Introduction to Entity Framework

Introduction to ORM

- If you are a relative newcomer to the programming world, terms like Object-Relational-Mapper (ORM) can sound really intimidating.
- The nice part about ORMs, is that they make it easier to write code once you get the hang of them.
- Before we talk about what an Object-Relational-Mapper is, it might be better to talk about Object-Relational-Mapping as a concept first.

Introduction to ORM

- Usually, a SQL query looks like this:

```sql
SELECT * FROM users WHERE email = "test@test.com";
```

- Object-Relational-Mapping is the idea of being able to write queries like this, as well as much more complicated ones, using your preferred object-oriented programming language.
- Long story short, we are trying to interact with our database using our language of choice, instead of SQL.

Introduction to ORM

- Here's where the Object-Relational-Mapper comes in.
- When most people say ORM, they are referring to a library that implements this technique.
- For example, the previous query would now look something like this:

```
using Generic.Orm.Library;

var context = new UserContext();

var user = context.Users.Where(u => u.Email == "test@test.com");
```

- As you can see, we are using an imaginary ORM library to execute the exact same query, using C# (or whatever language you love).

ORM Pros

- You get to write in the language you are already using.
  - To be honest, programmers aren't greatest at writing SQL statements.
  - SQL is a ridiculously powerful language, but most of us don't write it often.
  - We do, however, tend to be much more fluent in one language or another and being able to leverage that fluency is awesome!
- It abstracts away the database system so that switching from SQL Server to MySQL, or whatever flavor you prefer, is easy-peasy.
- Depending on the ORM, you get a lot of advanced features out of the box, such as support for transactions, connection pooling, migrations, seeds, streams, and all sorts of other goodies.
- Many of the queries you write will perform better than if you wrote them yourself.

ORM Cons

- If you are a master at SQL, you can probably get more performant queries by writing them yourself.
- There is overhead involved in learning how to use any given ORM.
- The initial configuration of an ORM can be a headache.
- As a developer, it is important to understand what is happening under the hood.
  - Since ORMs can serve as a crutch to avoid understanding databases and SQL, it can make you a weaker developer in that portion of the stack.

Popular ORMs

- Wikipedia has a great list of ORMs that exist for just about any language:
  - https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

Introduction to Entity Framework

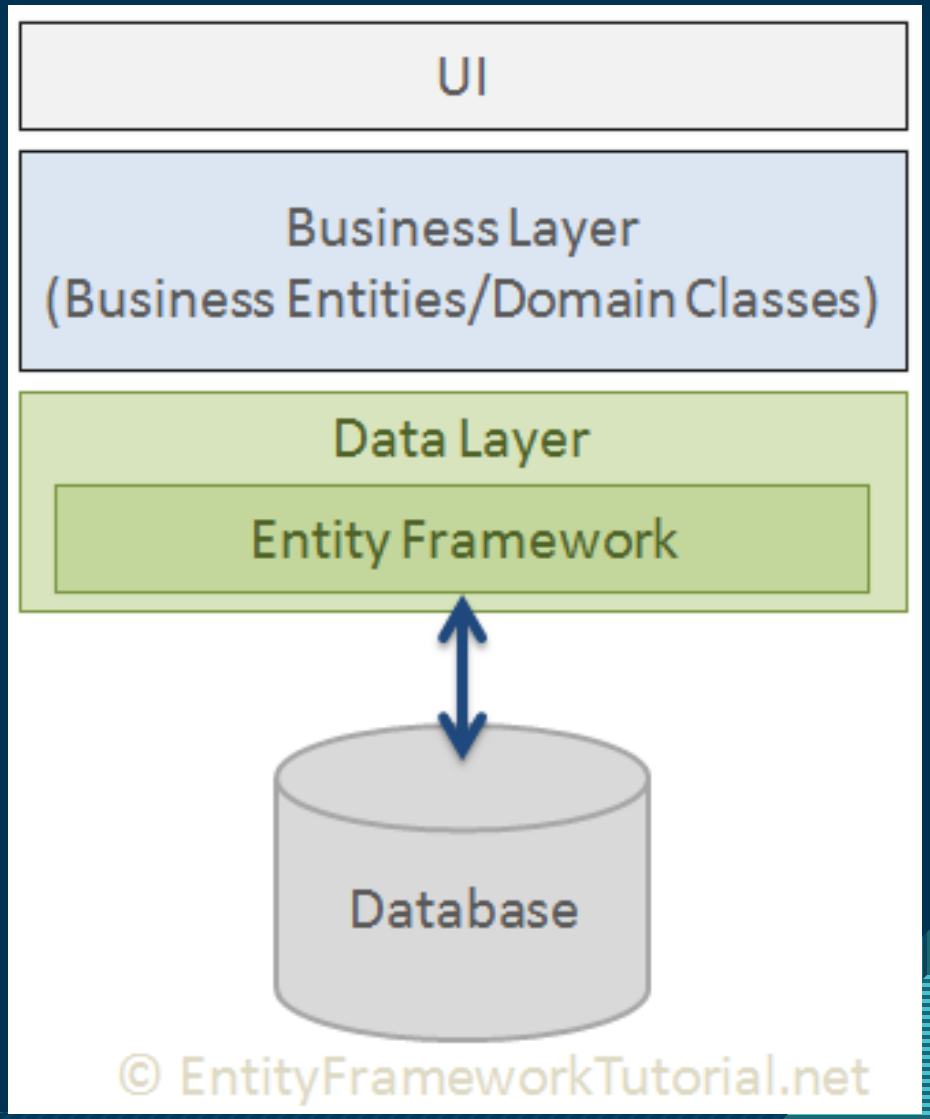
- Entity Framework (EF) was first released as part of the .NET Framework 3.5 with Service Pack 1 back in late 2008.
- Since then, Entity Framework has evolved, as Microsoft has observed how programmers use an object-relational mapping (ORM) tool in the real world.
- ORMs use a mapping definition to associate columns in tables to properties in classes.
- Then, a programmer can interact with objects of different types in a way that they are familiar with, instead of having to deal with knowing how to store the values in a relational table.

Introduction to Entity Framework

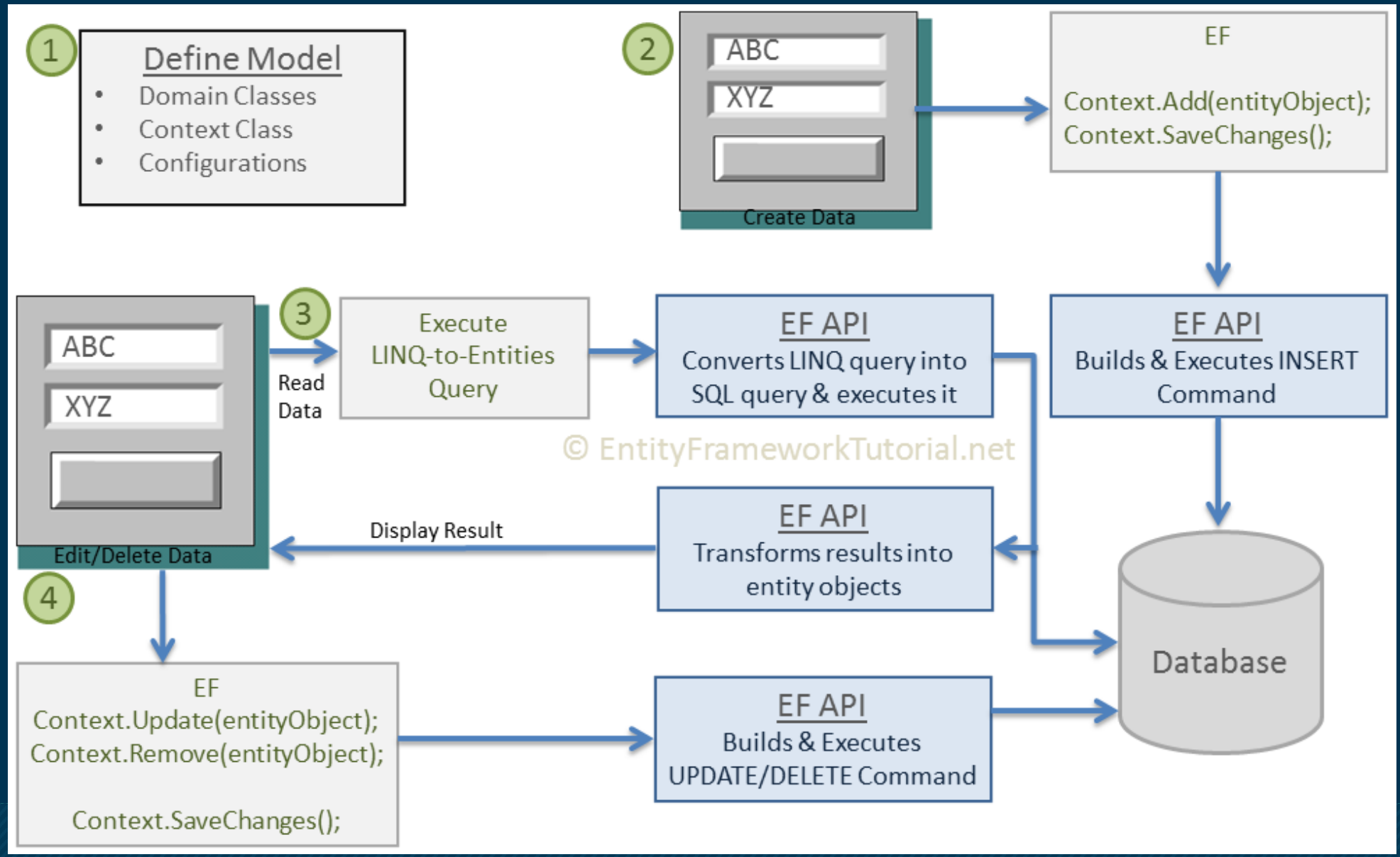
- Prior to .NET 3.5, developers often used to write ADO.NET code to save or retrieve application data from the underlying database.
- They used to:
  - open a connection to the database,
  - create a DataSet to fetch or submit the data to the database,
  - convert data from the DataSet to .NET objects or vice-versa to apply business rules.
- This was a cumbersome and error prone process.
- Entity Framework automates all these database related activities for your application.
- With the Entity Framework, developers can work at a higher level of abstraction when they deal with data and can create and maintain data-oriented applications with less code compared with traditional applications.

Introduction to Entity Framework

- Definition:
  - Entity Framework is an object-relational mapper (ORM) that enables .NET developers to work with a database using .NET objects.
  - It eliminates the need for most of the data-access code that developers usually need to write.
- Entity Framework fits between the business entities (domain classes) and the database.
- It saves data stored in the properties of business entities and also retrieves data from the database and converts it to business entities objects automatically.



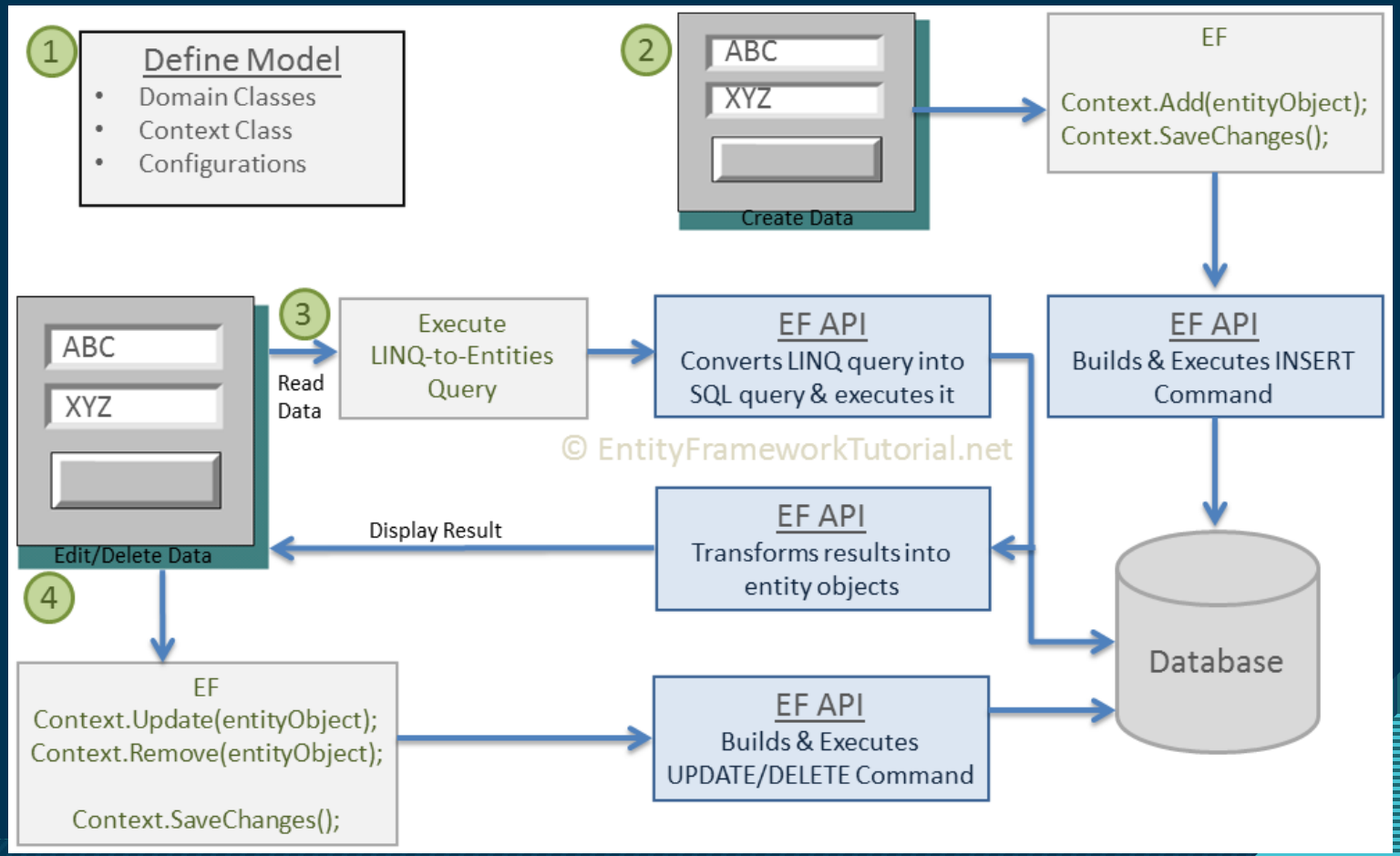
Basic Workflow in Entity Framework



Basic Workflow in Entity Framework

1. First of all, you need to define your model.

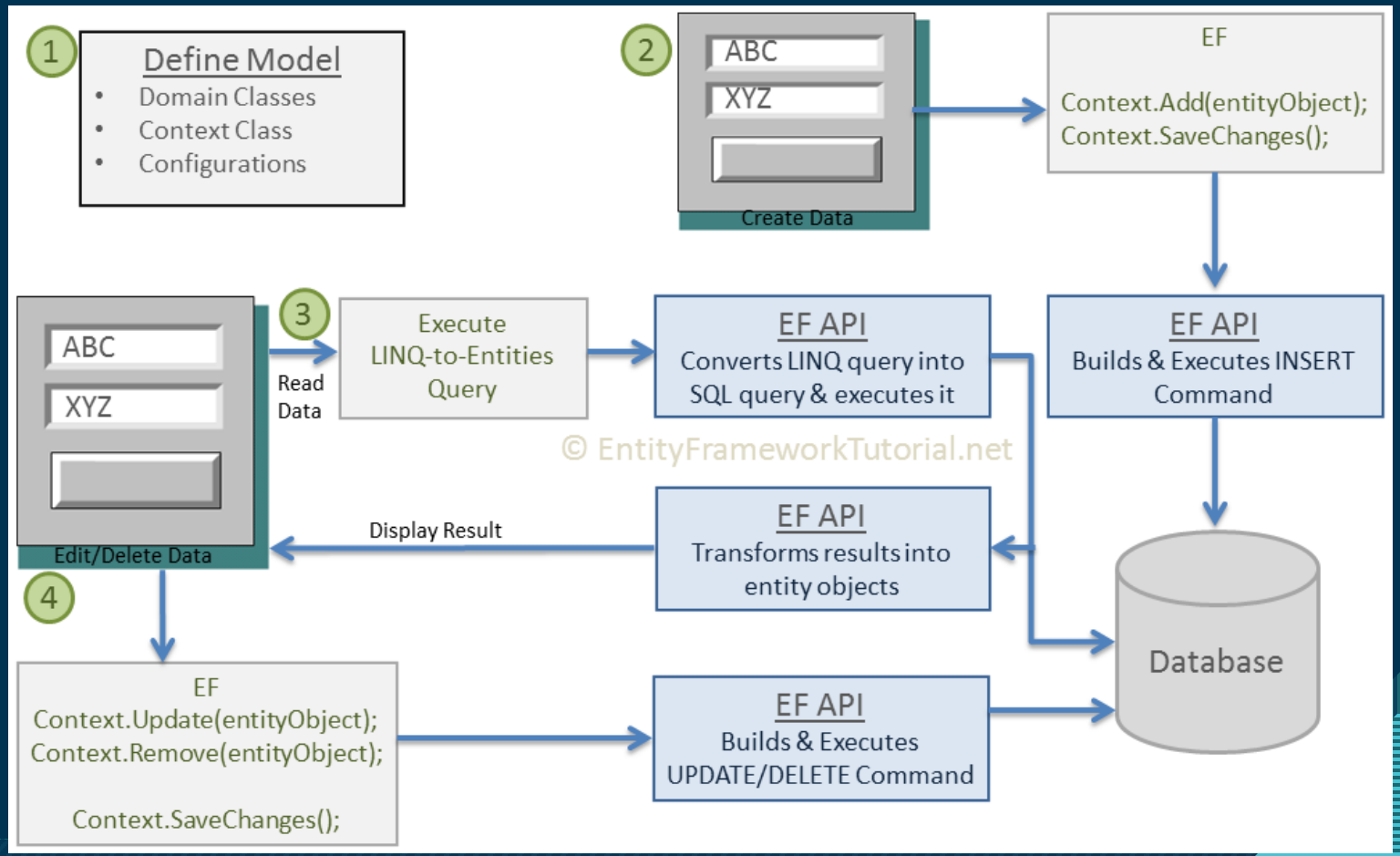
- Defining the model includes defining your domain classes, context class derived from `DbContext`, and configurations (if any).
- EF will perform CRUD operations based on your model.



Basic Workflow in Entity Framework

2. To insert data, add a domain object to a context and call the `SaveChanges()` method.

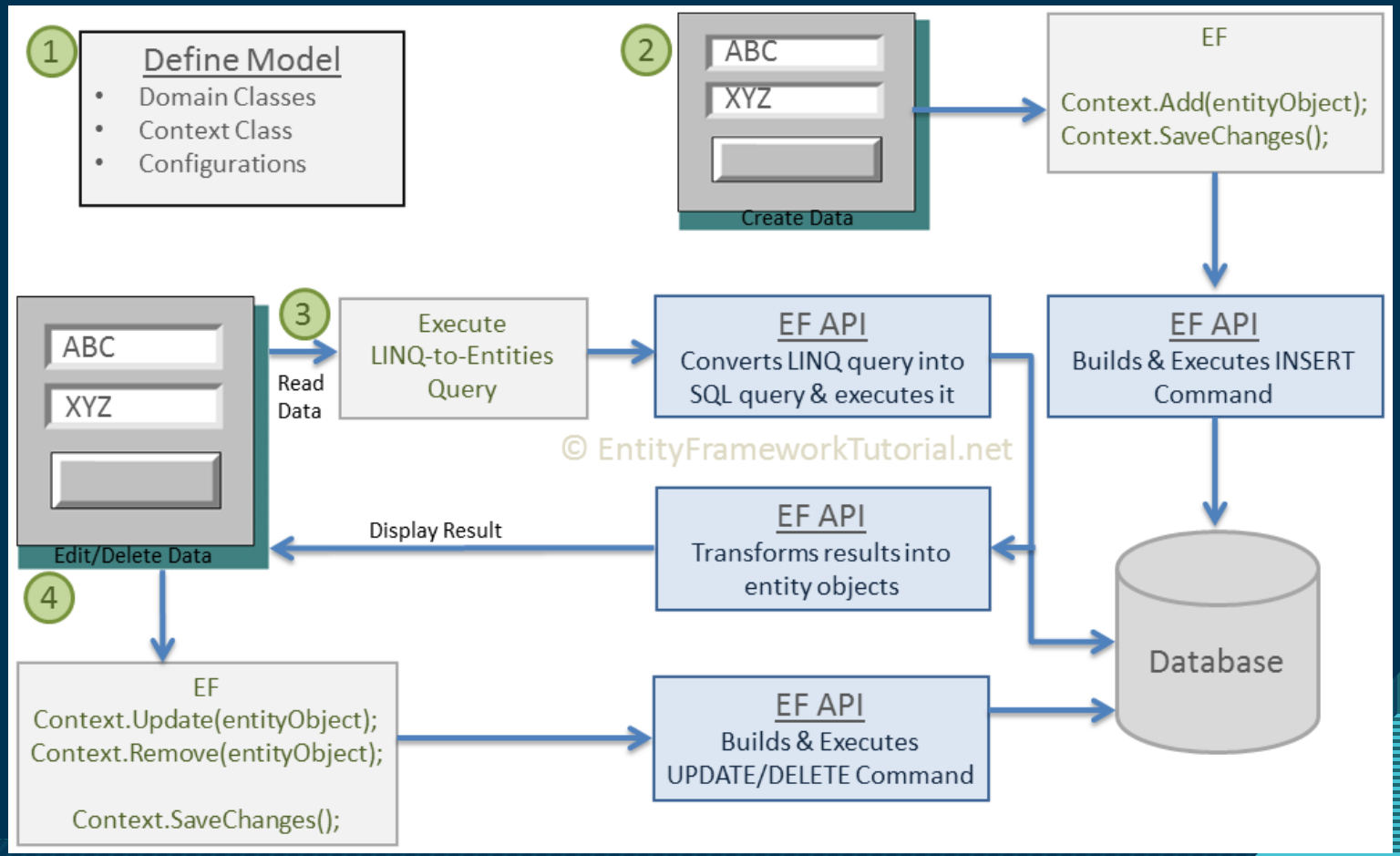
- EF will build an appropriate INSERT command and execute it to the database.



Basic Workflow in Entity Framework

3. To read data, execute the LINQ-to-Entities query in your preferred language (C#/VB.NET).

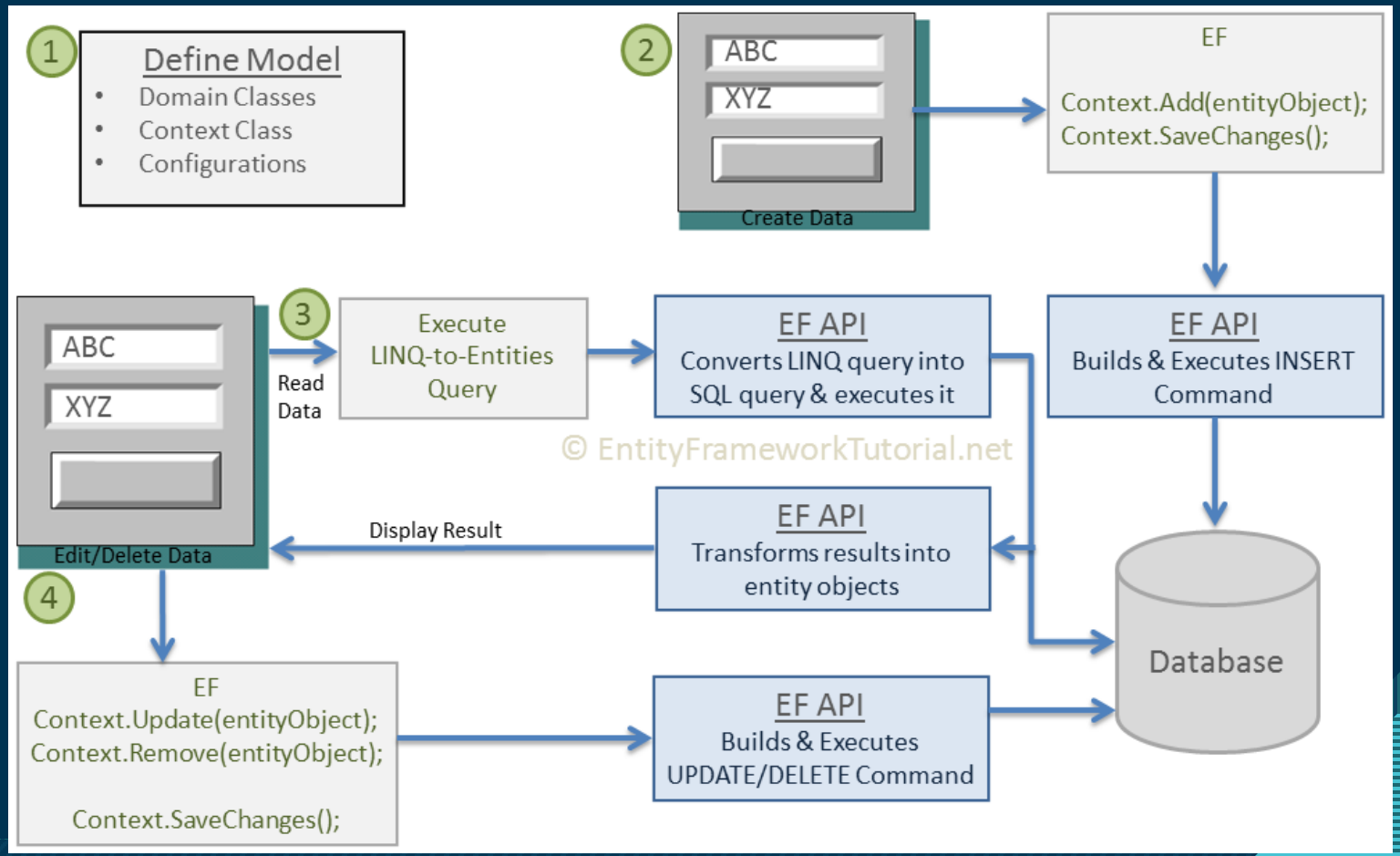
- EF will convert this query into SQL query for the underlying relational database and execute it.
- The result will be transformed into domain (entity) objects and displayed on the UI.



Basic Workflow in Entity Framework

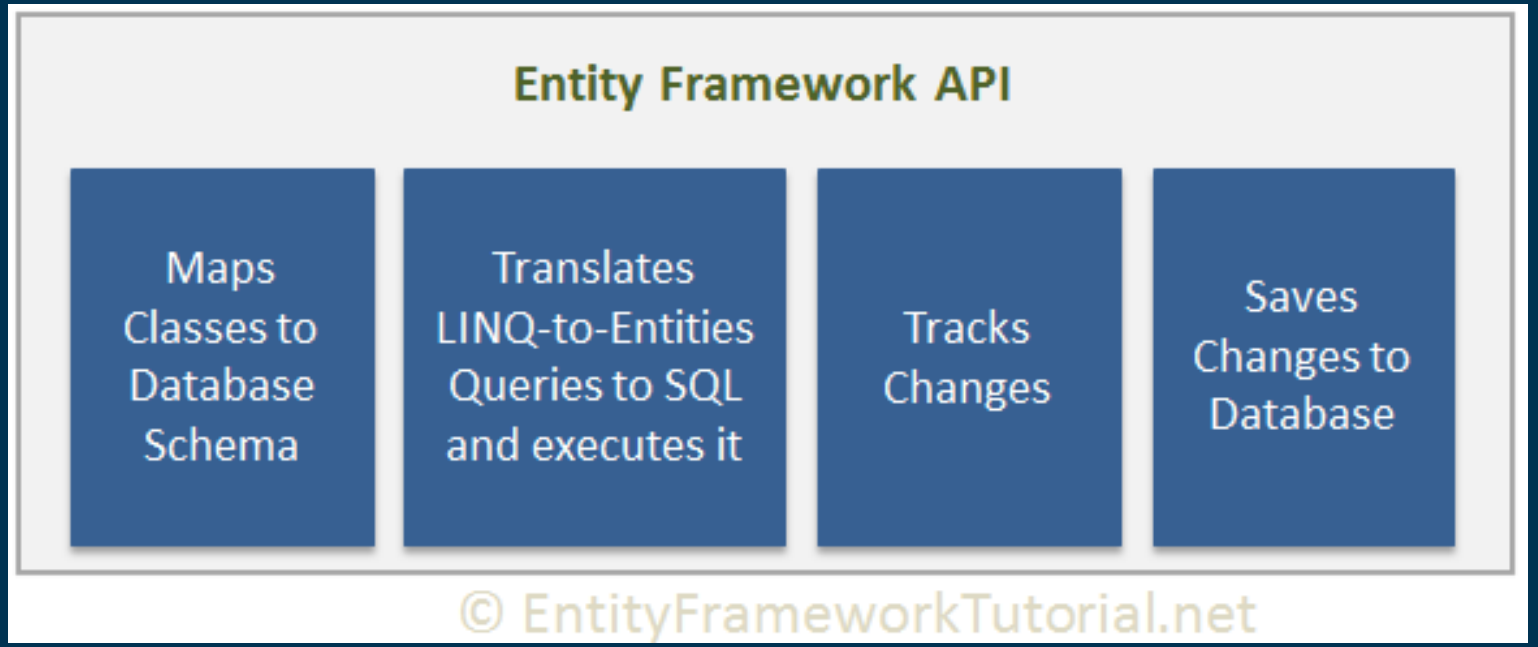
4. To edit or delete data, update or remove entity objects from a context and call the `SaveChanges()` method.

- EF will build the appropriate UPDATE or DELETE command and execute it to the database.



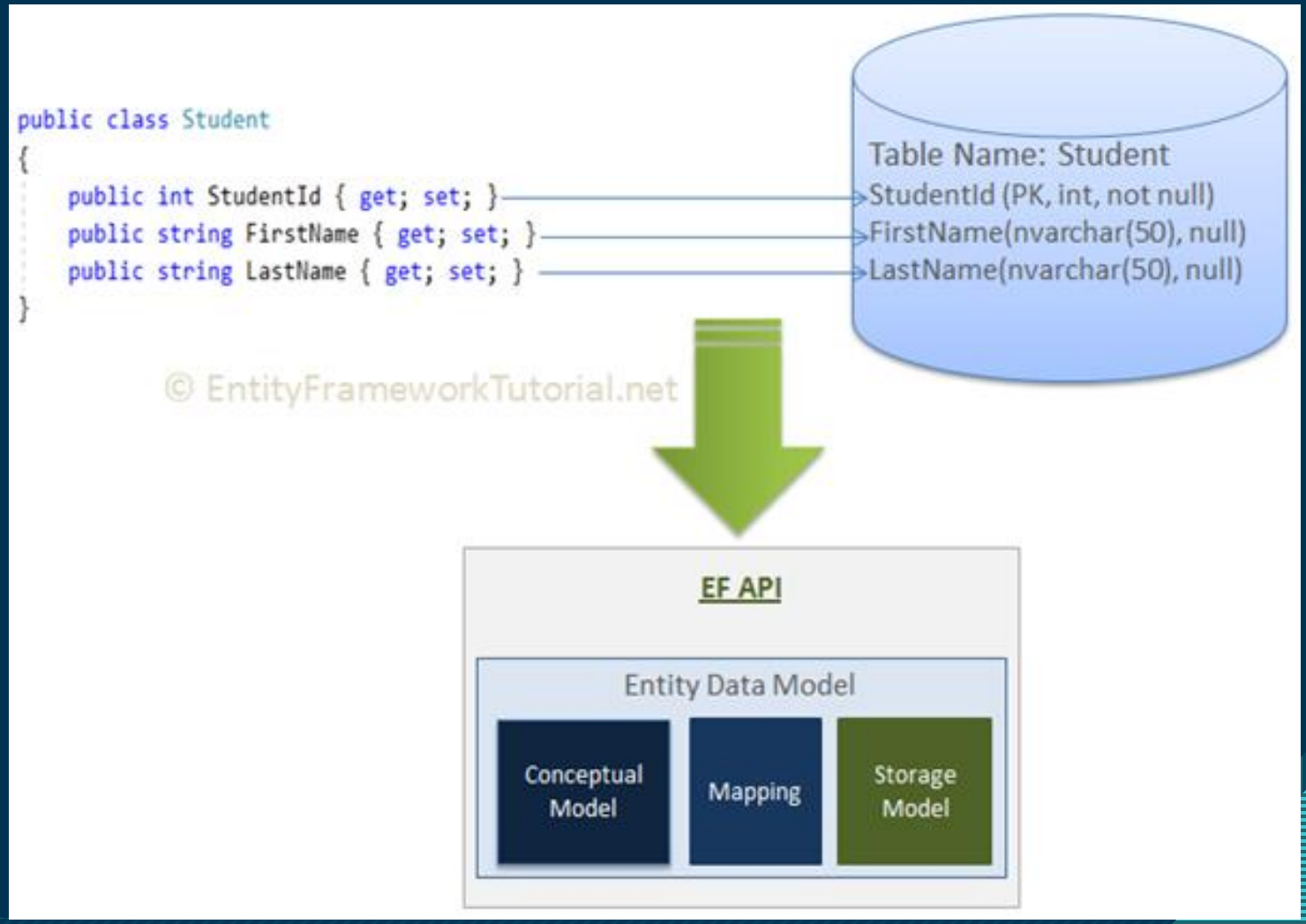
How Entity Framework Works?

- Entity Framework API (EF6 & EF Core) includes:
  - the ability to map domain (entity) classes to the database schema,
  - translate & execute LINQ queries to SQL,
  - track changes occurred on entities during their lifetime,
  - and save changes to the database.



Entity Data Model (EDM)

- The very first task of EF API is to build an Entity Data Model (EDM).
- EDM is an in-memory representation of the entire metadata:
  - conceptual model,
  - storage model,
  - and mapping between them.



Entity Data Model (EDM)

- **Conceptual Model:**
  - The conceptual model contains the model classes and their relationships.
  - This will be independent from your database table design.

- **Storage Model:**
  - EF builds the storage model for the underlying database schema.
  - The storage model is the database design model which includes tables, views, stored procedures, and their relationships and keys.

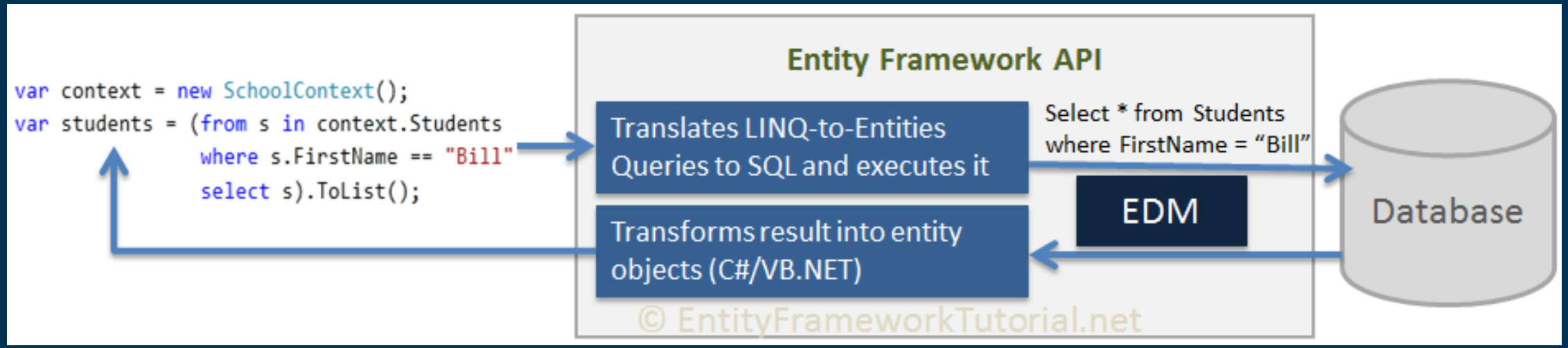
- **Mappings:**
  - Mapping consists of information about how the conceptual model is mapped to the storage model.

- EF performs CRUD operations using this EDM.
- It uses EDM in building SQL queries from LINQ queries, building INSERT, UPDATE, and DELETE commands, and transform database result into entity objects.

Entity Data Model (EDM)

Querying:

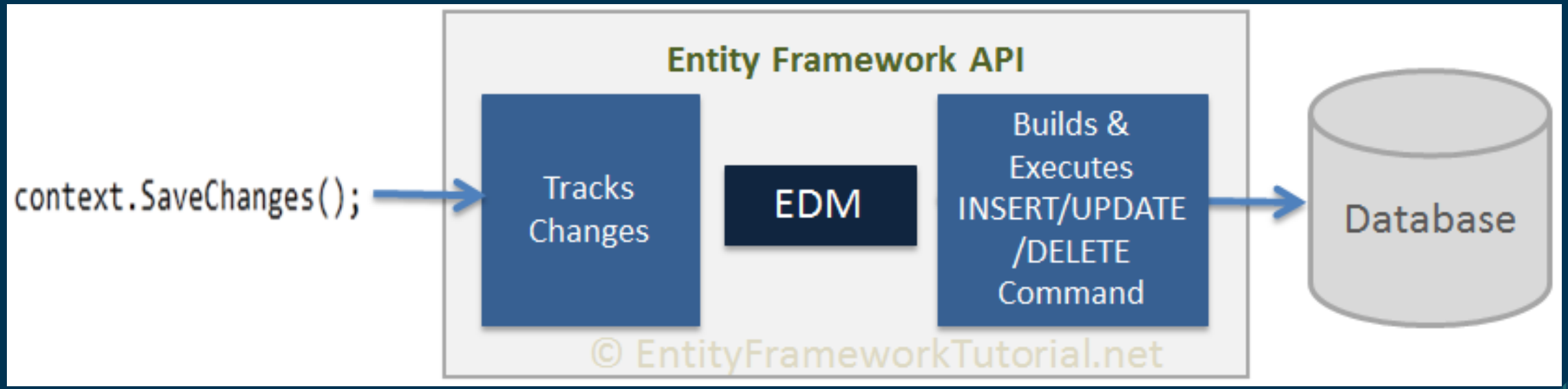
- EF API translates LINQ-to-Entities queries to SQL queries for relational databases using EDM and also converts results back to entity objects.



Entity Data Model (EDM)

Saving:

- EF API infers INSERT, UPDATE, and DELETE commands based on the state of entities when the `SaveChanges()` method is called.
- The `ChangeTrack` keeps track of the states of each entity as and when an action is performed.



Context Class in Entity Framework

- The context class is a most important class while working with EF.
- It represent a session with the underlying database using which you can perform CRUD (Create, Read, Update, Delete) operations.
- It is a class which derives from `System.Data.Entity.DbContextDbContext`.
- An instance of the context class represents Unit Of Work and Repository patterns wherein it can combine multiple changes under a single database transaction.
- It is used to query or save data to the database.
- It is also used to configure domain classes, database related mappings, change tracking settings, caching, transaction etc.

Context Class in Entity Framework

- The following `SchoolContext` class is an example of a context class.

```
using System.Data.Entity;

public class SchoolContext : DbContext
{
    // Constructor
    public SchoolContext()
    {

    }

    // Entities
    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```

What is an Entity in Entity Framework?

- An entity in Entity Framework is a class that maps to a database table.
- This class must be included as a `DbSet<TEntity>` type property in the `DbContext` class.
- EF maps each entity to a table and each property of an entity to a column in the database.
- For example, the following `Student`, and `Grade` are domain classes in the school application.

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }

    public Grade Grade { get; set; }
}
```

```
public class Grade
{
    public int GradeID { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Students { get; set; }
}
```

What is an Entity in Entity Framework?

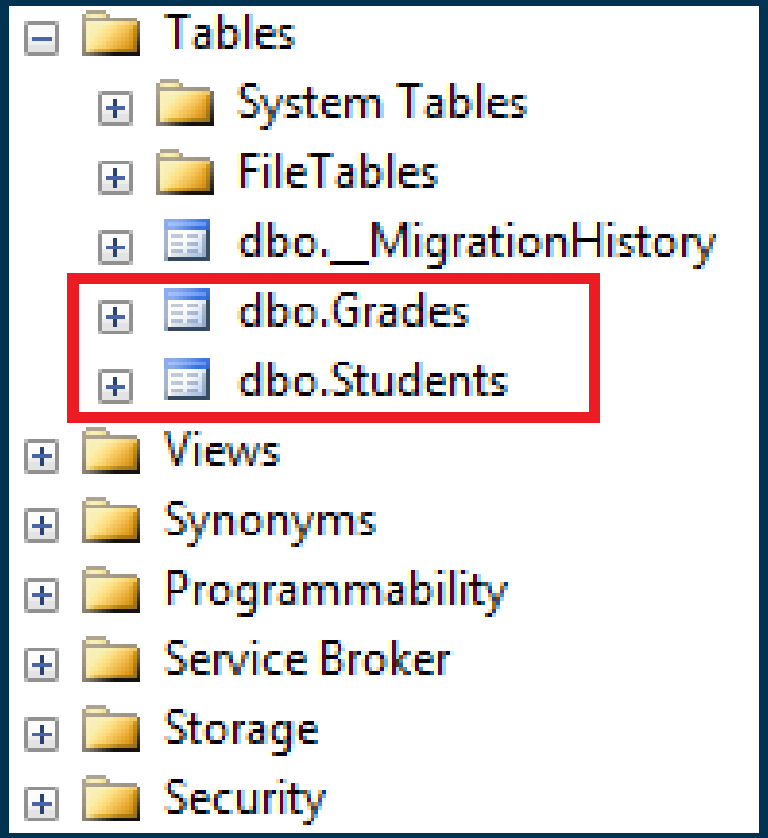
- These classes become entities when they are included as `DbSet<TEntity>` properties in a context class (the class which derives from `DbContext`), as shown below.

```
public class SchoolContext : DbContext
{
    public SchoolContext()
    {

    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```

- In the above context class, `Students`, and `Grades` properties of type `DbSet<TEntity>` are called entity sets. The `Student`, and `Grade` are entities.

What is an Entity in Entity Framework?

- EF will create the Students and Grades tables in the database:



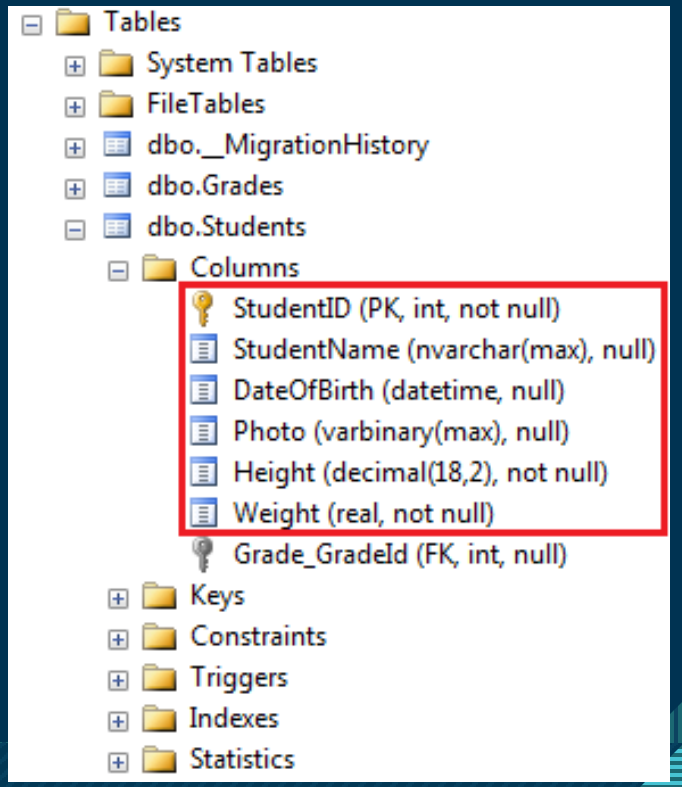
What is an Entity in Entity Framework?

- An Entity can include two types of properties:
  - Scalar Properties
  - Navigation Properties

Scalar Property

- The primitive type properties are called scalar properties.
- Each scalar property maps to a column in the database table which stores an actual data.
- For example, `StudentID`, `StudentName`, `DateOfBirth`, `Photo`, `Height`, `Weight` are the scalar properties in the `Student` entity class.
- EF will create a column in the database table for each scalar property.

```
public class Student
{
    // scalar properties
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    //reference navigation properties
    public Grade Grade { get; set; }
}
```



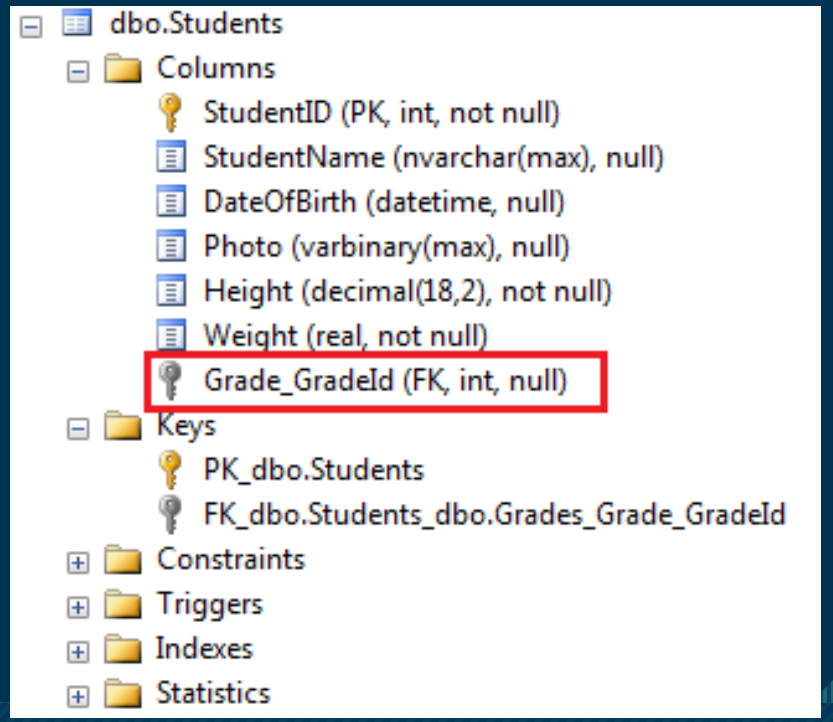
Navigation Property

- The navigation property represents a relationship to another entity.
- There are two types of navigation properties:
  - Reference Navigation
  - Collection Navigation

Reference Navigation Property

- If an entity includes a property of another entity type, it is called a Reference Navigation Property.
  - It points to a single entity and represents multiplicity of one (1) in the entity relationships.
- EF will create a ForeignKey column in the table for the navigation properties that points to a PrimaryKey of another table in the database.
- For example, `Grade` are reference navigation properties in the following `Student` entity class.
- In the database, EF will create a ForeignKey `Grade_GradeId` in the Students table.

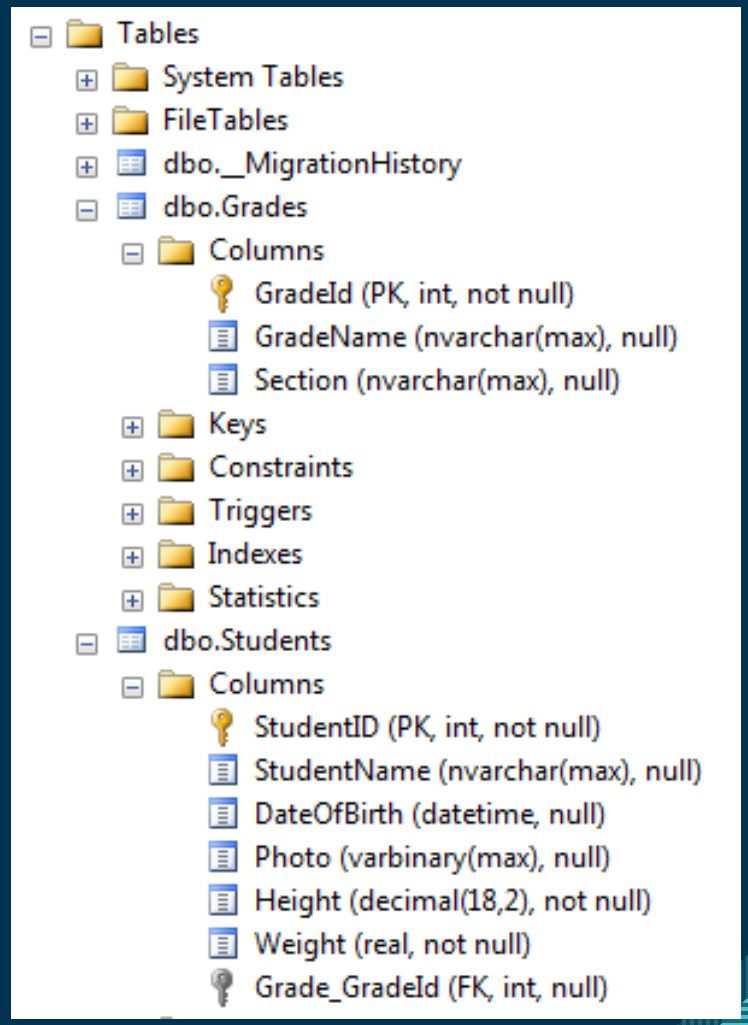
```
public class Student
{
    // scalar properties
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    //reference navigation properties
    public Grade Grade { get; set; }
}
```



Collection Navigation Property

- If an entity includes a property of generic collection of an entity type, it is called a collection navigation property.
  - It represents multiplicity of many (*).
- EF does not create any column for the collection navigation property in the related table of an entity.
- For example, the following `Grade` entity contains a generic collection navigation property `ICollection<Student>`.

```
public class Grade
{
    public int GradeID { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Students { get; set; }
}
```

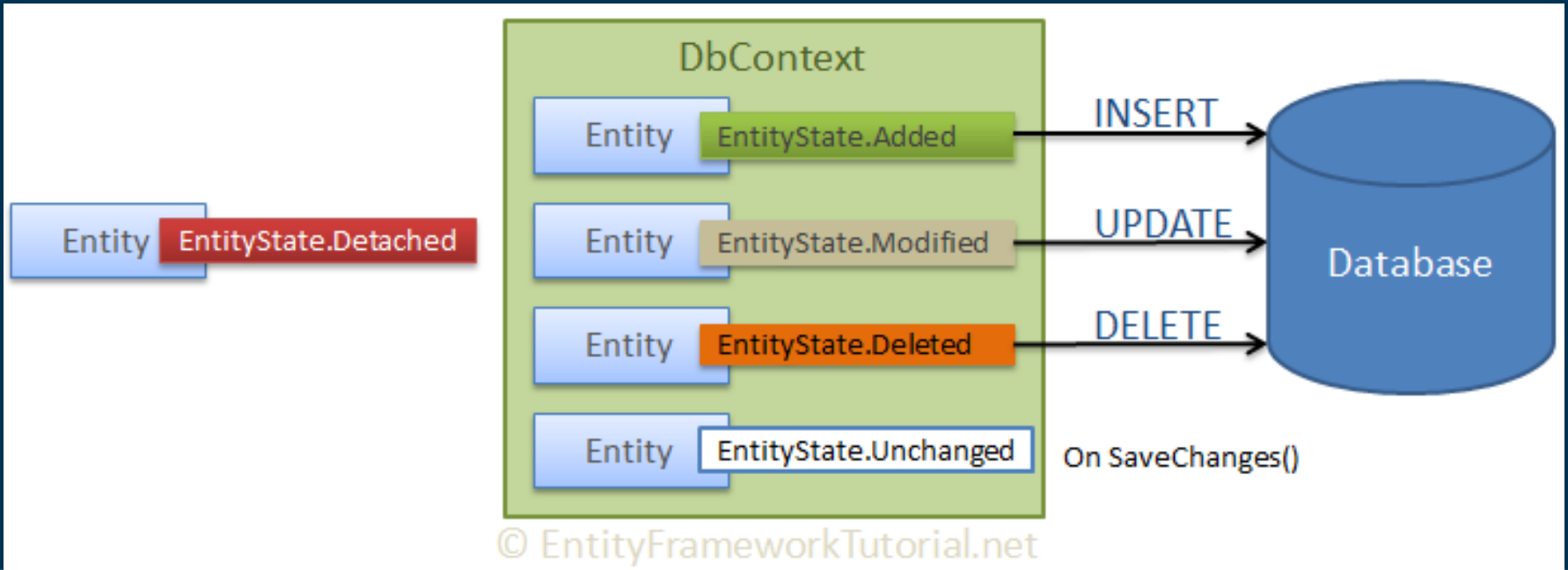


`EntityState` in Entity Framework

- EF maintains the state of each entity during its lifetime.
- Each entity has a state based on the operation performed on it.
- The entity state is represented by an enum `EntityState` with the following values:
  - `Added`
  - `Modified`
  - `Deleted`
  - `Unchanged`
  - `Detached`

EntityState in Entity Framework

- EF builds and executes the INSERT, UPDATE, and DELETE commands based on the state of an entity when the `context.SaveChanges()` method is called.
- It executes the INSERT command for the entities with `Added` state, the UPDATE command for the entities with `Modified` state and the DELETE command for the entities in `Deleted` state.
- The context does not track entities in the `Detached` state.

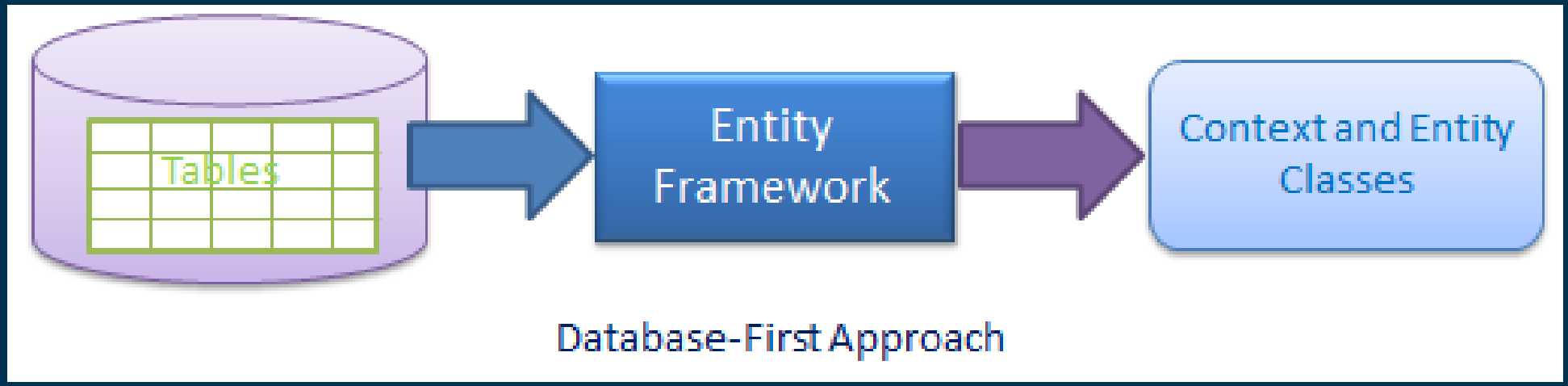


Development Approaches with Entity Framework

- There are three different approaches you can use while developing your application using Entity Framework:
  - Database-First
  - Code-First
  - Model-First

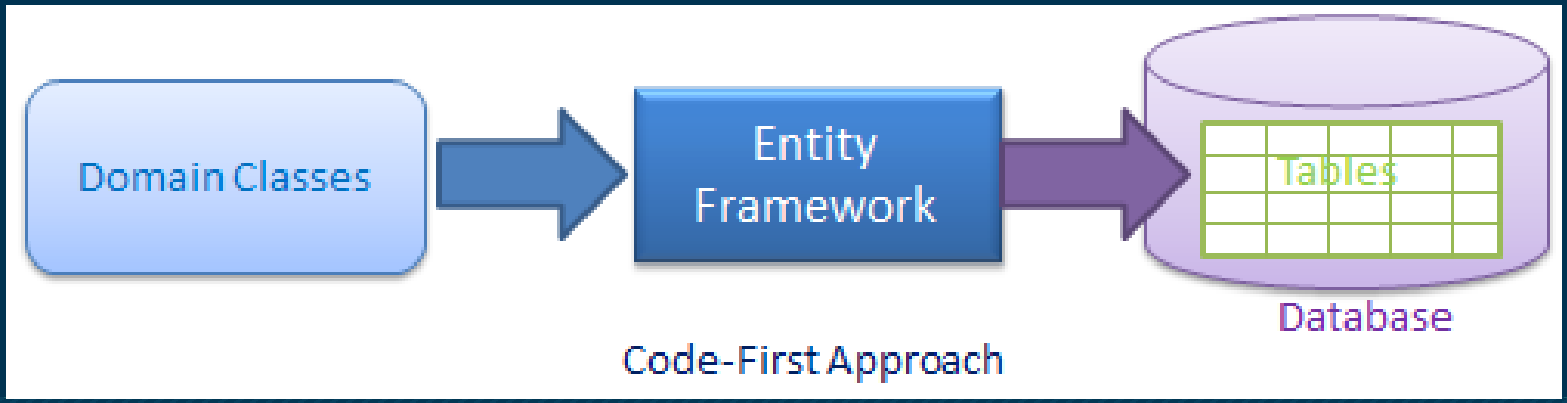
Database-First Approach

- In the database-first development approach, you generate the context and entities for the existing database using EDM wizard integrated in Visual Studio.



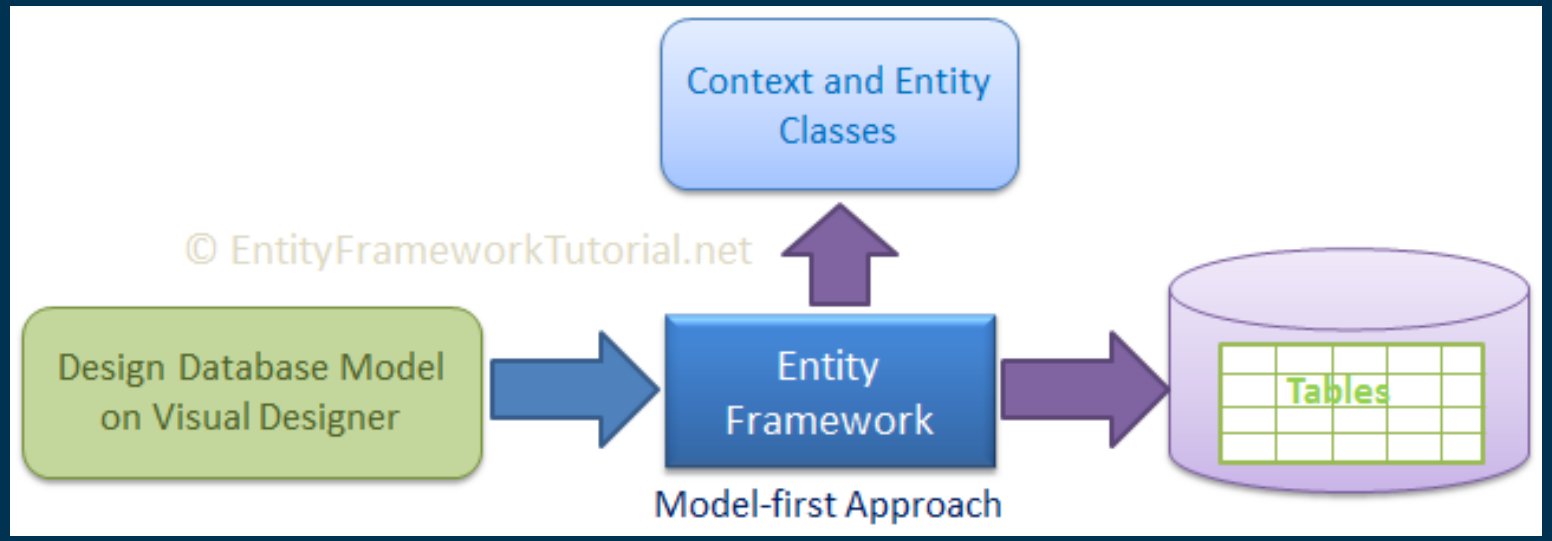
Code-First Approach

- Use this approach when you do not have an existing database for your application.
- In the code-first approach, you start writing your entities (domain classes) and context class first and then create the database from these classes using migration commands.
- Developers who follow the Domain-Driven Design (DDD) principles, prefer to begin with coding their domain classes first and then generate the database required to persist their data.



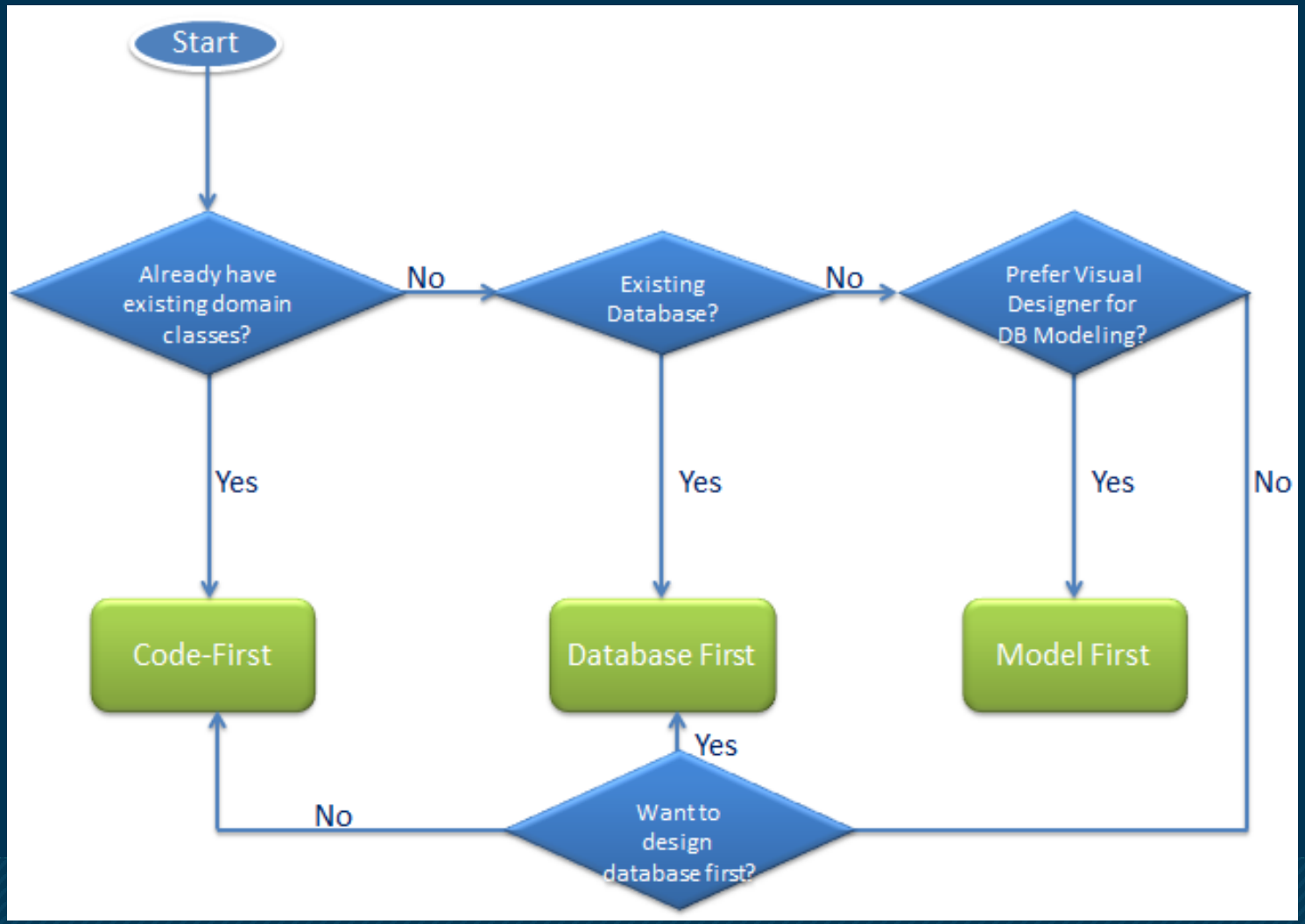
Model-First Approach

- In the model-first approach, you create entities, relationships, and inheritance hierarchies directly on the visual designer integrated in Visual Studio and then generate entities, the context class, and the database script from your visual model.



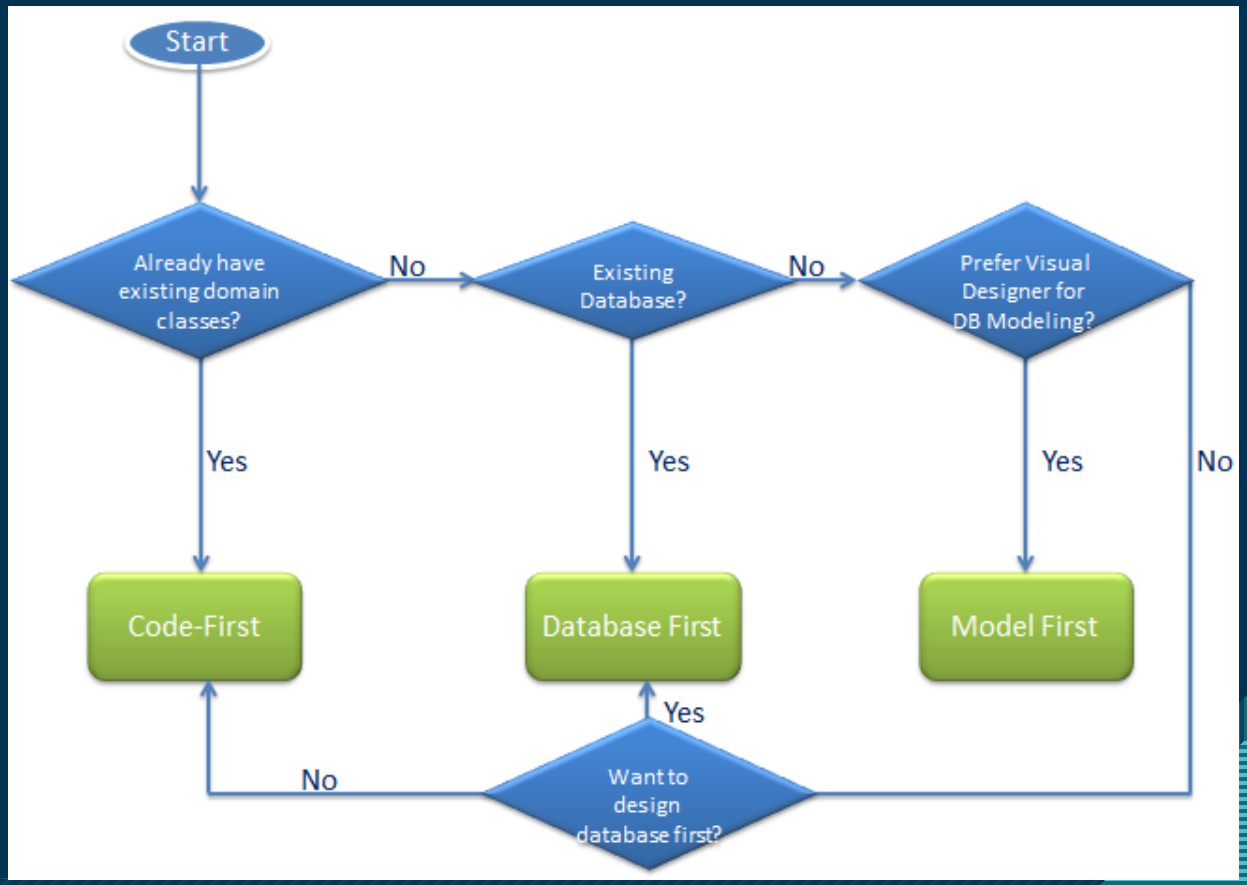
Choosing the Development Approach for Your Application

- Use the following flow chart to decide which is the right approach to develop your application using Entity Framework:



Choosing the Development Approach for Your Application

- If you already have an existing application with domain classes, then you can use the code-first approach because you can create a database from your existing classes.
- If you have an existing database, then you can create an EDM from an existing database in the database-first approach.
- If you do not have an existing database or domain classes, and you prefer to design your DB model on the visual designer, then go for the Model-first approach.



Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Most of the material has been taken as is from:

- What is an ORM and Why You Should Use it:
  - https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a
- What is Entity Framework?:
  - https://www.entityframeworktutorial.net/what-is-entityframework.aspx
- Entity Framework Core:
  - https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx