

.NET Framework: Developing Modern Web Apps with ASP.NET MVC – Workshop*PLUS*

Wael Kdoh

Senior Consultant

v1.0

Module 2: Models

Module Overview

Module 2: Models

Section 1: Model Fundamentals

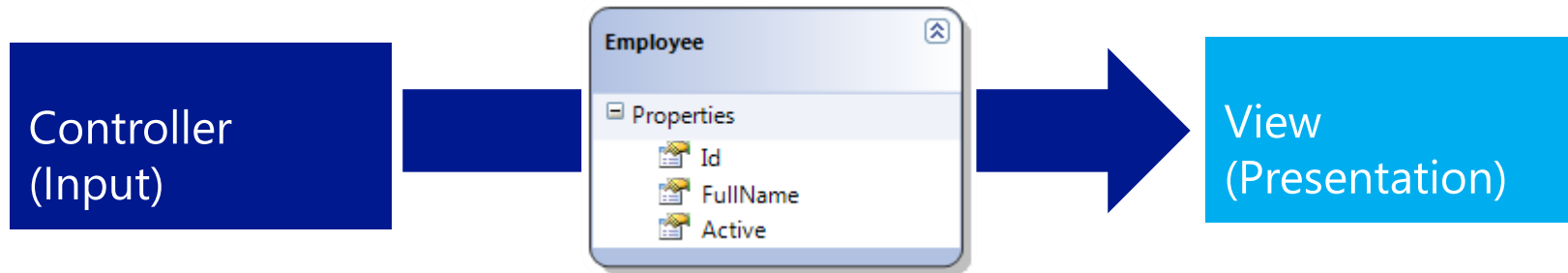
Lesson: Role of Models

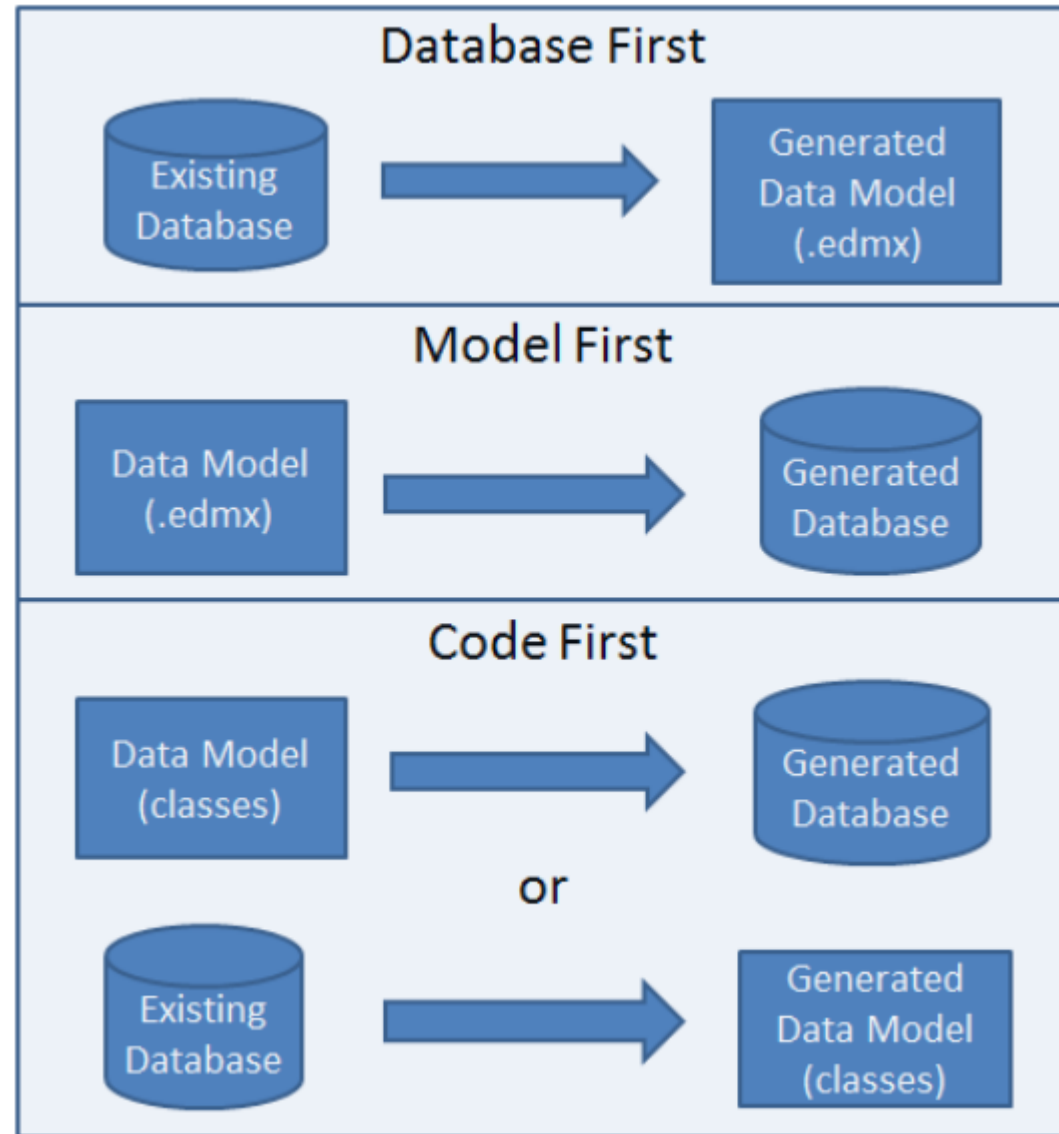
Model

- It is a set of .NET classes that:
 - Describes data that the application is working with
 - Implements the **business 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
 - WS-* web services
- It can use any data access technology for accessing and manipulating data
 - Entity Framework (EF) for example.
 - ADO.NET

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





Module 2: Models

Section 2: Model Development

Lesson: Development with Entity Framework

Model Development

- A model can be created with a .NET class
- Private 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 class Enrollment {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public decimal? Grade { get; set; }
        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```

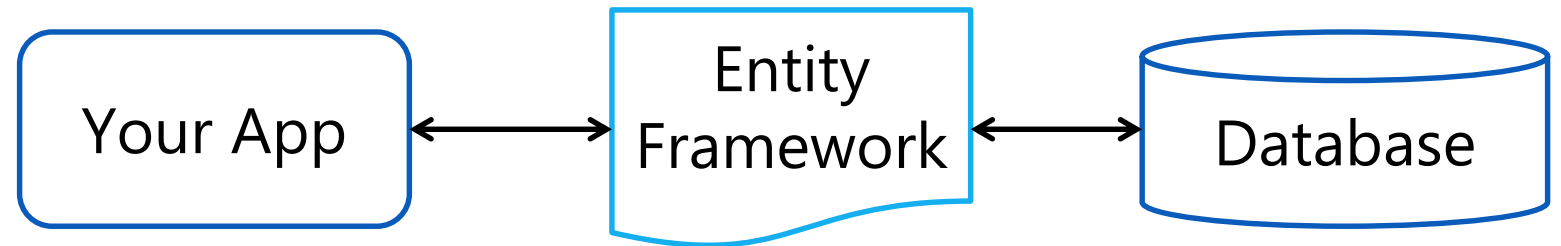

Model Relationships

- Navigation property
 - Navigational property holds other entities (for example, *Courses*) that are related to this entity (for example, *Enrollment*).
 - Navigation properties are defined as virtual to take advantage of Entity Framework lazy loading
 - *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 class Enrollment {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public decimal? Grade { get; set; }
        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```

Entity Framework

- Object-relational mapping framework by Microsoft
 - Enables .NET developers to work with relational data using domain-specific objects.
 - Understands how to store .NET objects in a relational database.
 - Retrieves and manipulates data as strongly typed objects using LINQ query.
 - Eliminates the need for most of the data-access code that developers usually need to write.
- It provides:
 - Change tracking
 - Identity resolution
 - Lazy loading, etc.
- It is open sourced at <https://github.com/aspnet/EntityFramework6>



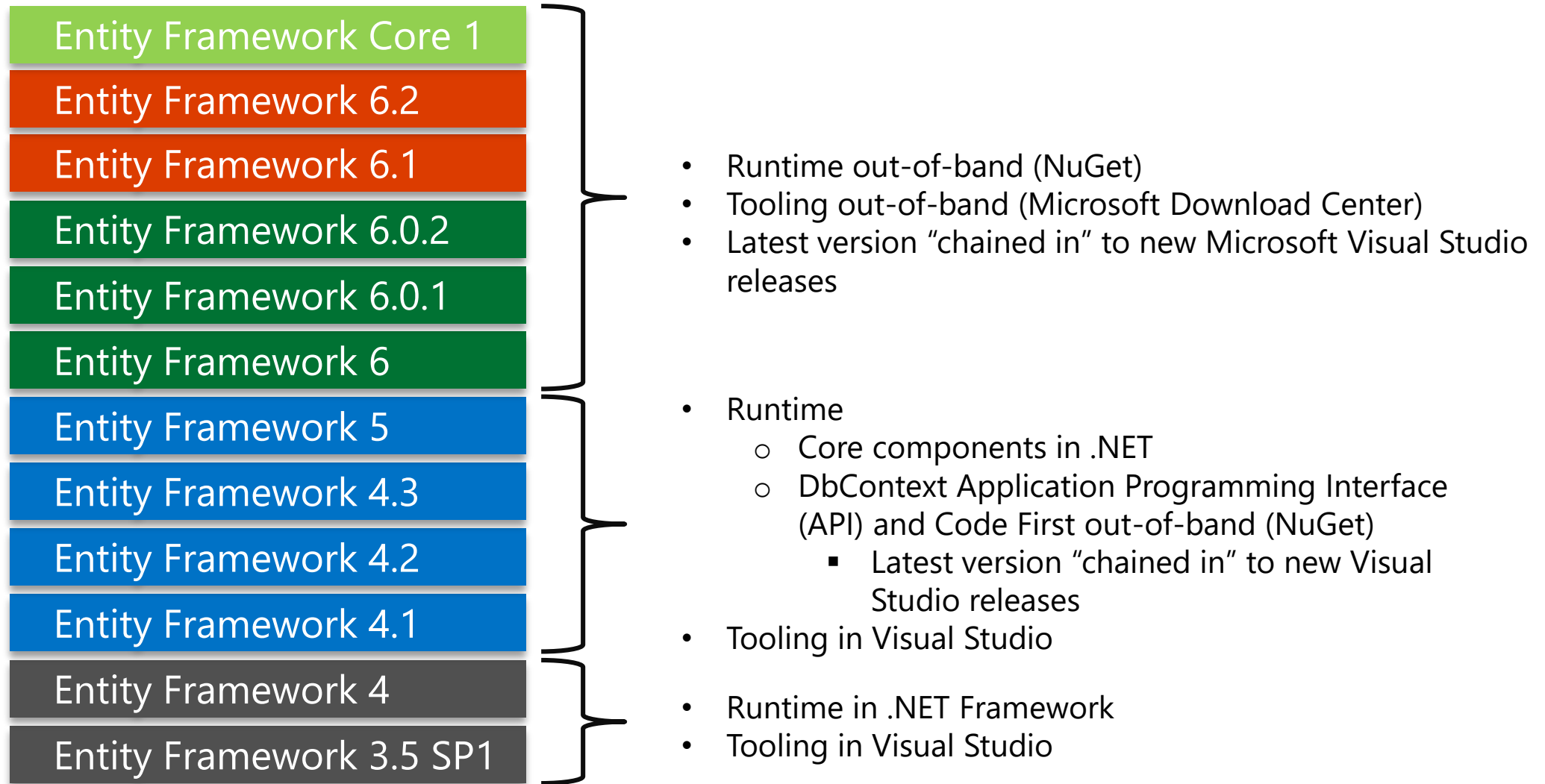
Note: It is not mandatory to be used with Model View Controller (MVC)

Entity Framework and ADO.NET

- Entity Framework built on top of ADO.NET since first Entity Framework version
- Entity framework generates T-SQL and then passes it to SQL Server via ADO.NET
- ADO.NET code is faster at runtime, because it doesn't need to generate T-SQL
- Entity Framework code saves developers time

Demo: ADO.NET

Out of Band | Release History



Entity Framework Model Approaches

Database First (EDMX based)

Model First (EDMX based)

Code First (Blank or from existing db)

Database First Approach

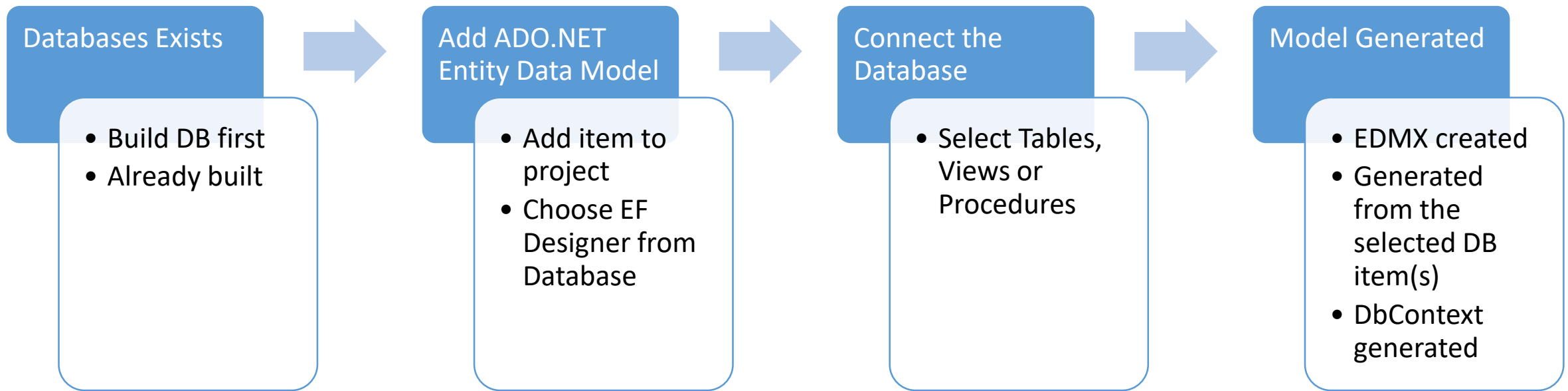
- Pros

- This approach is very popular when we have an existing database.
- The wizard creates automatically the entities and all of the relationships.
- We need to write less code and put less effort into creating the entity model
- Easy update if any changes are made to the database

- Cons

- Very difficult to maintain a large and complex model/database.
- To get a better control the change should come from the database and not from the model.

Database First Diagram



Demo: Database First

Model First Approach

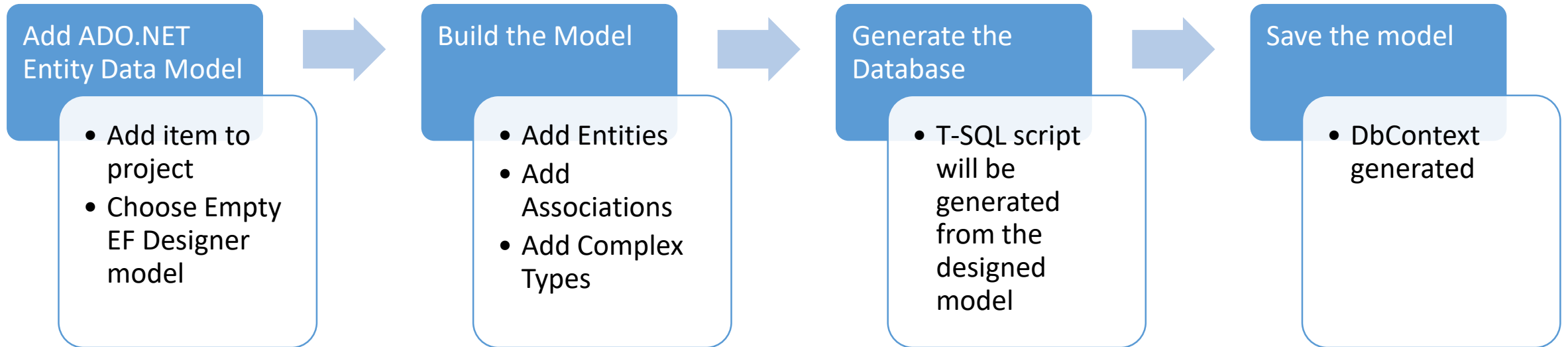
- Pros

- Easy to design and create a database schema in the EF Designer.
- Easily update the model diagram from the database.
- Entity Framework generates the T-SQL for the database schema.

- Cons

- When you change the model and generate the SQL, the script will drop the tables and you will lose the existing data.
- Requires a good knowledge of Entity Framework to update the model and database.

Model-First Model



Demo: Model First

Module 2: Models

Section 2: Model Development

Lesson: Code-based Modeling

Entity Framework Development Approaches



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 are used with Entity Framework without an .edmx mapping file
- Code First can also work with existing database
 - Code is used for mapping instead of visual designer and XML

Code-First Approach: Pros and Cons

- Pros

- Can realize complicated domain requirements
- Can have a very clean and elegant domain model that is represented with Plain Old CLR Object (POCO)
- Not tied with Entity Framework

- Cons

- Needs more effort in creating domain classes
- Needs to map the domain model to data model (with Data Annotations)

Code-based Modeling vs. EDMX-based Database/Model First

- Source control merging, conflicts, and code reviews
 - Easier with C#/VB code than XML files
- Developers know how to write and debug code
 - Edmx XML is hard to debug and understand
- Ability to customize
 - Manipulating XML-based model is hard
- Code-based modeling is less-repetitive
 - Convention-based code
- Data Migrations
 - Incremental SQL script for Database schema updates without having to recreate product database

Database Creation Using Entity Framework - I

- EF 4.1 or later version supports code-first style of development:
 - Model is created using C# or VB classes
 - Creation of database schema or opening of Visual Studio designer is *not* required

```
namespace ContosoUniversity.Models
{
    public class Enrollment {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public decimal? Grade { get; set; }
        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```




Diagram illustrating the mapping from the C# code to the database schema. The code defines an `Enrollment` class with properties `EnrollmentID`, `CourseID`, `StudentID`, and `Grade`, along with virtual references to `Course` and `Student`. The corresponding database table structure is shown in the 'Edit Table - Enrollment' window.

Column Name	Data Type	Length	Allow Nulls	Unique	Primary Key
EnrollmentID	int	4	No	No	Yes
CourseID	int	4	No	No	No
StudentID	int	4	No	No	No
Grade	numeric	9	Yes	No	No

Database Creation Using Entity Framework - II

- Database Context class is a gateway to the database
- Context class needs to inherit from EF's DbContext class
- All object types are registered as DbSet<T>
 - T is the object type that needs to be persisted in the database

```
namespace ContosoUniversity.Models
{
    public class SchoolContext : DbContext
    {
        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }
    }
}
```

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

Database Seeding

- Database Initializers are deprecated in EF Core
 - CreateDatabaseIfNotExists
 - DropCreateDatabaseIfModelChanges
 - DropCreateDatabaseAlways

Configure method in Startup.cs

Seed Method

```
namespace CodeFirstDemo.Migrations
{
    using Models;
    using System.Data.Entity.Migrations;

    1 reference | 0 changes | 0 authors, 0 changes
    internal sealed class Configuration : DbMigrationsConfiguration<MyContext>
    {
        0 references | 0 changes | 0 authors, 0 changes
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
            ContextKey = "CodeFirstDemo.Models.MyContext";
        }

        0 references | 0 changes | 0 authors, 0 changes
        protected override void Seed(MyContext context)
        {
            // This method will be called after migrating to the latest version.

            // You can use the DbSet<T>.AddOrUpdate() helper extension method
            // to avoid creating duplicate seed data. E.g.
            //
            context.People.AddOrUpdate(
                p => p.Name,
                new Person { Name = "Andrew Peters" },
                new Person { Name = "Brice Lambson" },
                new Person { Name = "Rowan Miller" }
            );
        }
    }
}
```

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

Migration Methods

Up Method

```
public partial class CreateIdentitySchema : Migration
{
    0 references
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "AspNetRoles",
            columns: table => new
            {
                Id = table.Column<string>(nullable: false),
                ConcurrencyStamp = table.Column<string>(nullable: true),
                Name = table.Column<string>(nullable: true),
                NormalizedName = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_IdentityRole", x => x.Id);
            });
        migrationBuilder.CreateTable(
            name: "AspNetUsers",
```

Down Method

```
0 references
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable("AspNetRoleClaims");
    migrationBuilder.DropTable("AspNetUserClaims");
    migrationBuilder.DropTable("AspNetUserLogins");
    migrationBuilder.DropTable("AspNetUserRoles");
    migrationBuilder.DropTable("AspNetRoles");
    migrationBuilder.DropTable("AspNetUsers");
}
```

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

Module 2: Models

Section 3: 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
 - 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; }
}
```

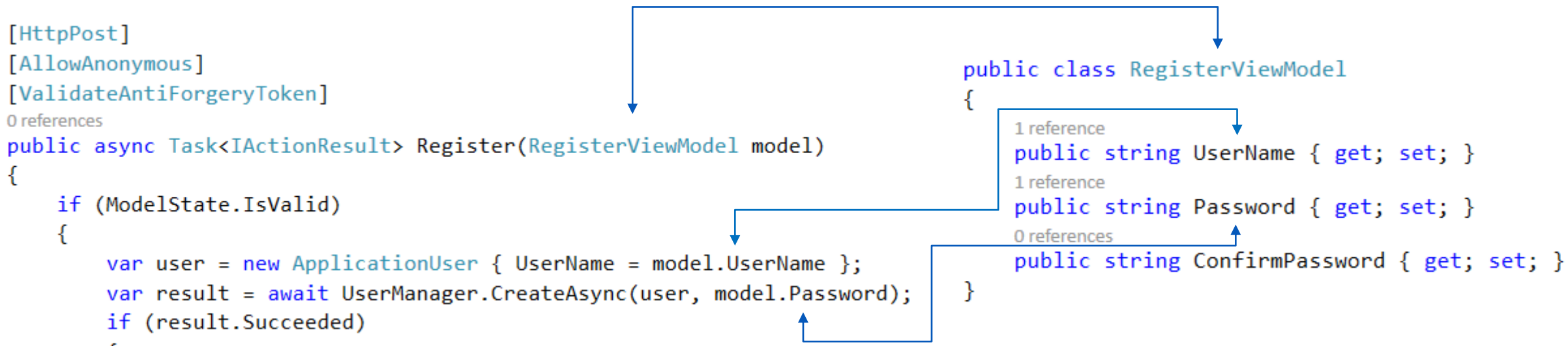
Module 2: Models

Section 3: 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



Model Development Best Practices

- Strive for fat models and skinny controllers
- Eager loading versus Lazy loading strategy
 - **Eager loading:** Loads all data using a single query

```
public ActionResult Index()
{
    var enrollments = db.Enrollments.Include(e => e.Course).Include(e => e.Student);
    return View(enrollments.ToList());
}
```

- **Lazy loading:** Loads data on as-needed basis

```
public ActionResult Index()
{
    var enrollments = db.Enrollments;
    return View(enrollments.ToList());
}
```


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 best practices



Lab: Models



