet they will be easy to fix
- Repeat every time you update the models
- Note: if the second command fails then one of your changes can not be done over a previous migration. The best thing to do here is to manually delete the migrations you've made so far in the Migrations folder (do not delete any ones you didn't create youself though), clear the data you added in the ApplicaitonDBContext file, run the first command, delete the new migraiton, add the data back into ApplicationDBContext, and run the two commands. (You might have to manually delete the tables you already have if any)
  - I know this is annoying but so is Microsoft

Example of class used to insert data:

```
namespace Something.Models{
  public class DBHandler{
    readonly ApplicationDbContext context;
    public DBHandler(ApplicationDbContext _context){
      context = _context;
    }
    public void add_myThing(myThing mything){
      context.myThing.Add(mything);
      context.SaveChanges();
    }
  }
}
```

Controller:

```
new DBHandler(new Data.ApplicationDbContext()).add_message(m);
```

Note: for auto-incrementing IDs, use this tag on your field:
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

Function in same class to read data:

```
public List<myThing> get_myThing(){
    return context.myThing.ToList();
}
```

Controller:

```
public IActionResult Index(){
    List<Message> data = new DBHandler(new
Data.ApplicationDbContext()).get_Messages();
    return View("Index", data);
}
```

View:

```
@model myThing;

<h1> @Model.something </h1>
```

Can also be a collecion/list:

```
@model List<myThing>;
<ul>
  @foreach (myThing m in Model){
    <li>
      @m.something
    </li>
  }
</ul>
```

Ch14 – Identity Provider

ASP has the ability to generate the back and front end services for user accounts automatically when you create your project. Simply create your desired **ASP Web Application** project and, when seleting the type (preferrably MVC Application), select **Authentication** and select **Individual User Accounts**.

This will add a Data file with the needed migrations for tables that will hold the user's information and passwords. These are to be migrated into the default database locaation which you can alter in appsettings.json. By defualt, it looks like:

```
"ConnectionStrings": {
"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-APPNAME-69BF51E7-E15C-47D8-AE46-2C25F3731A7B; Trusted_Connection=True; MultipleActiveResultSets=true"
},
```

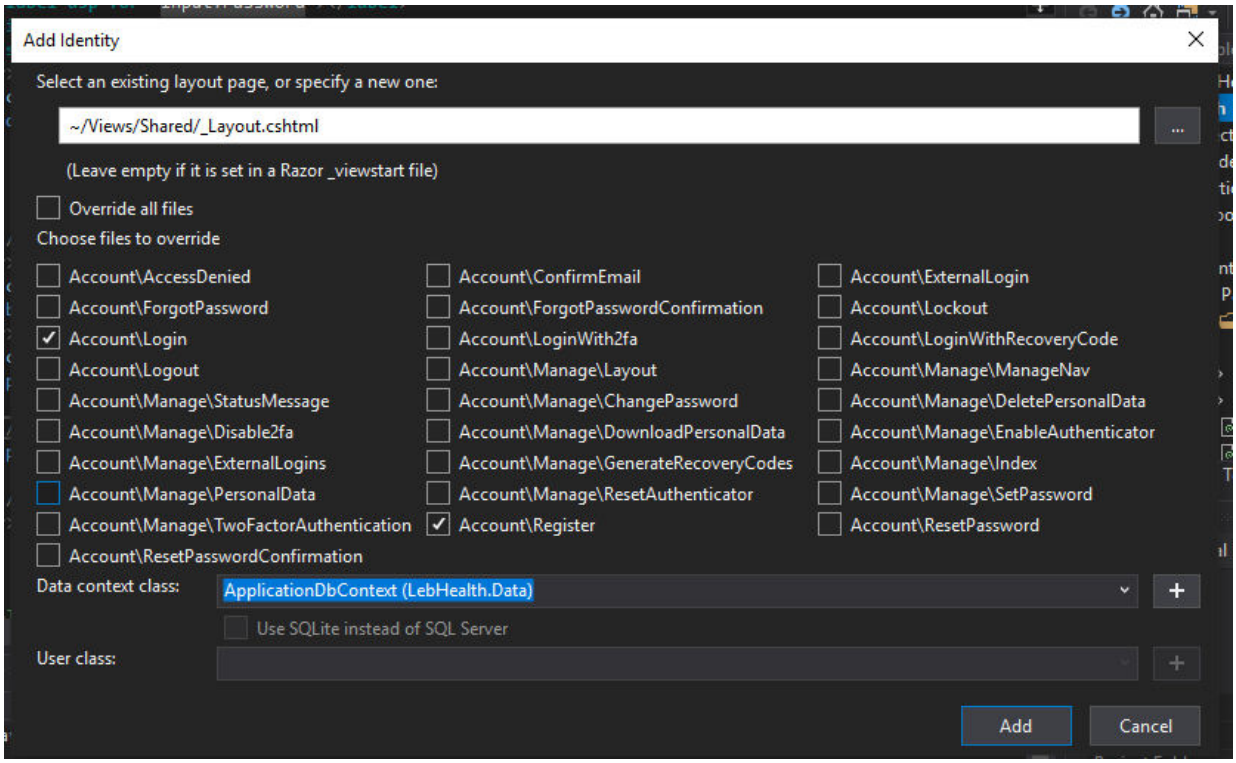
You can keep it like so or change it to your desired database. Afterwards to migrate it go into **Packet Managerment Console** and run the following command: **`Update-Database`**

This will migrate the tables and initiate the database.

The process also adds up to 30 different views connected to this database for logging in, registering new accounts, forgot password, adding two-way authentication, and other views you may need for user accounts. These are by default stored in the program. To edit them, you need to **scaffold** the specific pages you need (or all of them, but that will be hard to manage as there are 30). To scaffold them:

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **ADD**
- In the ADD Identity dialog, select the views you want
- Set the layout as your default layout and set the data context class from the dropdown menu (it was automatically created by the identity provider alongside the migration)
- Select **ADD**

In this example, I only want the Login and Register pages. I can still use the other pages but I don't want to make changes to them so I don't have to scaffold them.



An alternative would be to make your own new pages completely and use the generated syntax to access the created database just like Entity Framework

Extension: User roles and claims

Inside Startup.cs, add this to **ConfigureServices**

```
    services.AddIdentity<IdentityUser, IdentityRole>(options =>
    {
        options.Password.RequiredLength = 3;
        options.Password.RequireLowercase = false;
        options.Password.RequireUppercase = false;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireDigit = false;
    })
    .AddDefaultUI(UIFramework.Bootstrap4)
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
    services.AddAuthorization(options =>
    {
        options.AddPolicy("Admin", policy => policy.RequireRole("Admin"));
        options.AddPolicy("SomeRole", policy => policy.RequireRole("SomeRole"));
        options.AddPolicy("AnotherRole", policy => policy.RequireRole("AnotherRole"));
    });
```

Adding users with roles:

```
    public async Task<IActionResult> DocCreate(AdminAddUserDataModel incomingmodel)
    {
        var user = new IdentityUser { UserName = "username" Email = "email" };
        var result = await _userManager.CreateAsync(user, "password");
        IdentityUser X = await _userManager.FindByEmailAsync("email");
        await _userManager.AddClaimAsync(X, new Claim(ClaimTypes.Role, "SomeRole"));
        return RedirectToAction("SomeAction");
    }
```

Getting users of a specific role:

```
            foreach (IdentityUser user in _userManager.Users.ToList())
            {
                IList<Claim> claims = await _userManager.GetClaimsAsync(user);
                if(claims.First().Value == "Doctor")
                    rems.Add(user);
```