



.NET Core: Developing Cross-Platform Web Apps with ASP.NET Core – Workshop*PLUS*

Wael Kdouh - @waelkdouh

Senior Customer Engineer

v3.0

Conditions and Terms of Use

Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

© 2016 Microsoft Corporation. All rights reserved.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at

<http://www.microsoft.com/en-us/legal/intellectualproperty/permissions/default.aspx>

Azure, Microsoft, Microsoft Corporate Logo, SQL Server, Visual Studio and Windows Phone are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

Module 2: Models

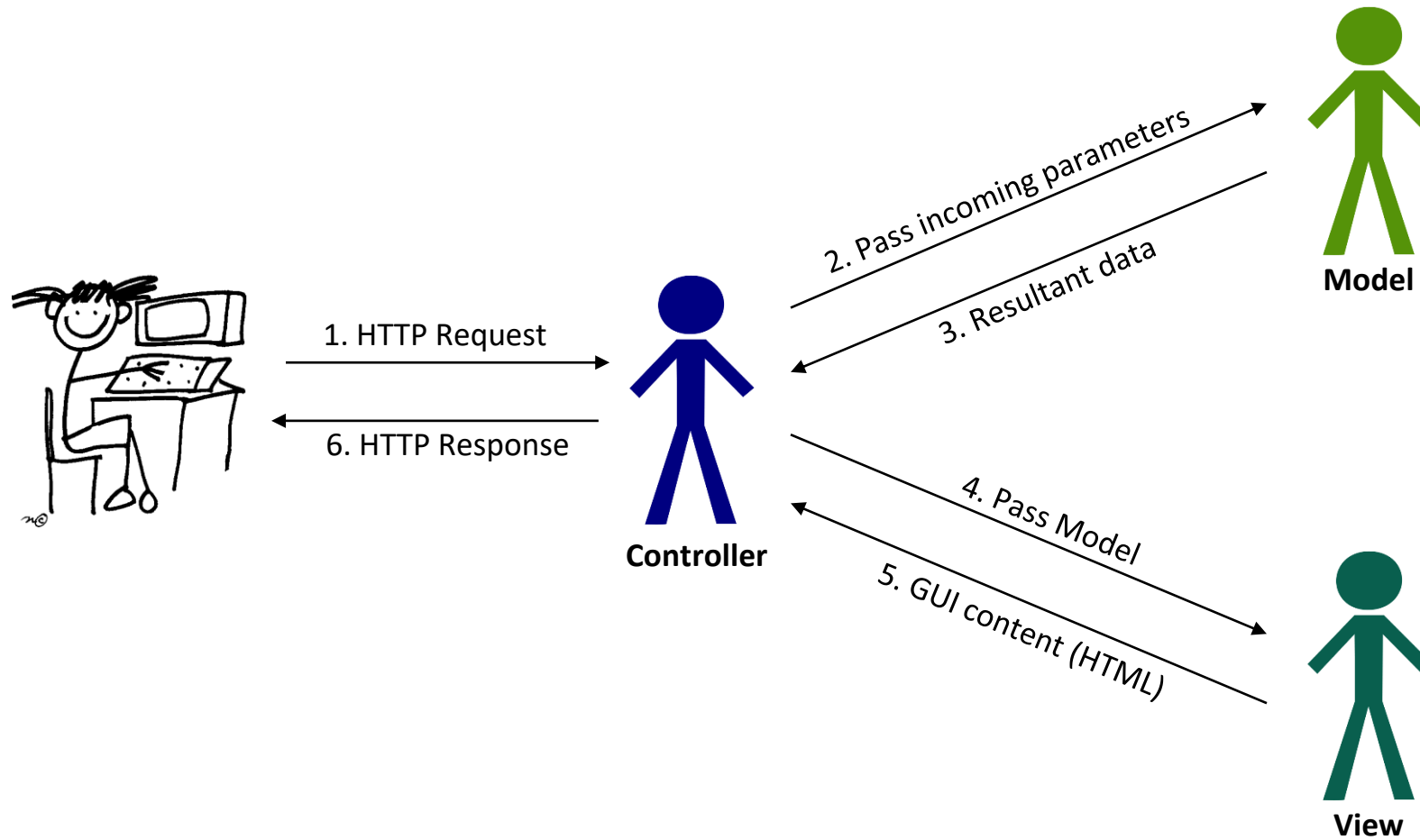
Module Overview

Module 2: Models

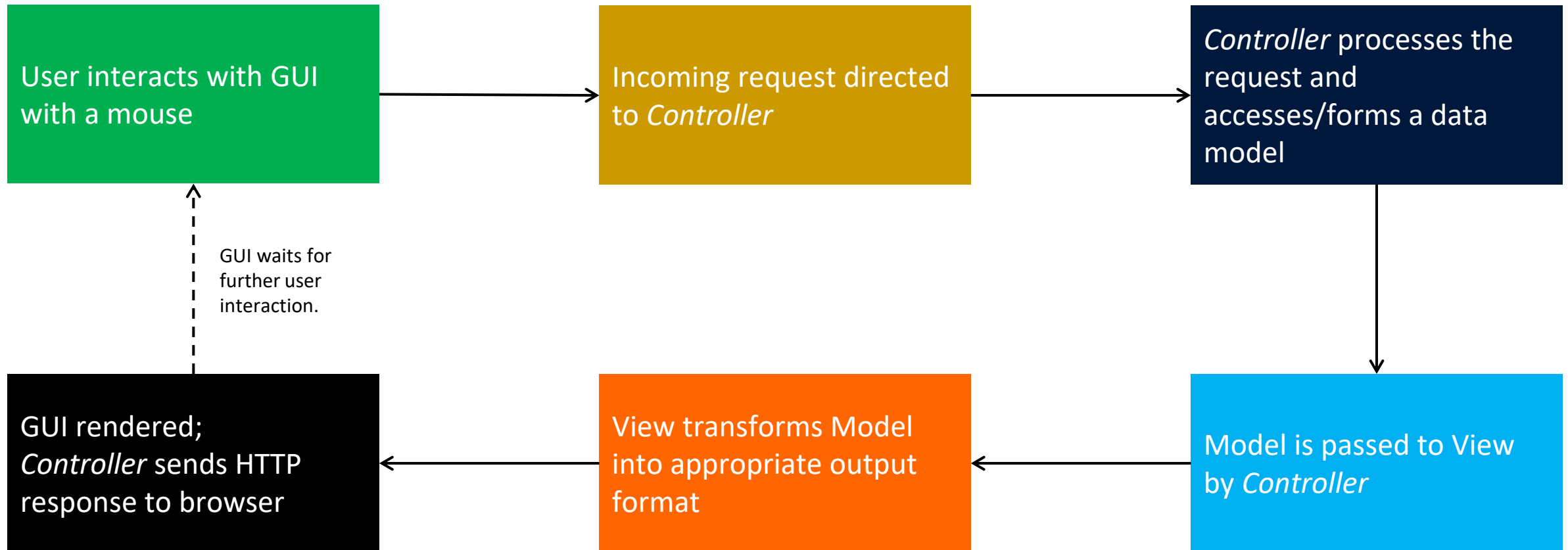
Section 1: MVC Design Pattern

Lesson: Overview

Model View Controller (MVC) Design Pattern



MVC Control Flow



Module 2: Models

Section 2: Model Fundamentals

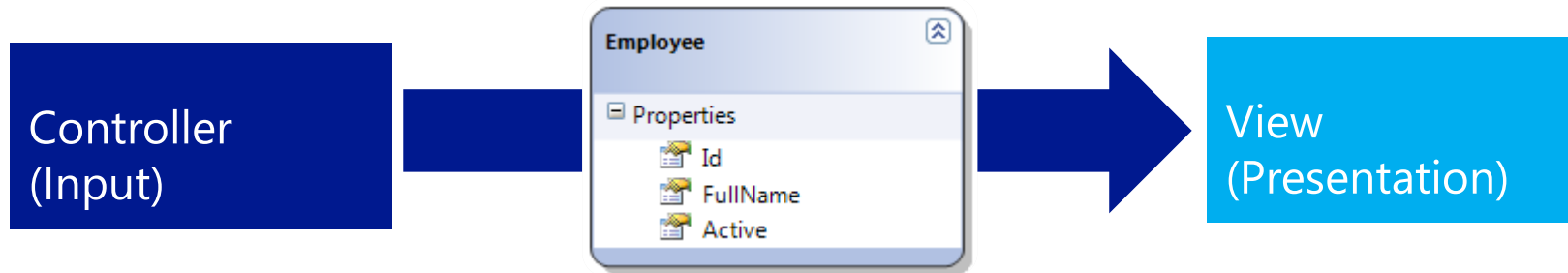
Lesson: Role of Models

Model

- A set of .NET classes that:
 - Describe data that the application is working with
 - Implement the **rules** or **logic** for how the data can be changed/manipulated
- Model state can be retrieved and stored in any form:
 - Relational databases
 - Comma-separated text files
 - RESTful web services
- It can use any data access technology for accessing and manipulating data
 - Object Relational Mapping frameworks like Entity Framework (EF)

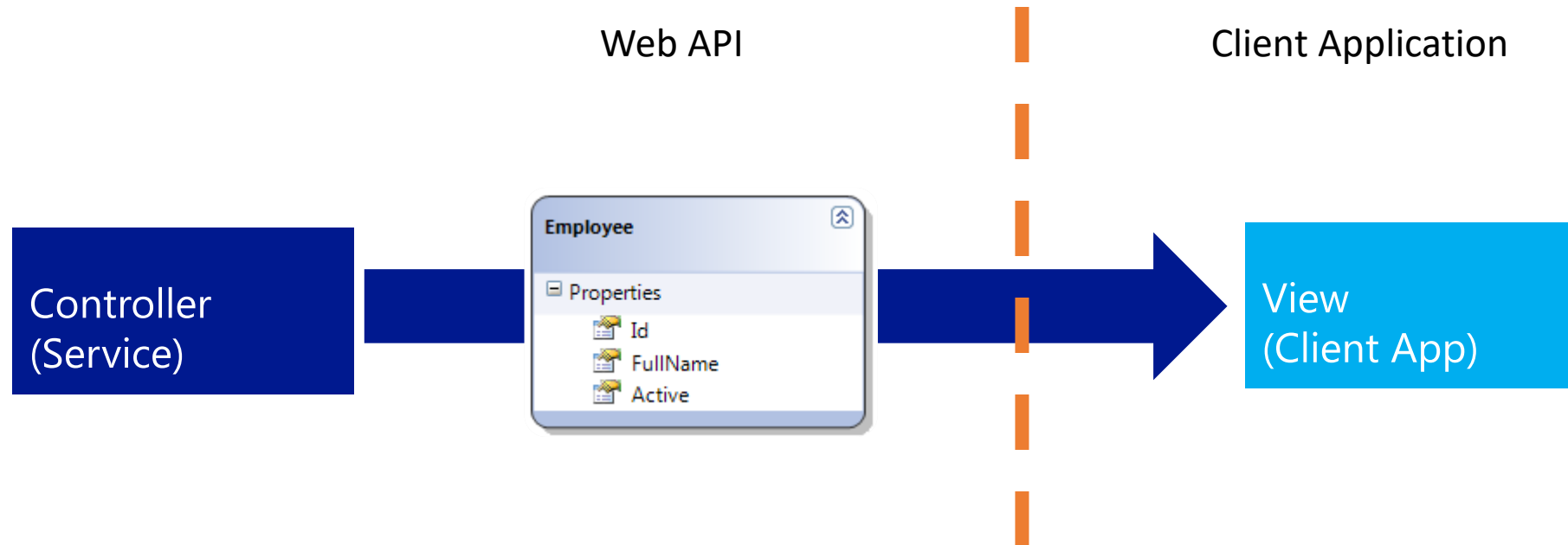
Role of a Model

- The “Model” is the medium of communication between **Controllers** and **Views**
- It responds to requests for information about its state (usually from view)
- It changes states in the data source as per the request of controller



Role of a Model

- Building a RESTful service or WebAPI? The pattern still applies!



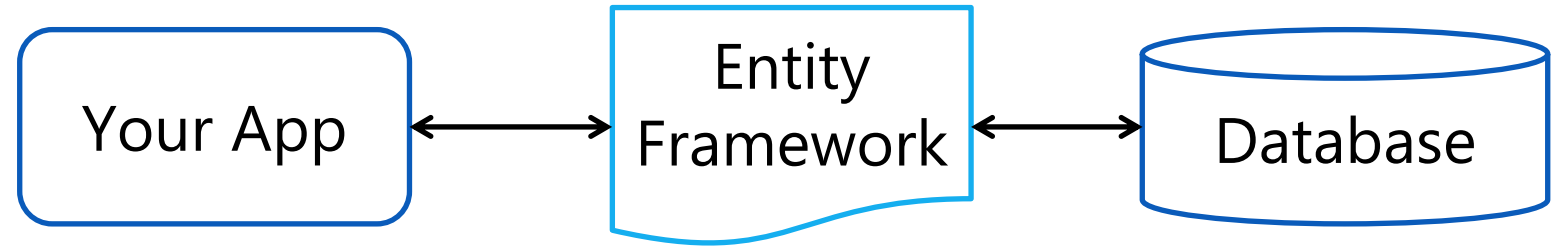
Module 2: Models

Section 3: Model Development

Lesson: Development with Entity Framework

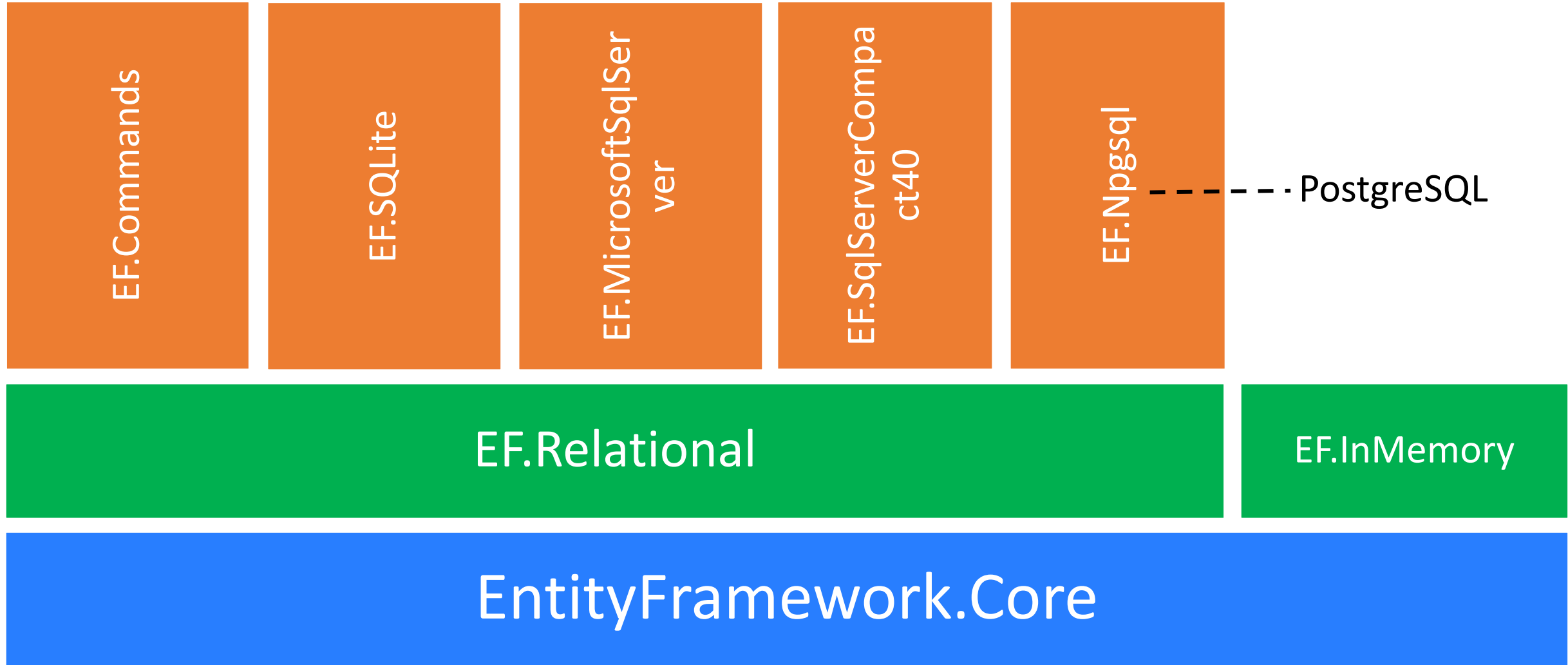
Entity Framework Core

- Object-relational mapping framework by Microsoft
 - It understands how to store .NET objects in a **relational** database.
 - It retrieves and manipulates data as strongly typed objects using LINQ query
- It provides:
 - Change tracking
 - Identity resolution
 - Dev-time tooling
 - Query translation
 - More!
- Open-source and Cross-platform!
- Both Entity Framework v6 and Core will continue to develop separately

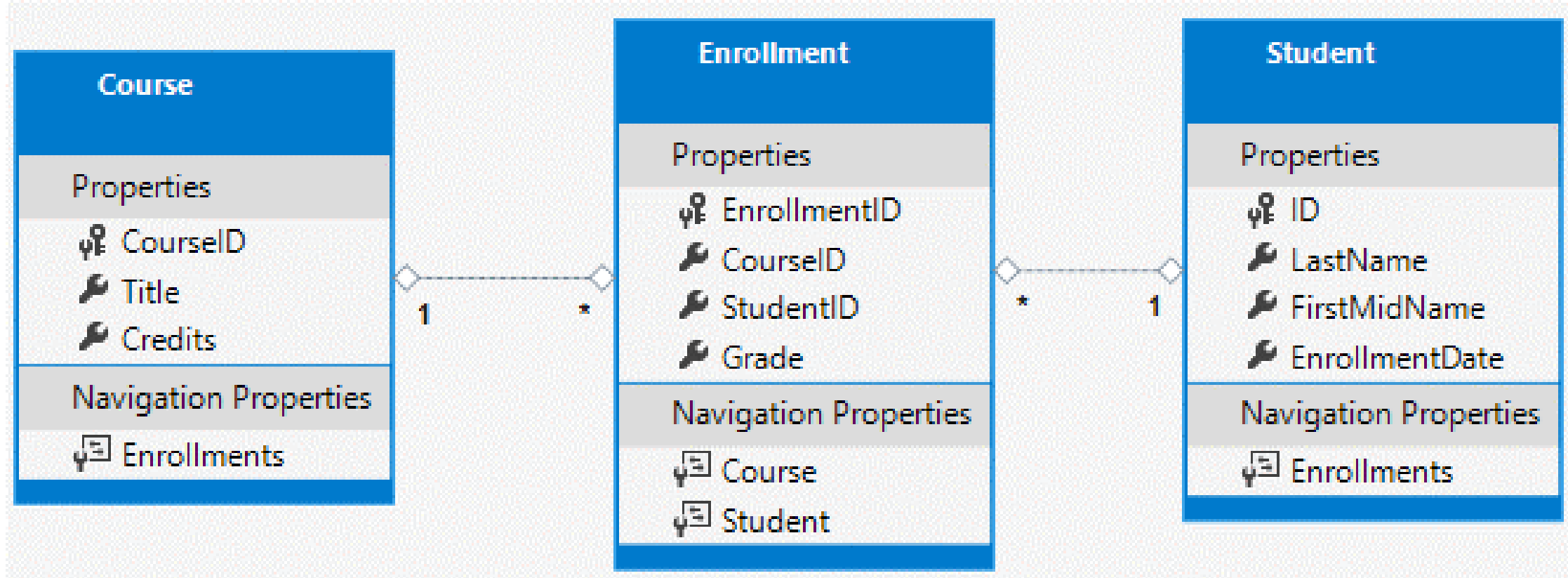


Note: It is not mandatory to be used with Model View Controller (MVC) and vice-versa

EF Architecture



Our Data Domain – Contoso University



Model Development

- A model can be created with a .NET class
- Primary key, foreign key, and navigation properties are defined in the class
- Class (Enrollment) will be converted into a database table
- Class variables (*EnrollmentID*, *CourseID*, etc.) will be converted into table attributes

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```


Model Relationships

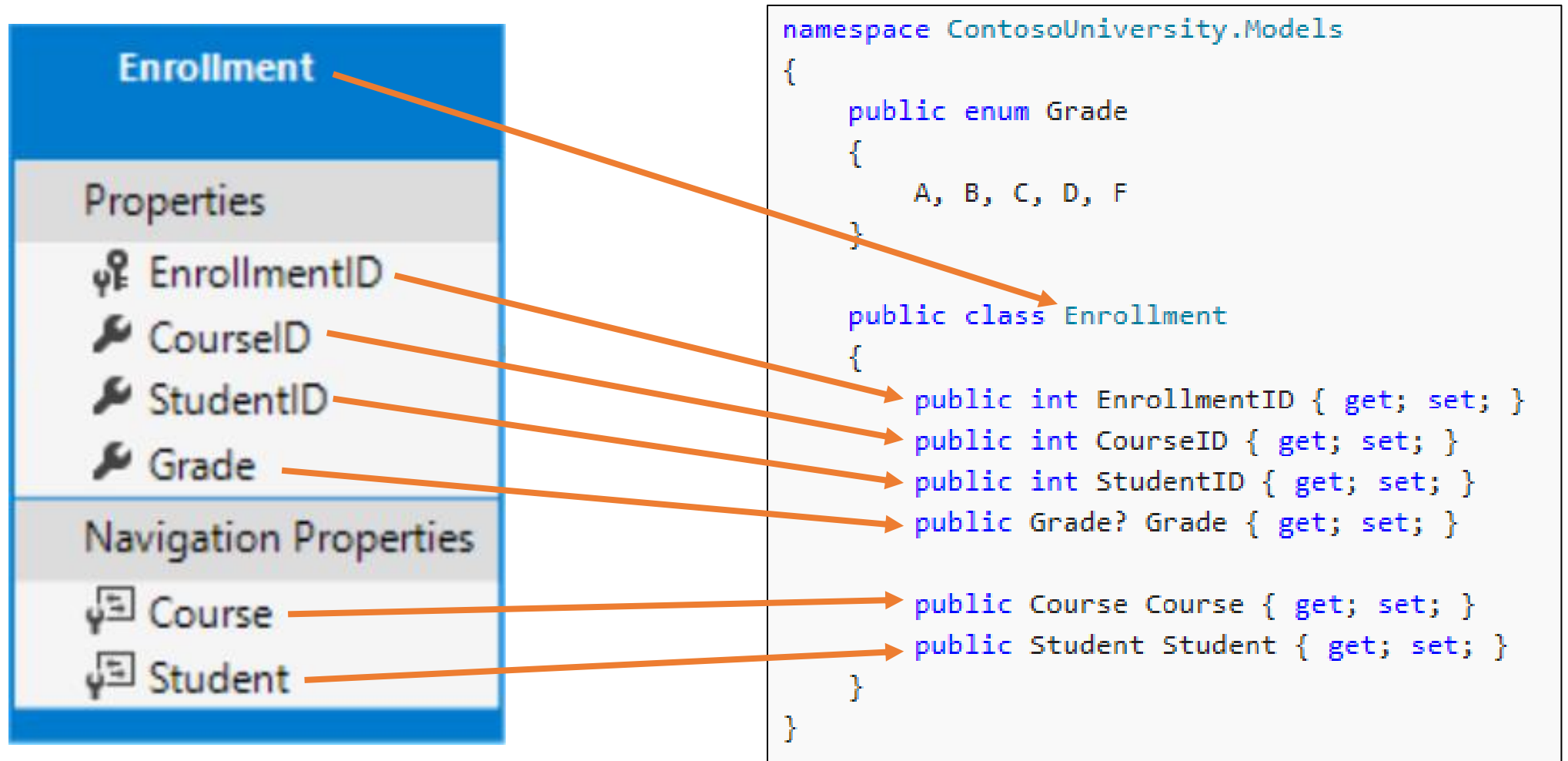
- Navigation property
 - Navigational property holds other entities that are related to this entity
 - *Student* and *Course* are navigation properties.
- Foreign key property
 - It is not required in a model object
 - It is used for convenience
 - *CourseID* and *StudentID* are foreign key properties

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Model Relationships



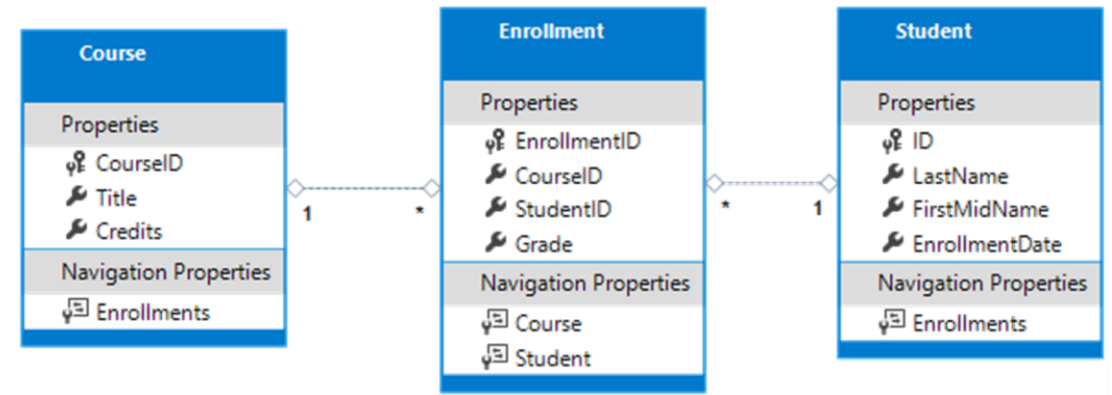
Model Relationships - DbContext

- The DbContext is how we expose our classes to EF
- Inherits from `Microsoft.EntityFrameworkCore.DbContext`
- It is also our gateway into the database in code

```
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; }
}
```

- `DbSet<T>` Where T is Class
 - How we tell EF which models to track relationships between
- Here we have told EF to track *Course*, *Enrollment*, and *Student* entities and their relationships



View Specific Models (DTOs)

- Data Transfer Objects (DTOs) or ViewModels can be used to create versions of your Entities that can be sent over the internet
- Prevents exposing schema information like relationships
- Creates more specific models for an application, like a flattened search result model
- Can be manually mapped or use a library like AutoMapper to move from entity to DTO and back

```
public class Enrollment
{
    5 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int EnrollmentID { get; set; }
    12 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int CourseID { get; set; }
    12 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int StudentID { get; set; }
    9 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Grade? Grade { get; set; }

    2 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Course Course { get; set; }
    2 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Student Student { get; set; }
}
```

View Specific Models (DTOs)

```
public class Enrollment
{
    5 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int EnrollmentID { get; set; }
    12 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int CourseID { get; set; }
    12 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int StudentID { get; set; }
    9 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Grade? Grade { get; set; }

    2 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Course Course { get; set; }
    2 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Student Student { get; set; }
}
```

```
public class EnrollmentDTO
{
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int EnrollmentID { get; set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int StudentID { get; set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string StudentLastName { get; set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string StudentFirstName { get; set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public DateTime StudentEnrollmentDate { get; set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int CourseID { get; set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public string CourseTitle { get; set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int CourseCredits { get; set; }
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public Grade? Grade { get; set; }
}
```

Demo: Code-based Model

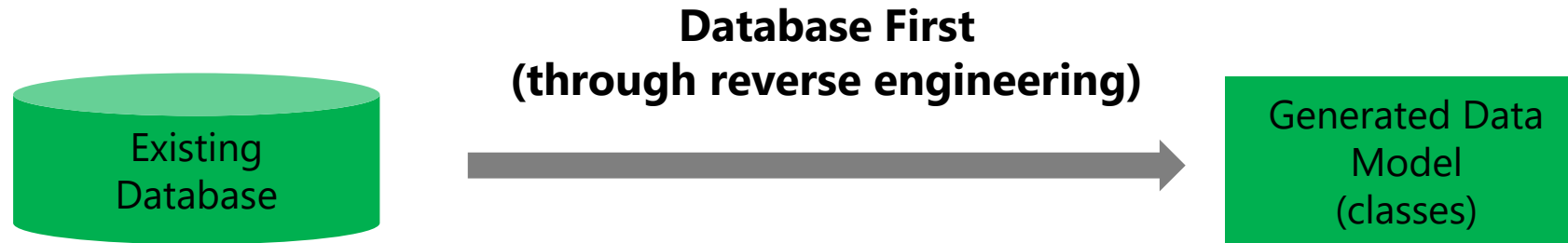
Module 2: Models

Section 3: Model Development

Lesson: Code-based Modeling

Entity Framework Core **only** supports
Code-based Modeling
(that is, Code First approach)

Entity Framework Development Approaches



Code-based modeling is the only approach supported in Entity Framework Core

Code-First Development

- Model code is written in .NET classes; model and database are created from the code
 - .NET Classes correspond to database tables
 - Properties correspond to database table columns
 - Classes can be used with or without EF!
- Relationships can be customized via the fluent API in the OnModelCreating override
- Code First can also work with existing database
 - Code is used for mapping instead of visual designer and XML

Tooling

- Entity Framework Core dotnet CLI – Our dev/design-time tooling
- Add the following to your .csproj file

```
<ItemGroup>  
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />  
</ItemGroup>
```

.csproj

dotnet ef must be installed as a global or local tool

Most developers will install dotnet ef as a global tool with the following command:
dotnet tool install --global dotnet-ef

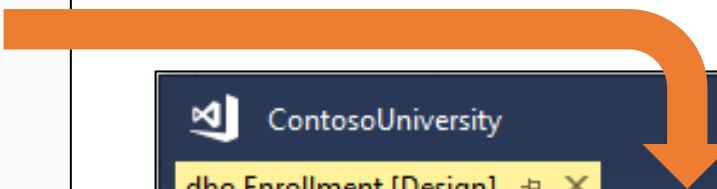
- Enables dotnet ef * commands at the command line in the project directory, e.g.,
 - dotnet ef migrations add Initial
 - dotnet ef database update Initial
 - dotnet ef dbcontext scaffold ...

Database Creation Using Entity Framework - I

- Model is created using .NET classes
- Executed using dotnet ef CLI tooling

```
public class Enrollment
{
    public int EnrollmentID { get; set; }
    public int CourseID { get; set; }
    public int StudentID { get; set; }
    public Grade? Grade { get; set; }

    public Course Course { get; set; }
    public Student Student { get; set; }
}
```



ContosoUniversity

dbo.Enrollment [Design] ✕

Update | Script File: **dbo.Enrollment.sql**

	Name	Data Type	Allow Nulls	Default
PK	EnrollmentID	int	<input type="checkbox"/>	
	CourseID	int	<input type="checkbox"/>	
	Grade	int	<input checked="" type="checkbox"/>	
	StudentID	int	<input type="checkbox"/>	
			<input type="checkbox"/>	

Database Creation Using Entity Framework - II

- Tooling scans project for `Microsoft.EntityFrameworkCore.DbContext` based classes
- Contexts are used as the entry point into your code base
- We can override or reinforce how EF interprets relationships via the fluent API

```
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; }
}
```

For example, EF will create Students, Enrollments and Courses tables in the database

Database Seeding

- Database Initializers are deprecated in EF Core
 - Use DI to write and inject your own
- This code should be executed in Main, outside your app

```
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
        }

        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();
    }
}
```


Database Seeding

- Database Initializers are deprecated in EF Core
 - Use DI to write and inject your own
- This code should be executed in Main, outside your app

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the database.");
        }
    }

    host.Run();
}
```

Program.cs

Configuring Connections with Entity Framework

- Database connection string is typically stored in configuration (often appsettings.json)

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

Startup.cs

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

appsettings.json

Code First Migrations

- Enables changing the data model and deploying the change in production without dropping and re-creating the database
- Effective strategy for real-world production databases
- **Up** method used for creating/updating database schema
- **Down** method used for rollback logic
- Maintains version of each change
- Not required, but are very helpful if your schema changes

Migration Methods

Up Method

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Course",
        columns: table => new
        {
            CourseID = table.Column<int>(type: "int", nullable: false),
            Credits = table.Column<int>(type: "int", nullable: false),
            Title = table.Column<string>(type: "nvarchar(max)", nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Course", x => x.CourseID);
        });

    migrationBuilder.CreateTable(
        name: "Student",
        columns: table => new
        {
            StudentID = table.Column<int>(type: "int", nullable: false)
                .Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),
            EnrollmentDate = table.Column<DateTime>(type: "datetime2", nullable: false),
            FirstName = table.Column<string>(type: "nvarchar(max)", nullable: true),
            LastName = table.Column<string>(type: "nvarchar(max)", nullable: true)
        },
        constraints: table =>
```

Down Method

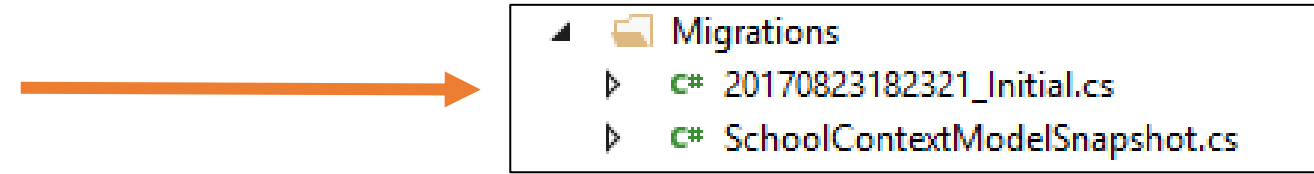
```
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Enrollment");

    migrationBuilder.DropTable(
        name: "Course");

    migrationBuilder.DropTable(
        name: "Student");
}
```

Creating and Applying Migrations

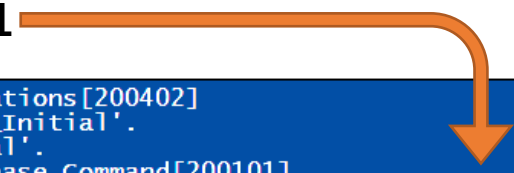
\$ dotnet ef migrations add Initial



Migrations

- 20170823182321_Initial.cs
- SchoolContextModelSnapshot.cs

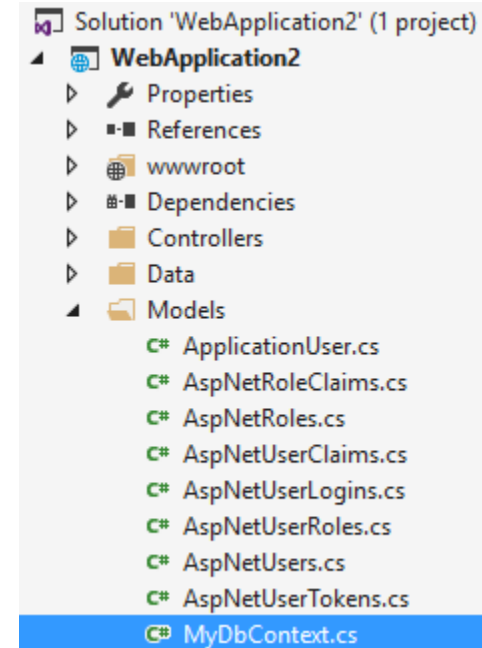
\$ dotnet ef database update Initial



```
info: Microsoft.EntityFrameworkCore.Migrations[200402]
  Applying migration '20170823182321_Initial'.
Applying migration '20170823182321_Initial'.
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
  Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
  CREATE TABLE [Course] (
    [CourseID] int NOT NULL,
    [Credits] int NOT NULL,
    [Title] nvarchar(max) NULL,
    CONSTRAINT [PK_Course] PRIMARY KEY ([CourseID])
  );
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
  Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
  CREATE TABLE [Student] (
    [StudentID] int NOT NULL IDENTITY,
    [EnrollmentDate] datetime2 NOT NULL,
    [FirstName] nvarchar(max) NULL,
    [LastName] nvarchar(max) NULL,
    CONSTRAINT [PK_Student] PRIMARY KEY ([StudentID])
  );
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
  Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
  CREATE TABLE [Enrollment] (
    [EnrollmentID] int NOT NULL IDENTITY,
    [CourseID] int NOT NULL,
    [Grade] int NULL,
    [StudentID] int NOT NULL,
    CONSTRAINT [PK_Enrollment] PRIMARY KEY ([EnrollmentID]),
    CONSTRAINT [FK_Enrollment_Course_CourseID] FOREIGN KEY ([CourseID]) REFERENCES [Course] ([CourseID]) ON DELETE CASCADE,
    CONSTRAINT [FK_Enrollment_Student_StudentID] FOREIGN KEY ([StudentID]) REFERENCES [Student] ([StudentID]) ON DELETE CASCADE
  );
```

Entity Framework Tools & CLI

```
C:\Users\igorsych\Documents\Visual Studio 2015\Projects\WebApplication2>dotnet ef dbcontext scaffold "Data Source=mydb.database.windows.net;Initial Catalog=mydbcatalog;Password=P@ssw0rd!w;User ID=igorsych@mydb" "Microsoft.EntityFrameworkCore.SqlServer" --context MyDbContext --output-dir Models
Project WebApplication2 (.NETCoreApp,Version=v1.0) will be compiled because Input items added from last build
Compiling WebApplication2 for .NETCoreApp,Version=v1.0
Done
```



Demo: Entity Framework Code First (DB Creation)

Code First (Existing Database)

- Database schema reverse-engineered to Model classes
- Creates POCO classes
- POCO classes modified to customize database generation
- Corresponding partial classes used for customization
- Originally generated classes are replaced with each generation
- Indexes, functions and stored procedures ignored

Demo: Entity Framework Code First (with Existing Database)

Module 2: Models

Section 4: Model Design

Lesson: Code First Development

Code-First Conventions - I

- Naming
 - Class Name or Object Type → Table Name
- Primary Key
 - Property named 'Id' or '<class name>Id' → Primary key value
 - Auto-increment is set for primary key values
- Relationship Inverses
 - Both types define *only one* navigation property
 - `Product.Category` and `Category.Products` represents different ends of the same relationship

```
public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public Category Category { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    public string Name { get; set; }
    public ICollection<Product> Products { get; set; }
}
```

Code-First Conventions - II

- Type Discovery
 - Referenced object types are automatically included in the model without explicitly registering them as object sets
- Foreign Keys
 - The following conventions are used for foreign keys:
 - <navigation property name> <primary key property name>
that is, '**SubjectISBN**';
 - <principal class name> <primary key property name>
that is, '**BookISBN**';
 - <primary key property name> that is, '**ISBN**';
- Code-First conventions can be overridden using **Data Annotations**, which can in turn be overridden using Fluent API

```
public class BookReview
{
    public int Id { get; set; }
    public Book Subject { get; set; }
    public string SubjectISBN { get; set; }
}

public class Book
{
    [Key]
    public string ISBN { get; set; }
    public string Name { get; set; }
    public ICollection<BookReview> Reviews { get; set; }
}
```

View-Specific Model

- It is a model that exists just to supply information to a view
- It is mostly used for views that show accumulated data from different tables
- It is also used to prevent “over-posting” attack

```
public class Review
{
    public int ReviewID { get; set; } // Primary key
    public int ProductID { get; set; } // Foreign key
    public Product Product { get; set; } // Foreign entity
    public string Name { get; set; }
    public string Comment { get; set; }
    public bool Approved { get; set; }
}
```

Model created to
exclude *Approved* status



```
public class ReviewViewModel
{
    public string Name { get; set; }
    public string Comment { get; set; }
}
```

EF Core Fluent API

- Used inside of the `OnModelCreating` override in your `DbContext`
 - As of 2.0, can be defined in their own class and invoked inside `OnModelCreating`
- Can be used to override convention, explicitly define relationships, define custom conventions

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    ...
}
```

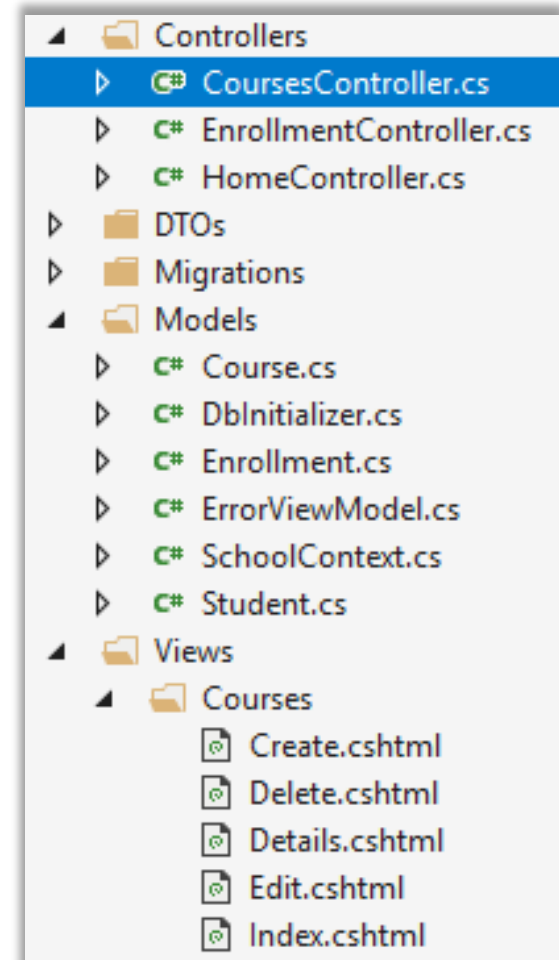
Module 2: Models

Section 4: Model Design

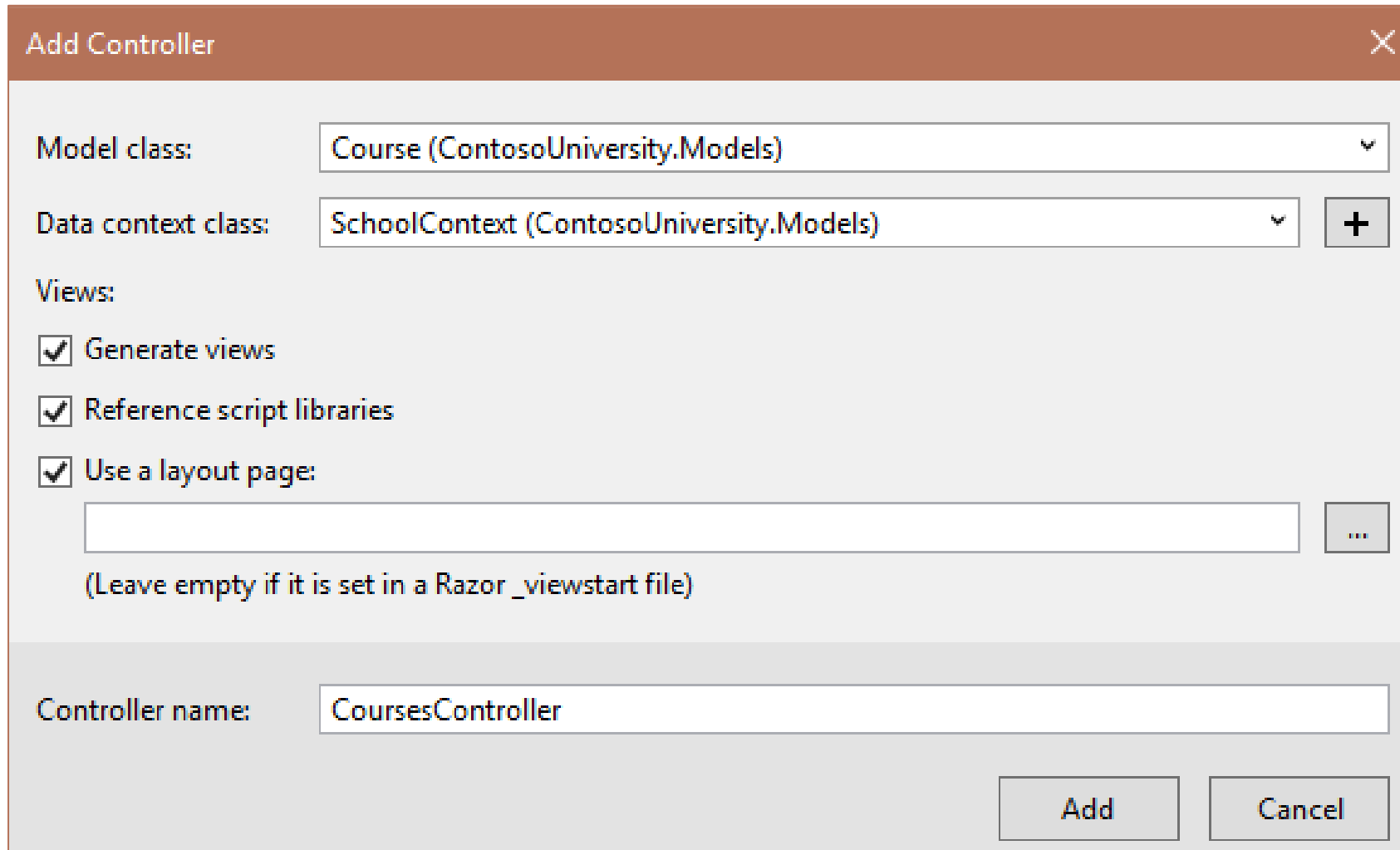
Lesson: Scaffolding

Scaffolding

- It means generating code for Create, Read, Update, and Delete (CRUD) functionality against a model
- It examines the type definition of model(s) to:
 - Generate controller(s)
 - Generate Controller's associated views
- It automatically names controllers and views
- All the generated controllers and views are placed correctly in the project structure



ASP.NET MVC Scaffolding in Visual Studio



The image shows the 'Add Controller' dialog box in Visual Studio. The dialog has a title bar with a close button. It contains several fields and checkboxes. The 'Model class' is set to 'Course (ContosoUniversity.Models)'. The 'Data context class' is set to 'SchoolContext (ContosoUniversity.Models)' with a plus button next to it. Under the 'Views' section, there are three checked checkboxes: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. Below the 'Use a layout page' checkbox is an empty text box and a button with three dots. A note below the text box says '(Leave empty if it is set in a Razor _viewstart file)'. At the bottom, the 'Controller name' is set to 'CoursesController'. There are 'Add' and 'Cancel' buttons at the bottom right.

Add Controller

Model class: Course (ContosoUniversity.Models)

Data context class: SchoolContext (ContosoUniversity.Models) +

Views:

- ☒ Generate views
- ☒ Reference script libraries
- ☒ Use a layout page:

...

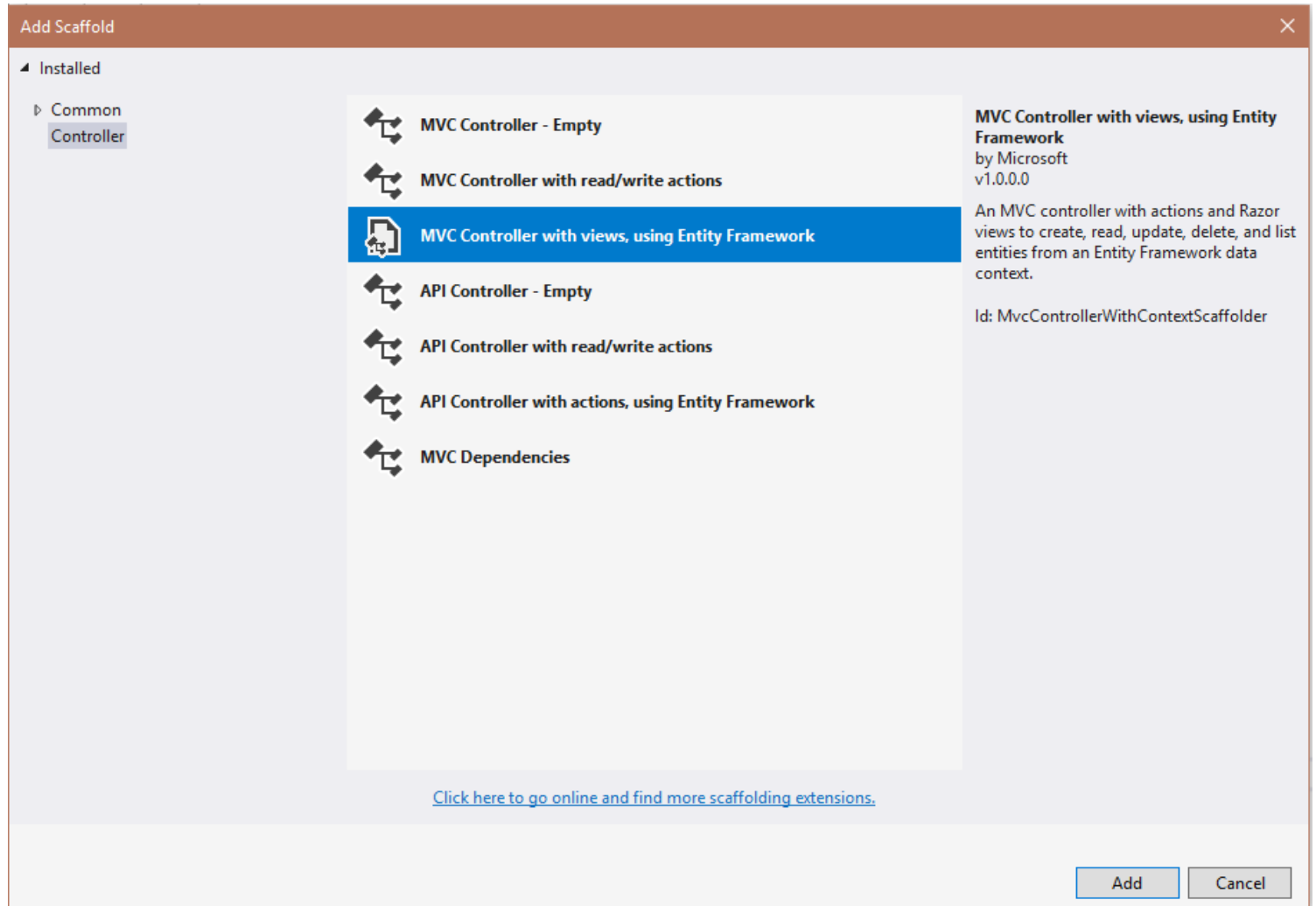
(Leave empty if it is set in a Razor _viewstart file)

Controller name: CoursesController

Add Cancel

Scaffolding Templates

- Scaffolding template determines how far would it go with code generation
- Alternative scaffolding templates are available through **NuGet**



Demo: Scaffolding Model Development

Module 2: Models

Section 4: Model Design

Lesson: Model Binding

Model Binding

- Model binder: Automatically maps posted form value to a .NET framework type based on naming conventions
- **Default Model Binder** is a default Model Binder implementation
 - Takes care of mundane property mapping and type conversion
 - Uses the name attribute of input elements
 - Automatically matches parameter names for simple data types
 - Complex objects are mapped by property name; use dotted notation

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.UserName };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // ...
        }
    }
}
```

```
public class RegisterViewModel
{
    1 reference
    public string UserName { get; set; }
    1 reference
    public string Password { get; set; }
    0 references
    public string ConfirmPassword { get; set; }
}
```

Async Query and Save

- What is it?
 - Task based async pattern for query and save
- Why did we build it?
 - Appropriate use of async can improve performance and scalability
- When should you use it?
 - Reduce server resource usage by freeing up blocked threads
 - Improve client UI responsiveness by not blocking main thread
 - Parallelism – but not on the same context instance

Module Summary

- In this module, you learnt about:
 - Model and its role in MVC pattern
 - Model development
 - Entity Framework Core
 - Scaffolding and scaffolding templates
 - Entity Framework development approaches
 - Code-first development and conventions
 - View-specific Model
 - Model binding and security
 - Model development Strategies



Lab: Models



