.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

Copyright and Trademarks

Module 2: Models

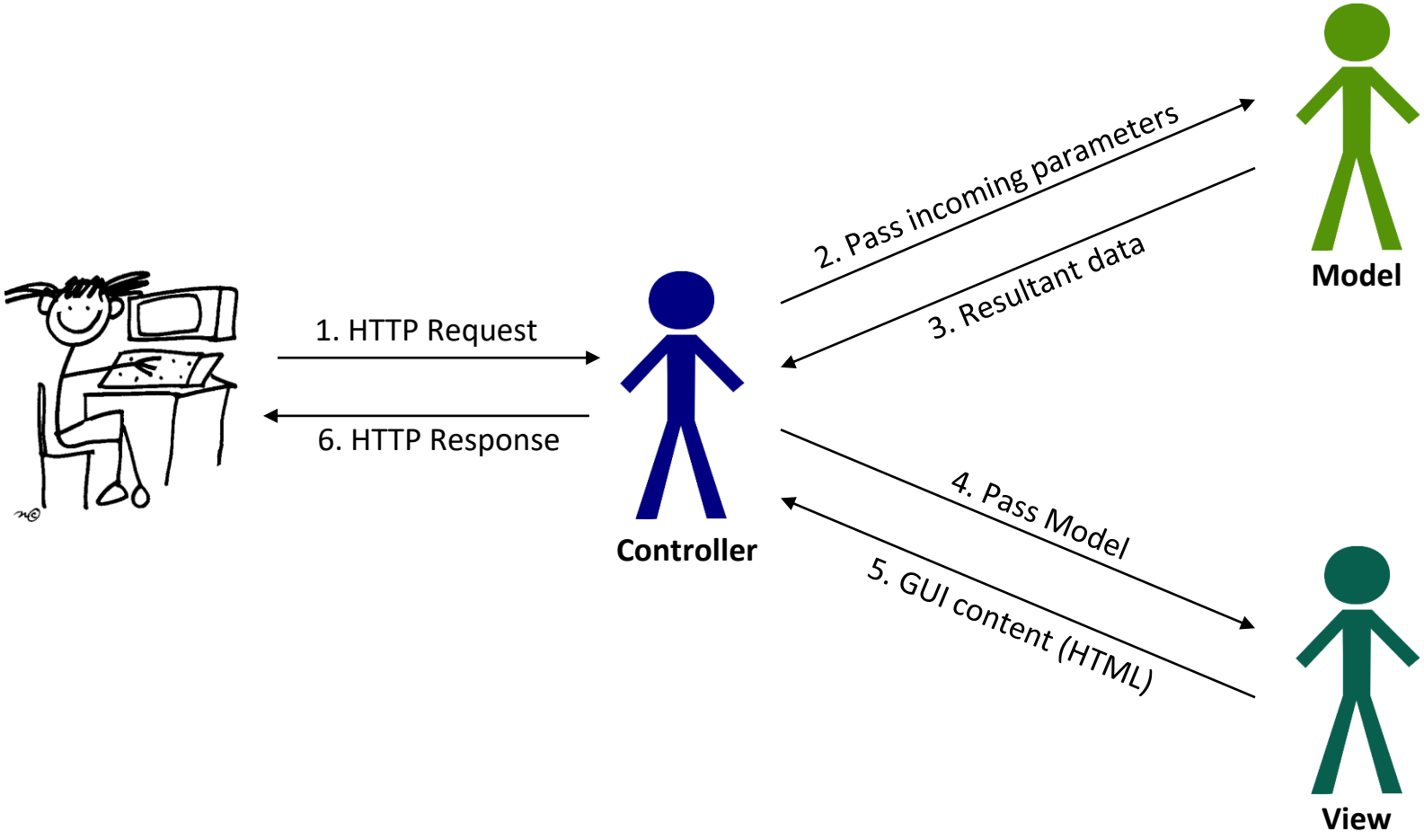
Module Overview

Module 2: Models

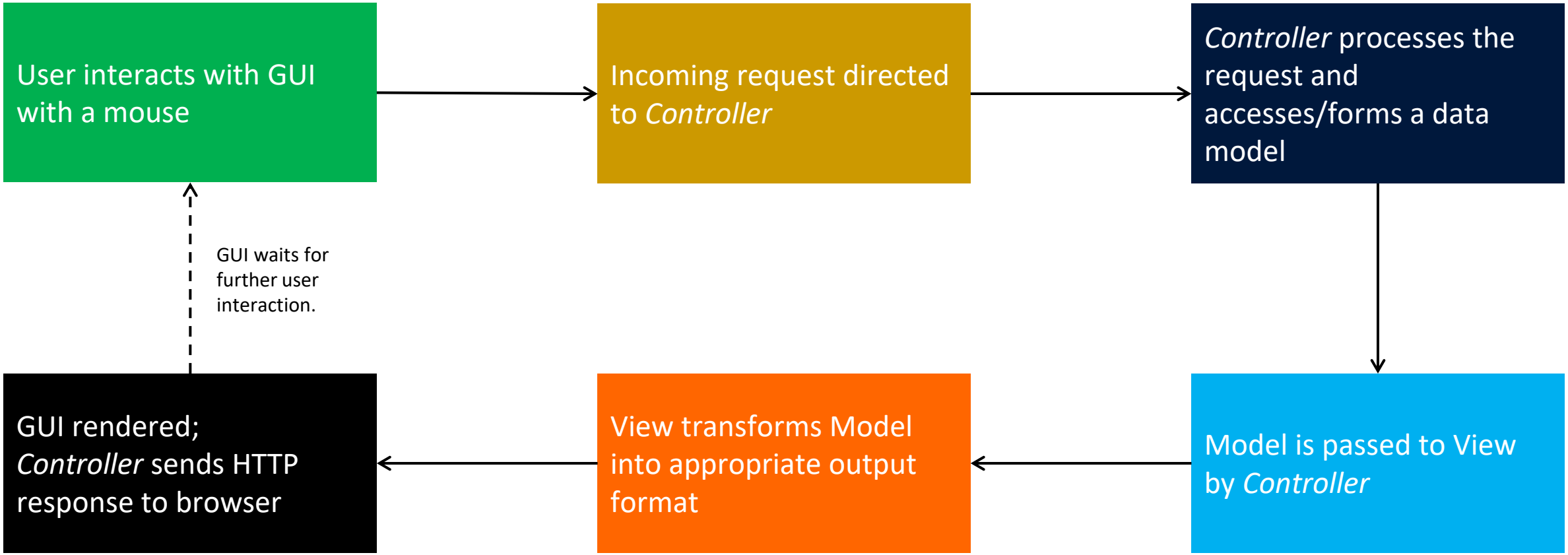
Section 1: MVC Design Pattern

Lesson: Overview

Model View Controller (MVC) Design Pattern



MVC Control Flow



Module 2: Models

Section 2: Model Fundamentals

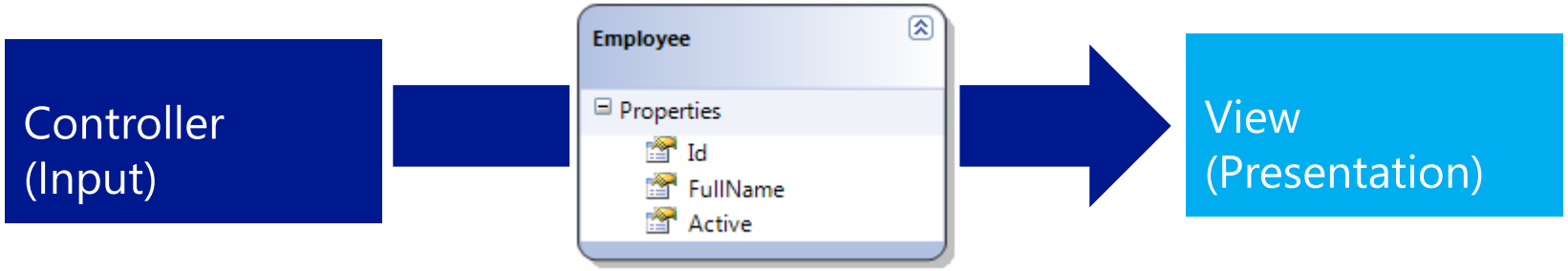
Lesson: Role of Models

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)

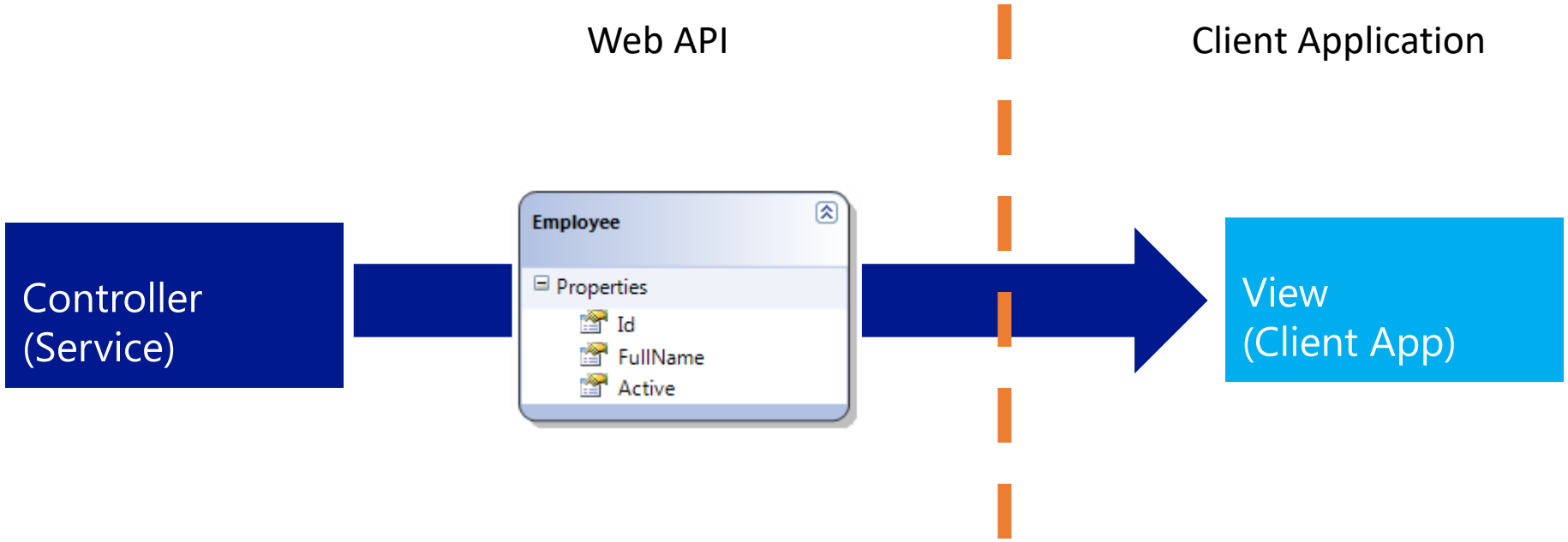
Role of a Model

- 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!



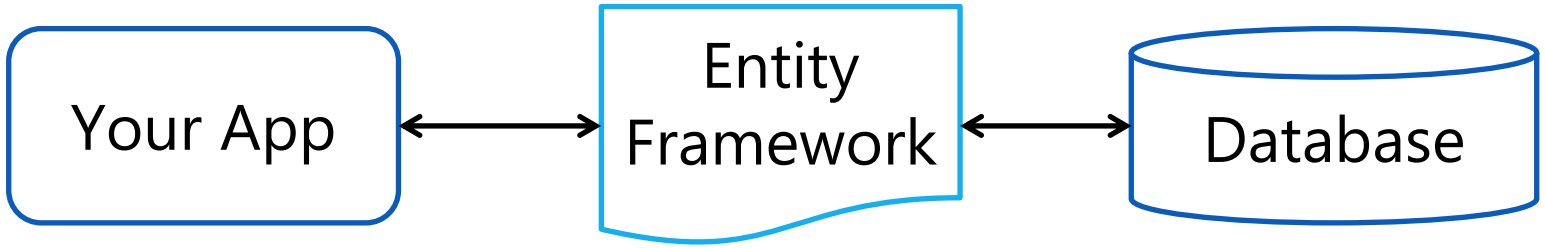
Module 2: Models

Section 3: Model Development

Lesson: Development with Entity Framework

Entity Framework Core

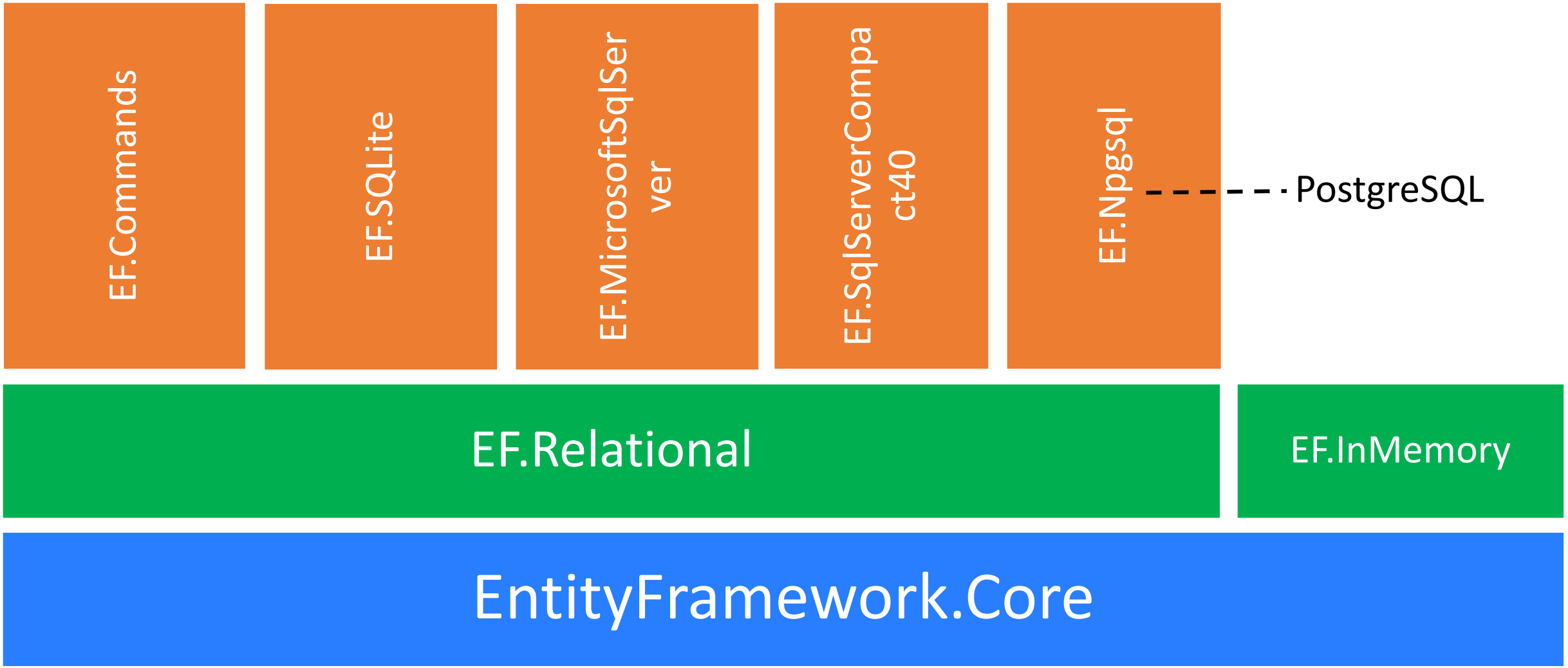
- Object-relational mapping framework by Microsoft
  - It understands how to store .NET objects in a **relational** database.
  - It retrieves and manipulates data as strongly typed objects using LINQ query
- It provides:
  - Change tracking
  - Identity resolution
  - Dev-time tooling
  - Query translation
  - More!



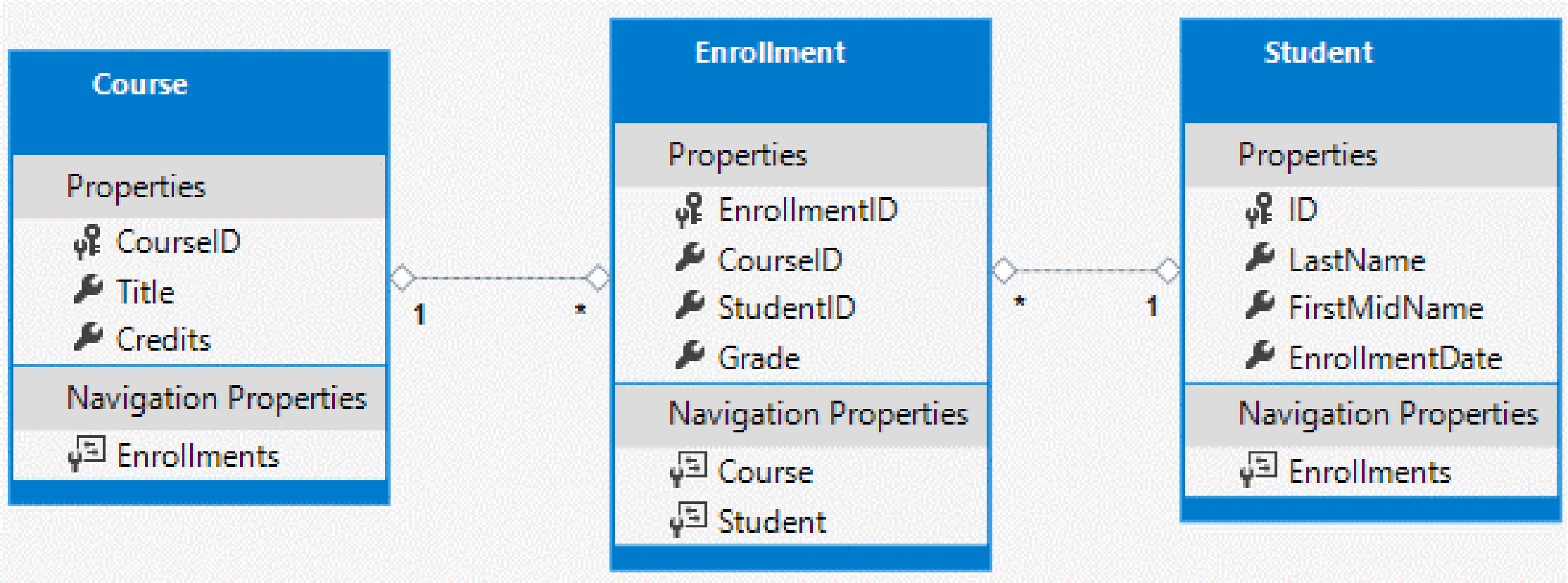
- Open-source and Cross-platform!
- Both Entity Framework v6 and Core will continue to develop separately

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.

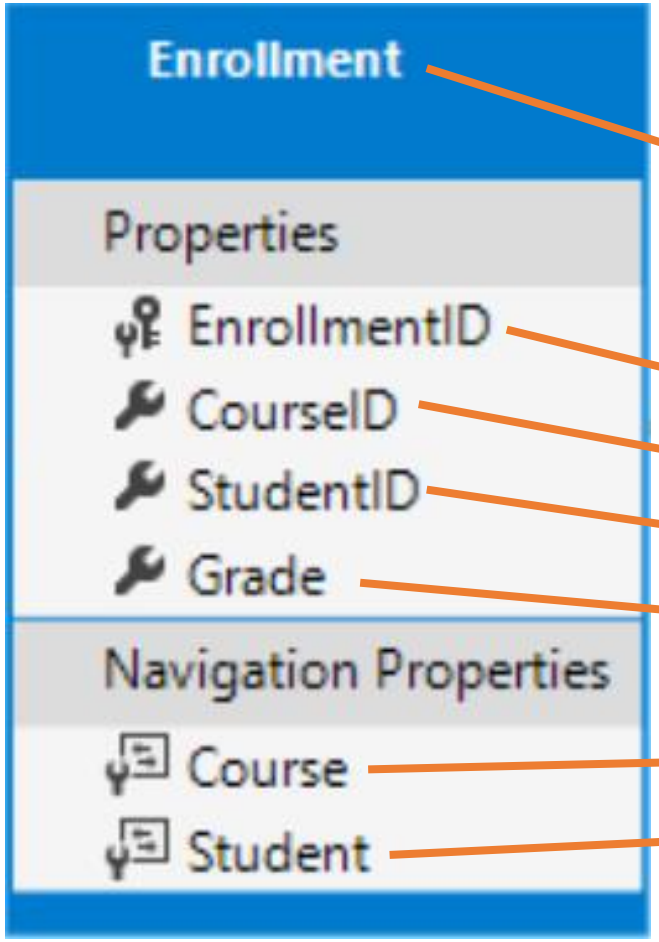
- Foreign key property
  - It is not required in a model object
  - It is used for convenience
  - *CourseID* and *StudentID* are foreign key properties

```
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




```
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; }
    }
}
```

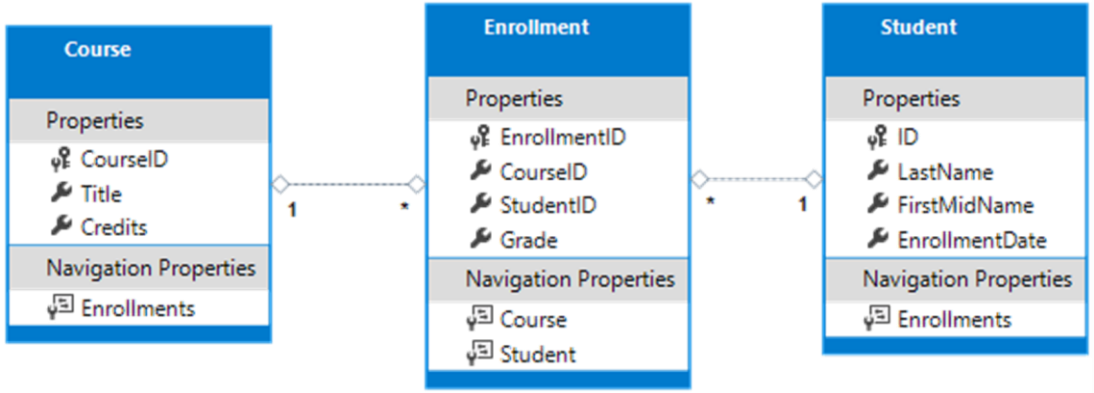
Model Relationships - DbContext

- 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

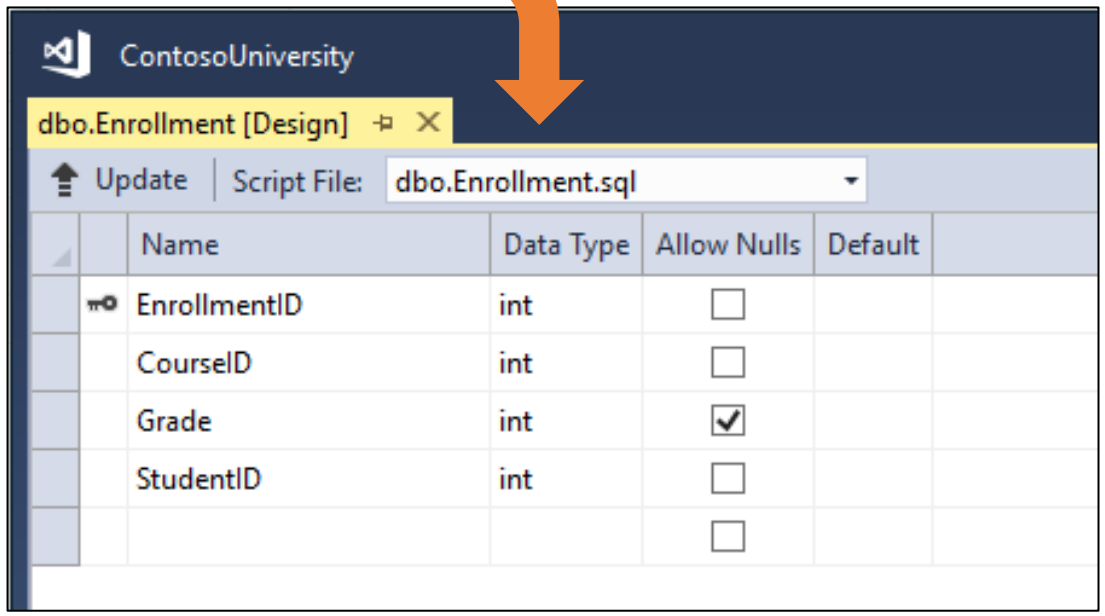
- Enables `dotnet ef *` commands at the command line in the project directory, e.g.,
  - `dotnet ef mgirations add Initial`
  - `dotnet ef database update Initial`
  - `dotnet ef dbcontext scaffold …`

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; }
    }
}
```



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", SqlServerValueGeneration
            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

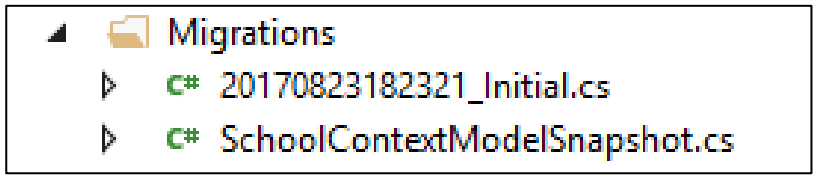
```
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Enrollment");

    migrationBuilder.DropTable(
        name: "Course");

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

Creating and Applying Migrations

```
$ dotnet ef migrations add Initial
```

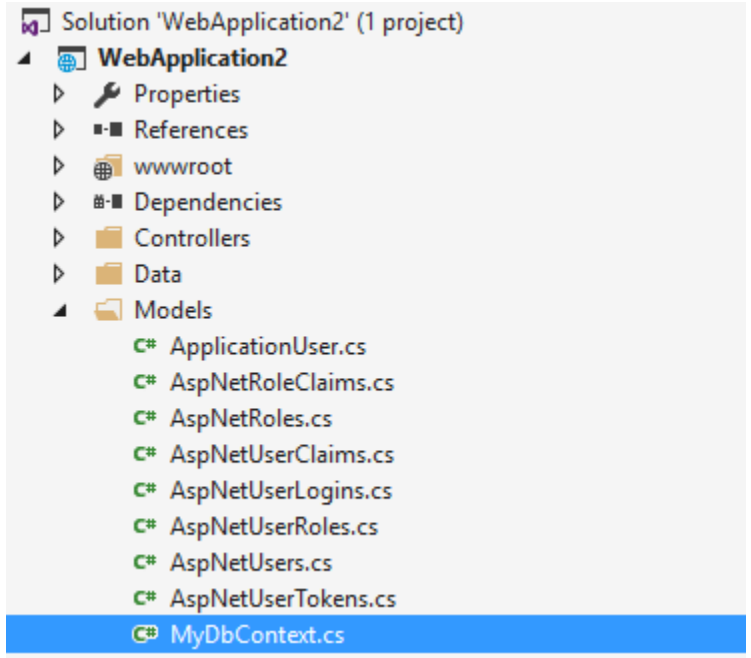



```
$ dotnet ef database update Initial
```

```
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.d
atabase.windows.net;Initial Catalog=mydbcatalog;Password=P@ssw0rd!w;User ID=igorsych@mydb" "Microsoft.EntityFrameworkCor
e.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

```
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; }
}
```

- 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

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 you 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)
{

}
```

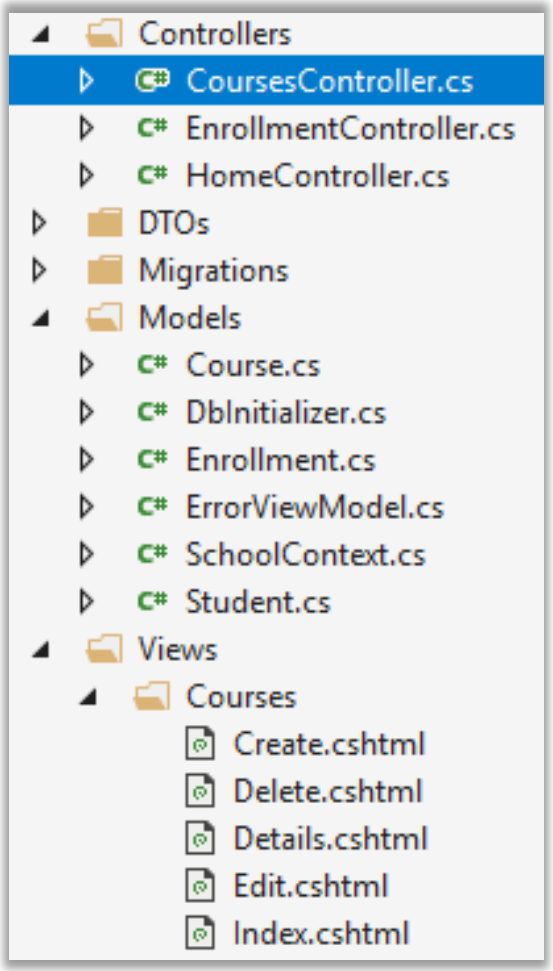
Module 2: Models

Section 4: Model Design

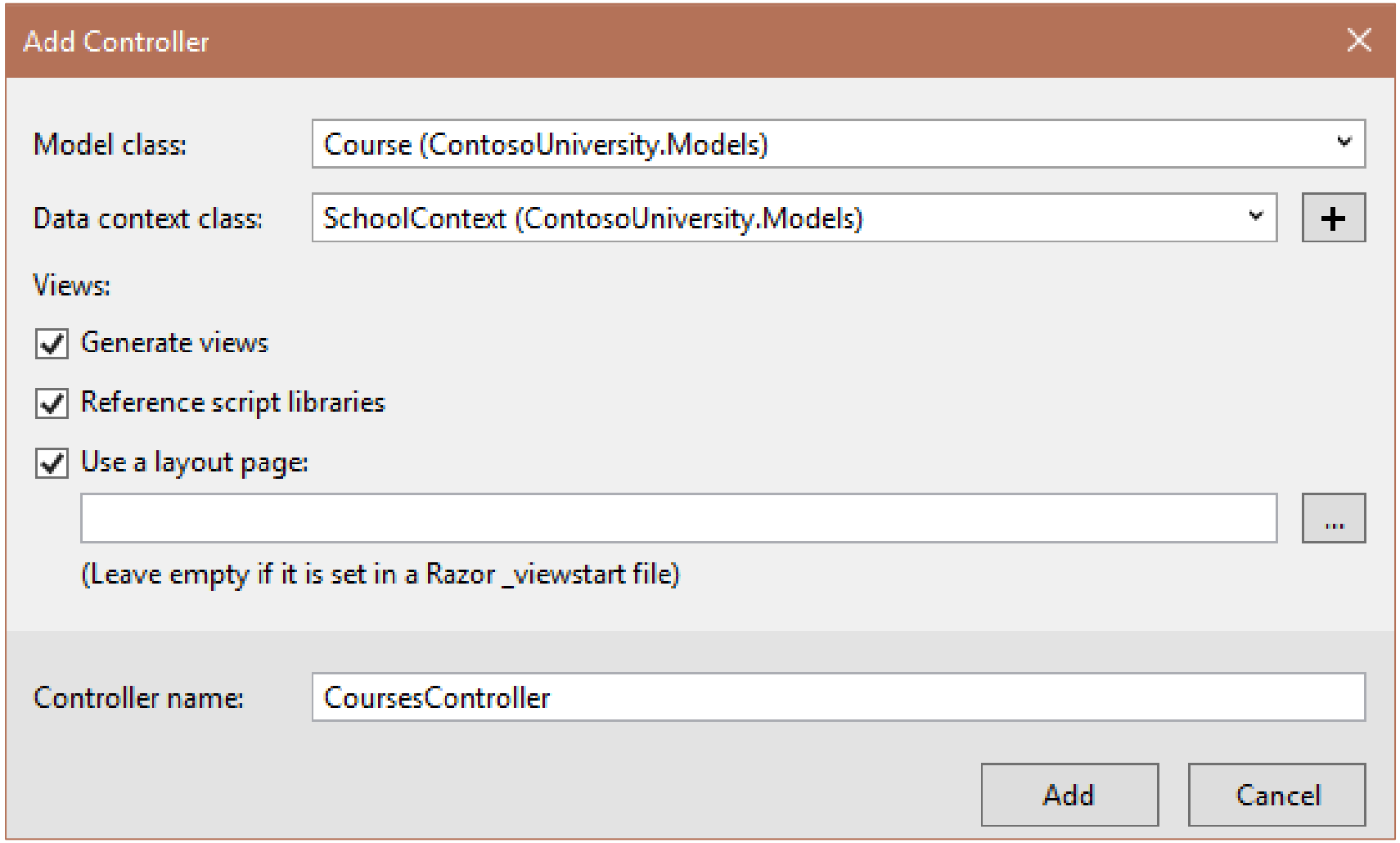
Lesson: Scaffolding

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

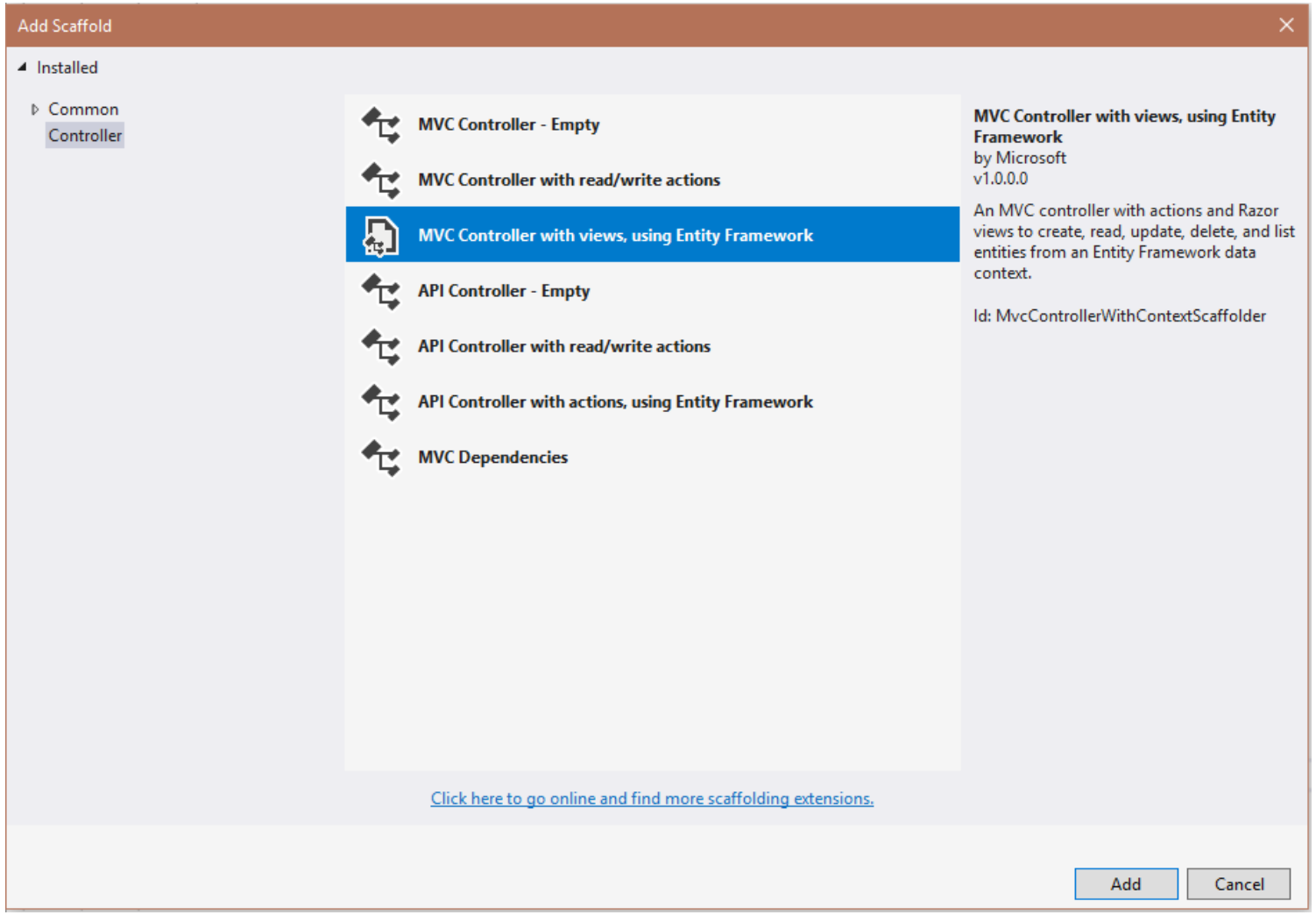


ASP.NET MVC Scaffolding in Visual Studio



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

Microsoft