INTEGRATED SYSTEMS CHECKPOINTS:

REGISTER/LOGIN USER

1. Create Solution

2. Create a ApplicationUser class that will inherit from IdentityUser

```csharp
public class ApplicationUser : IdentityUser
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string UserAddress { get; set; }
}
```

3. Replace IdentityUser with ApplicationUser in Program.cs, AppdbContext and in LoginPartial

Program.cs:
```csharp
builder.Services.AddDefaultIdentity<ApplicationUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

AppdbContext.cs
```csharp
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
```

LoginPartial.cshtml (Here you must provide full path to the ApplicationUser like below)
```csharp
@inject SignInManager<ConcertShop.Models.Identity.ApplicationUser>
SignInManager
@inject UserManager<ConcertShop.Models.Identity.ApplicationUser>
UserManager
```

4. Add Migration & Update Database Through *Nuget Packet Manager Console*

add-migration	update-database

5. Create Scaffolding For Identity (On Scaffolding Select Login & Register)

6. On Register.cshtml.cs Add Input Models For Your Properties

```csharp
public class InputModel
{
    [Required]
    [Display(Name = "FirstName")]
    public string FirstName { get; set; }

    [Required]
    [Display(Name = "LastName")]
    public string LastName { get; set; }

    [Required]
    [Display(Name = "UserAddress")]
    public string UserAddress { get; set; }

    . . . Here are other Identity Default Input Models
```

7. On same file below register your properties when a user is created

```csharp
if (ModelState.IsValid)
{
    var user = CreateUser();

    user.FirstName = Input.FirstName;
    user.LastName = Input.LastName;
    user.UserAddress = Input.UserAddress;

    . . . Other Identity Defauls Stuff
```

8. On Register.cshtml Update The Layout of the Register Page
(You now have User Registration and User Login)

Products with All CRUD operations (Add, Edit, Delete)

1. Create a class for your product

public class **ConcertPlay**
{
    public Guid Id { get; set; }
    public string ConcertName { get; set; }
    public string ConcertPlace { get; set; }
    public int ConcertPrice { get; set; }
    public string ConcertDate { get; set; }
}
	public class **Ticket**
	{
	    public Guid Id { get; set; }
	    public string NumberOfPeople { get; set; }
	
	    public ApplicationUser? User { get; set; }
	    public string UserId { get; set; }
	
	    public ConcertPlay? ConcertPlay { get; set; }
	    public Guid ConcertPlayId { get; set; }
	}

ConcertPlay Class: Defines properties for a concert, including its unique identifier (Id), name, place, price, and date.

Ticket Class:
- Id: Unique identifier for the ticket.
- NumberOfPeople: Indicates the number of people covered by the ticket.
- User: Represents the user who owns the ticket. The ? indicates that this property is nullable, meaning it may be null.
- UserId: Holds the foreign key for the associated user.
- ConcertPlay: Represents the concert associated with the ticket. Again, the ? indicates it's nullable.
- ConcertPlayId: Holds the foreign key for the associated concert.

GUID (Globally Unique Identifier): A GUID is a 128-bit integer used as a unique identifier in software development. It ensures uniqueness across space and time, making it suitable for generating primary keys in databases.

Nullable Types (**?**): The ? indicates that the property can be assigned a null value. In this case, User and ConcertPlay properties can be null, meaning a ticket may not necessarily be associated with a user or a concert.

Foreign Keys: UserId and ConcertPlayId are foreign key properties used to establish relationships between Ticket and ApplicationUser (representing users) and ConcertPlay (representing concerts), respectively. They reference the primary keys (Id) of the associated entities.

2. Add The New Classes to ApplicationDbContext

```
namespace ConcertShop.Data
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
            : base(options)
        {
        }
        public DbSet<ConcertShop.Models.Domain.ConcertPlay> ConcertPlay { get;
set; }
        public DbSet<ConcertShop.Models.Domain.Ticket> Ticket { get; set; }

    }

}
```

In essence, these properties provide access to the corresponding database tables in the Entity Framework Core database context, allowing CRUD (Create, Read, Update, Delete) operations on these entities.
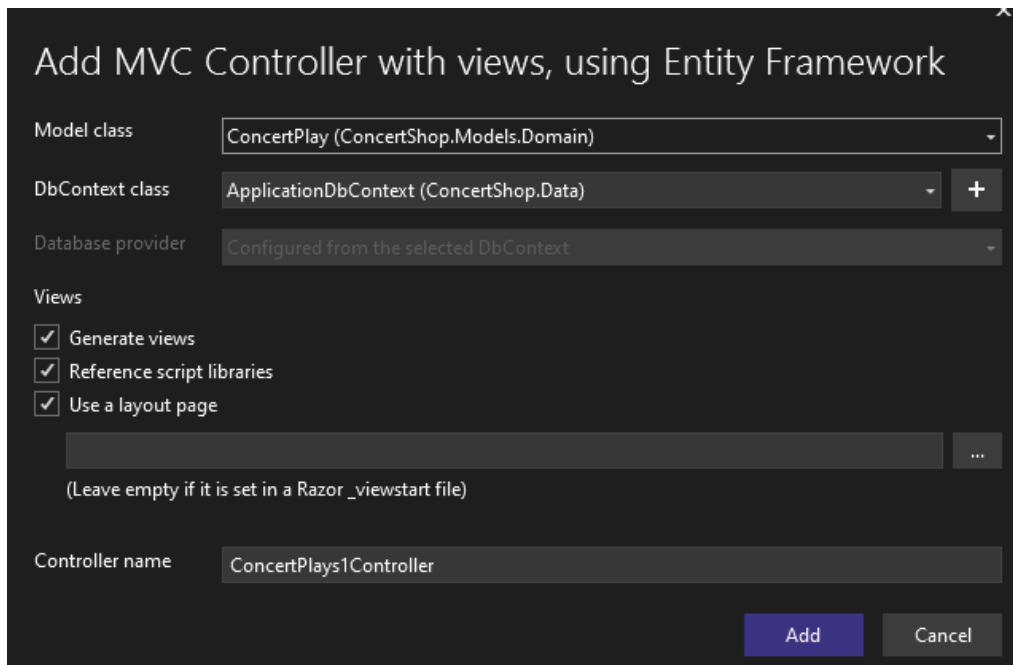
-Now Add Migration & Update Database *(Do this after every change you do to the classes like create class/edit/change properties etc...)*

Tip: If migration fails at any time and you are having difficulties fixing, find the Migrations folder and delete all migrations inside and try again, if it still fails than manually delete all the tables from SQL Server
(VisualStudio VIew Tab-> SQL Server Object Explorer)
This makes sure you are creating fresh tables, but if it still fails than the problem is on the code 🙁

3. Create Controller For The New Classes We Made

With this scaffolding we get ready made *Views* & *Controllers* for our classes. We now add Buttons on our Main Page so that we get to these new views.

4. Add Buttons To Our Main Page

Views/Shared/Layouts -

```
<li class="nav-item">
   <a class="nav-link text-dark" asp-area="" asp-controller="ConcertPlays"
asp-action="Index">Concerts</a>
</li>

<li class="nav-item">
   <a class="nav-link text-dark" asp-area="" asp-controller="Tickets"
asp-action="Index">Tickets</a>
</li>
```

Result Should Be These Two New Buttons(Concerts, Tickets)

All CRUD operations should work fine now.

Bonus Tip: If instead of seeing ID's like 8440ca03-17b1-4170-8455-7e11c579b49 you want to see the ConcertName or FirstName of User than you need to change it on Controller of the class you want it affected.

For Example, on TicketController on the GET: Tickets/Create Section We Have this:

```
// GET: Tickets/Create
public IActionResult Create()
{
    ViewData["ConcertPlayId"] = new SelectList(_context.ConcertPlay, "Id",
"ConcertName");
    ViewData["UserId"] = new SelectList(_context.Users, "Id", "FirstName");
    return View();
}
```

By default these green bolded lines are "Id" meaning they display those random Id numbers, but we can change to display ConcertName and FirstName, getting this when we want to create ticket:

NumberOfPeople

UserId

Berat

ConcertPlayId

Playboi Carti

Create

Back to List