Software Development

Presented By:
Khaled Gamal

ASP.NET Core

MVC

Section 6

ASP.NET Core MVC with EF Core - tutorial

Create the database context

- The main class that coordinates EF functionality for a given data model is the *DbContext* database context class.
- This class is created by deriving from the *Microsoft.EntityFrameworkCore.DbContext* class.
- The DbContext derived class specifies which entities are included in the data model.
- In this project, the class is named *SchoolContext*.

Create the database context

- In the project folder, create a folder named *Data*.
- In the Data folder create a *SchoolContext* class with the following code:

C# (SchoolContext.cs)

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;
namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options){ }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

Create the database context

- The preceding code creates a DbSet property for each entity set.
- In EF terminology:
  - An entity set typically corresponds to a database table.
  - An entity corresponds to a row in the table.
- The *DbSet<Enrollment>* and *DbSet<Course>* statements could be omitted and it would work the same. EF would include them implicitly because:
  - The Student entity references the Enrollment entity.
  - The Enrollment entity references the Course entity.

Create the database context

- When the database is created, EF creates tables that have names the same as the *DbSet* property names.
- Property names for collections are typically plural. For example, *Students* rather than *Student*.

Register the *SchoolContext*

- ASP.NET Core includes *dependency injection*.
- Services, such as the EF database context, are registered with dependency injection during app startup.
- Components that require these services, such as MVC controllers, are provided these services via constructor parameters.
- To register *SchoolContext* as a service, open Program.cs, and add the highlighted lines before calling *builder.Build()* method.

C# (Program.cs)

```
builder.Services.AddDbContext<SchoolContext>(options
=>options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")) );
```

Register the *SchoolContext*

- The name of the connection string is passed in to the context by calling a method on a *DbContextOptionsBuilder* object.
- For local development, the ASP.NET Core configuration system reads the connection string from the appsettings.json file.
- Open the *appsettings.json* file and add a connection string as shown in the following markup:

JSON (*appsettings.json*)

```
"ConnectionStrings": {
"DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;
MultipleActiveResultSets=true"
},
```

SQL Server Express LocalDB

- The connection string specifies *SQL Server LocalDB*.
- LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use.
- LocalDB starts on demand and runs in user mode, so there's no complex configuration.
- By default, LocalDB creates *.mdf* DB files in the *C:/Users/<user>/AppData\Local\Microsoft\Microsoft SQL Server Local DB\Instances\MSSQLLocalDB* directory.

Initialize DB with test data

- EF creates an empty database.
- The *EnsureCreated* method is used to automatically create the database.
- In the following code, a method is added that's called after the database is created in order to populate it with test data.
- In the *Data* folder, create a new class named *DbInitializer* with the following code:

Initialize DB with test data

C# (*DbInitializer.cs*)

```
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
public static class DbInitializer
{
public static void Initialize(SchoolContext context)
{
context.Database.EnsureCreated();
// Look for any students.
if (context.Students.Any())
{
return; // DB has been seeded
}
```

Initialize DB with test data

C# (*DbInitializer.cs*)

```
var students = new Student[]
{
new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-01")},
new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},
new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},
new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}
};
foreach (Student s in students)
{
context.Students.Add(s);
}
context.SaveChanges();
```

Initialize DB with test data

C# (*DbInitializer.cs*)

```
var courses = new Course[]
{
new Course{CourseID=1050,Title="Chemistry",Credits=3},
new Course{CourseID=4022,Title="Microeconomics",Credits=3},
new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
new Course{CourseID=1045,Title="Calculus",Credits=4},
new Course{CourseID=3141,Title="Trigonometry",Credits=4},
new Course{CourseID=2021,Title="Composition",Credits=3},
new Course{CourseID=2042,Title="Literature",Credits=4}
};
foreach (Course c in courses)
{
context.Courses.Add(c);
}
context.SaveChanges();
```

Initialize DB with test data

C# (*DbInitializer.cs*)

```
var enrollments = new Enrollment[]
{
new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
new Enrollment{StudentID=3,CourseID=1050},
new Enrollment{StudentID=4,CourseID=1050},
new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
new Enrollment{StudentID=6,CourseID=1045},
new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
};
foreach (Enrollment e in enrollments)
{
context.Enrollments.Add(e);
}
context.SaveChanges();
}
}
}
```

Initialize DB with test data

- The preceding code checks if the database exists:
- If the database is not found;
  - It is created and loaded with test data. It loads test data into arrays rather than List<T> collections to optimize performance.
- If the database is found, it takes no action.
- Then Update *Program.cs* with the following code:

C# (Program.cs)

```
var app = builder.Build();

var scope = app.Services.CreateScope();
var context = scope.ServiceProvider.GetRequiredService<SchoolContext>();
DbInitializer.Initialize(context);
```

Initialize DB with test data

- Program.cs does the following on app startup:
  - Get a database context instance from the dependency injection container.
  - Call the *DbInitializer.Initialize* method.
- The first time the app is run, the database is created and loaded with test data.

When the app is started, the DbInitializer.Initialize method calls EnsureCreated. EF saw that there was no database:

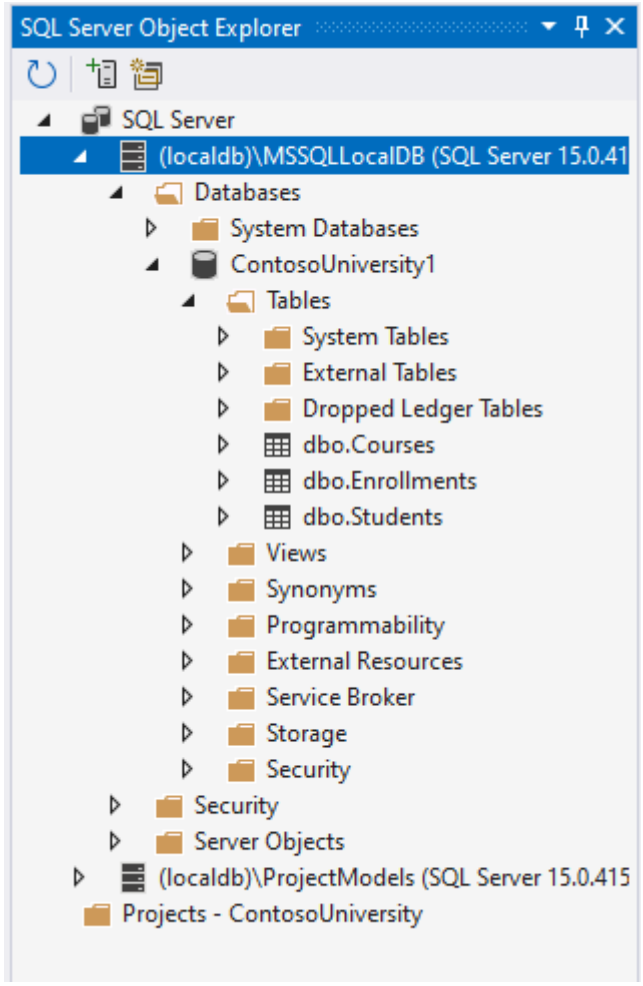
- So it created a database.
- The Initialize method code populated the database with data.

View the database

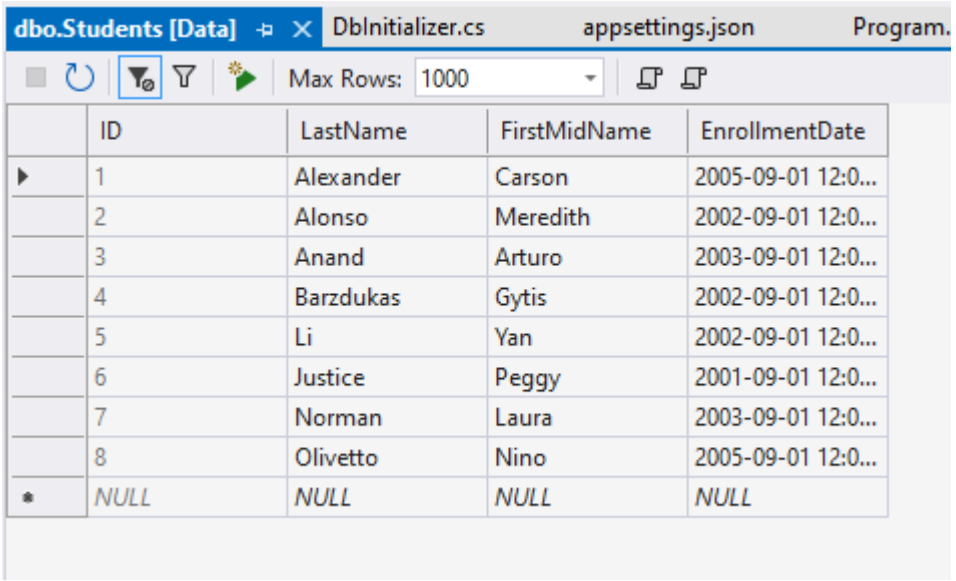
Use **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio:

- Select **SQL Server Object Explorer** from the **View** menu in Visual Studio. In SSOX, select **(localdb)\MSSQLLocalDB > Databases**.
- Select ContosoUniversity1, the entry for the database name that's in the connection string in the appsettings.json file.
- Expand the **Tables** node to see the tables in the database.

View the database



Right-click the **Student** table and click **View Data** to see the data in the table.



ID	LastName	FirstMidName	EnrollmentDate
1	Alexander	Carson	2005-09-01 12:0...
2	Alonso	Meredith	2002-09-01 12:0...
3	Anand	Arturo	2003-09-01 12:0...
4	Barzdukas	Gytis	2002-09-01 12:0...
5	Li	Yan	2002-09-01 12:0...
6	Justice	Peggy	2001-09-01 12:0...
7	Norman	Laura	2003-09-01 12:0...
8	Olivetto	Nino	2005-09-01 12:0...
NULL	*NULL*	*NULL*	*NULL*

DB code first migrations

- When you develop a new application, your data model changes frequently -- add, remove, or change entity classes or change your DbContext class --, and each time the model changes, it gets out of sync with the database.
- You usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column.
- The EF Core Code First Migrations feature solves this problem by enabling EF to update the database schema instead of creating a new database.
- Also, migrations can create a new database from scratch.
- To work with migrations, you can use the **Package Manager Console** (PMC) or the CLI.

Create Migration

To create an migration enter the following in PMC:

- dotnet ef migrations add “migration Name”
- Or
- dotnet-ef migrations add “migration Name”
- Or
- add-migration “migration Name”

Apply The Migration

enter the following command to create the database and tables in it.

- dotnet ef database update
- Or
- dotnet-ef database update
- Or
- update-database
- You'll notice the addition of an __EFMigrationsHistory table that keeps track of which migrations have been applied to the database.
- View the data in that table and you'll see one row for the first migration.

Creating a controller and a view for student Model

- Let's create an empty *controller* Name it as *StudentsController*.
- Inside its *Index* Action:

C# (StudentsController.cs)

```
public IActionResult Index()
{
Student s = new Student()
{
ID = 1,
FirstMidName = "Test1",
LastName = "Test1",
EnrollmentDate = DateTime.Now
};
return View(s);
}
```

- Create an empty View called Index with the following code:

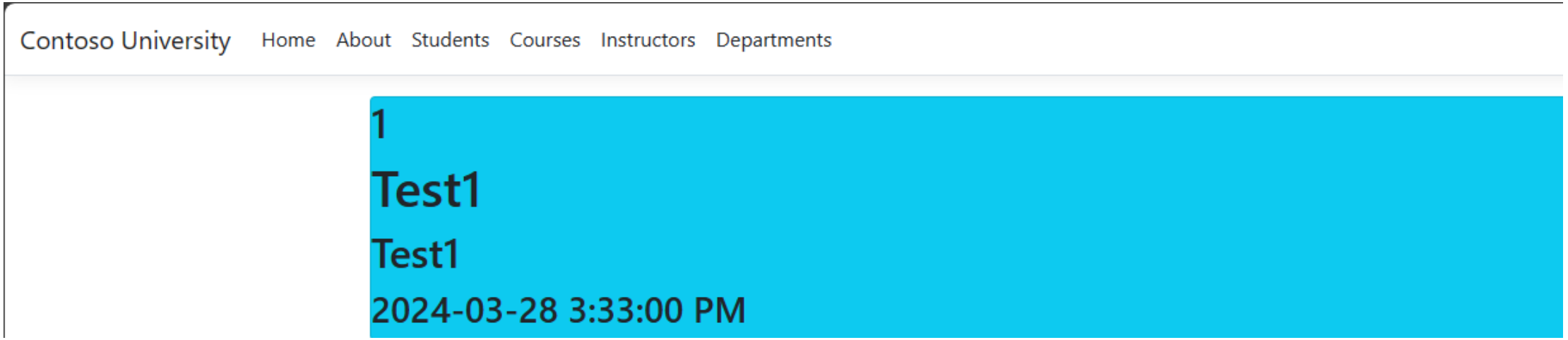
Creating a controller and a view for student Model

- Now we will see the **Strong_Typed Views** that use models objects.
- you can define what type the view can expect using an @model directive at the top.
- To interact with the instance of the model use ***Model*** (with an uppercase M).

CSHTML (Index.cshtml)

```
@model Student
@{
ViewData["Title"] = "Students";
}
<div class="card bg-info">
<h2>@Model.ID</h2>
<h1>@Model.FirstMidName</h1>
<h2>@Model.LastName</h2>
<h3>@Model.EnrollmentDate</h3>
</div>
```

Creating a controller and a view for student Model



Creating a controller and a view for student Model

- Let's pass a collection of students and display them in the View:

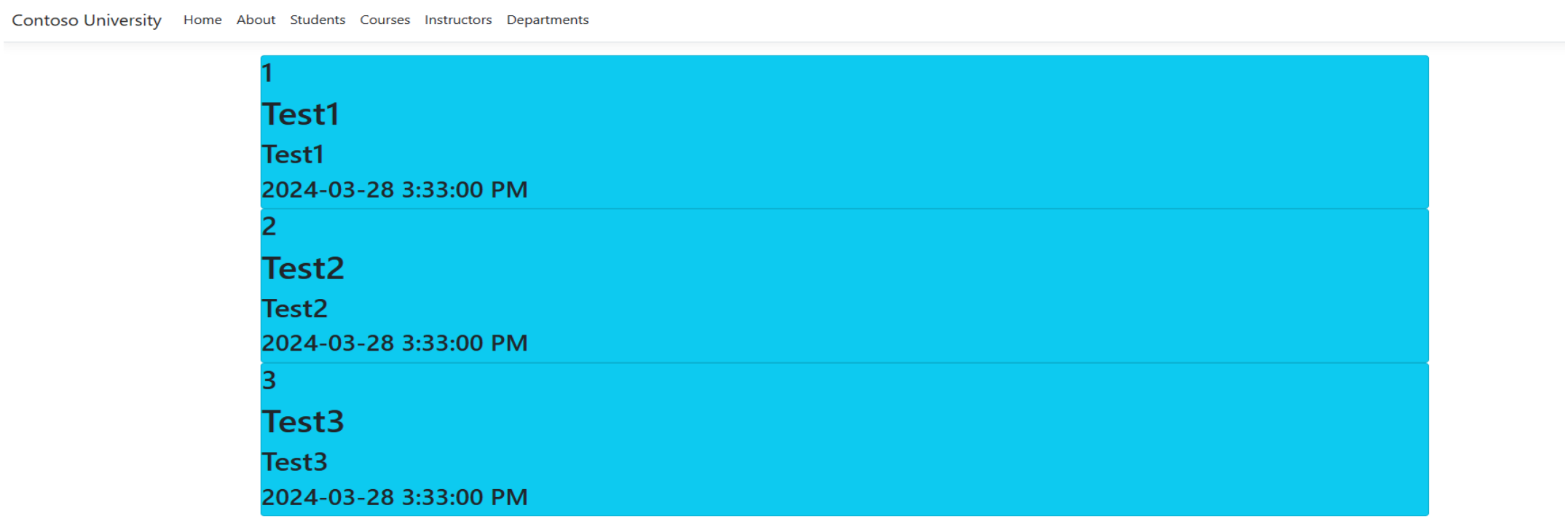
C# (StudentsController.cs)

```
public IActionResult Index()
{
List<Student> ss = new List<Student>() { new Student{
ID = 1, FirstMidName = "Test1",
LastName = "Test1",
EnrollmentDate = DateTime.Now },
new Student{
ID = 2, FirstMidName = "Test2",
LastName = "Test2",
EnrollmentDate = DateTime.Now },
new Student{
ID = 3, FirstMidName = "Test3",
LastName = "Test3",
EnrollmentDate = DateTime.Now },
};
return View(ss);
}
```

CSHTML (Index.cshtml)

```
@model List<Student>
@{
ViewData["Title"] = "Students";

}
@foreach(var s in Model)
{
<div class="card bg-info">
<h2>@s.ID</h2>
<h1>@s.FirstMidName</h1>
<h2>@s.LastName</h2>
<h3>@s.EnrollmentDate</h3>
</div>
}
```

Creating a controller and a view for student Model



Creating a controller and a view for student Model

- Let’s Show students data from database:
- To do this you need an object from the *SchoolContext* so, make the constructor of the controller takes a *SchoolContext* as a constructor parameter.
- ASP.NET Core dependency injection takes care of passing an instance of *SchoolContext* into the controller.
- In the controller’s *Index* action method, which displays all students in the database. The method gets a list of students from the Students entity set by reading the *Students* property of the *database context* instance.
- The code of the controller and view will be:

Creating a controller and a view for student Model

C# (StudentsController.cs)

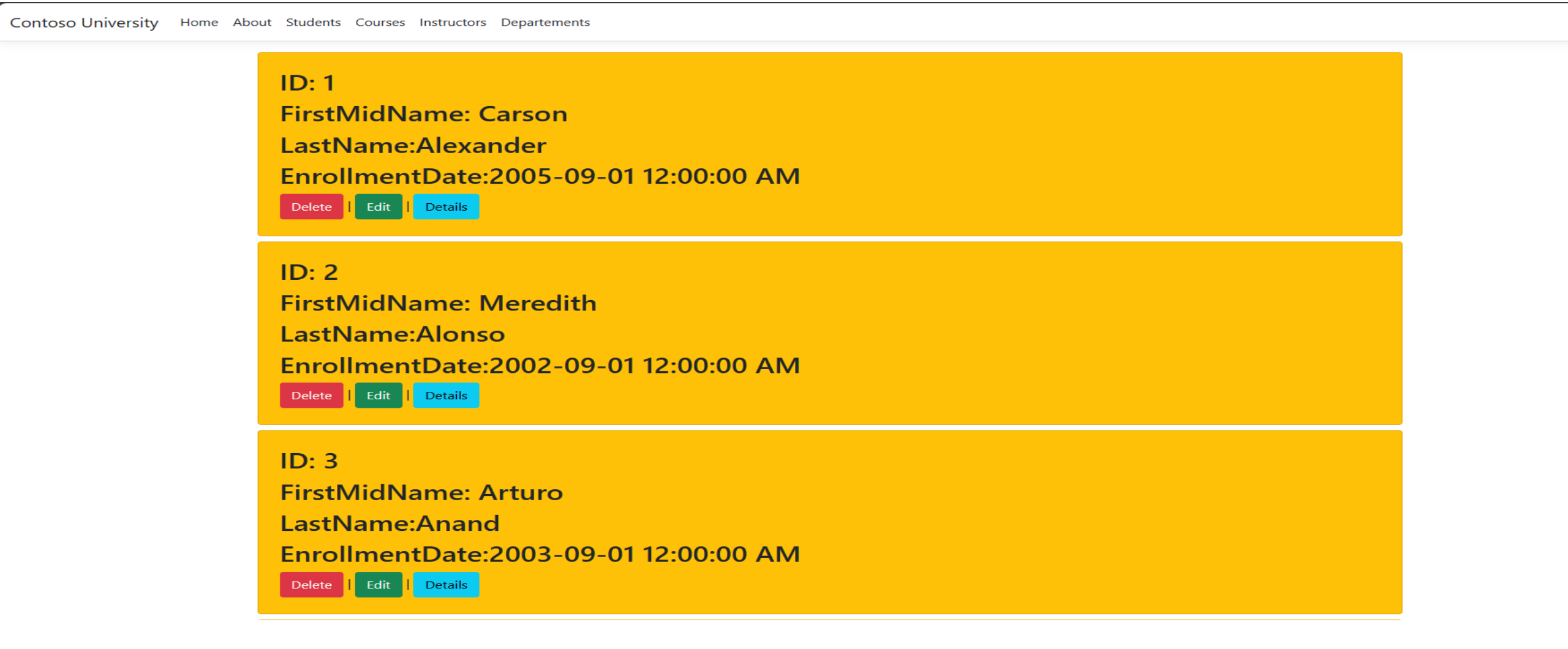
```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;

namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;
        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
        public IActionResult Index()
        {
            return View(_context.Students.ToList());
        }
    }
}
```

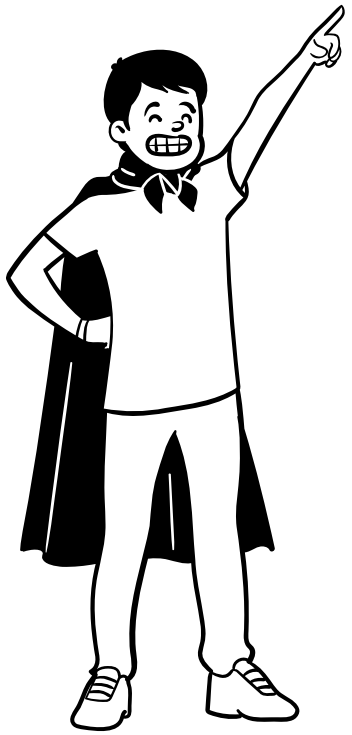
Creating a controller and a view for student Model

CSHTML (Index.cshtml)

```
@model IEnumerable<Student>
@{
ViewData["Title"] = "Students";
}
@foreach(var s in Model)
{
<div class="card bg-warning m-2">
<div class="card-body m-2">
<h3>@Html.DisplayNameFor(s => s.ID): @s.ID</h3>
<h2>@Html.DisplayNameFor(s => s.FirstMidName): @s.FirstMidName</h2>
<h3>@Html.DisplayNameFor(s => s.LastName): @s.LastName</h3>
<h4>@Html.DisplayNameFor(s => s.EnrollmentDate): @s.EnrollmentDate</h4>
<a class="btn btn-success" asp-action="Edit" asp-route-id="@s.ID">Edit</a> |
<a class="btn btn-primary" asp-action="Details" asp-route-id="@s.ID">Details</a> |
<a class="btn btn-danger" asp-action="Delete" asp-route-id="@s.ID">Delete</a>
</div>
</div>
}
```

Creating a controller and a view for student Model





Any Questions?


THANK
YOU!