Entity Data Model

Entity Framework 1-3

In an ASP.NET Web application, you can use an Object Relationship Mapping (ORM) framework to simplify the process of accessing data from the application.

An ORM framework performs the necessary conversions between incompatible type systems in relational databases and object-oriented programming languages.

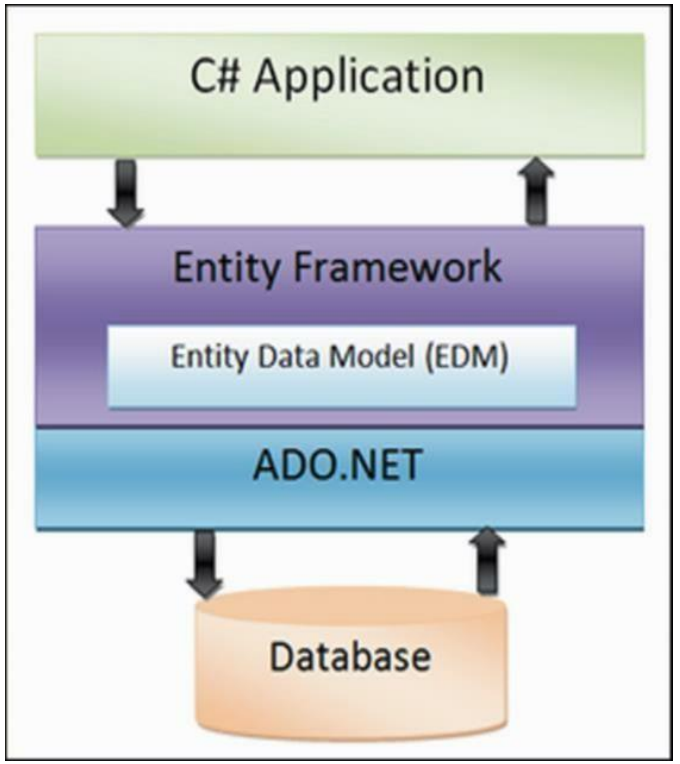
The ADO.NET Entity Framework is an ORM framework often used in .NET applications.

The Entity Framework is an implementation of the Entity Data Model (EDM) describing the entities and the associations that participate in an application.

EDM allows you to handle data access logic by programming against entities without having to worry about the structure of the underlying data store and how to connect with it.

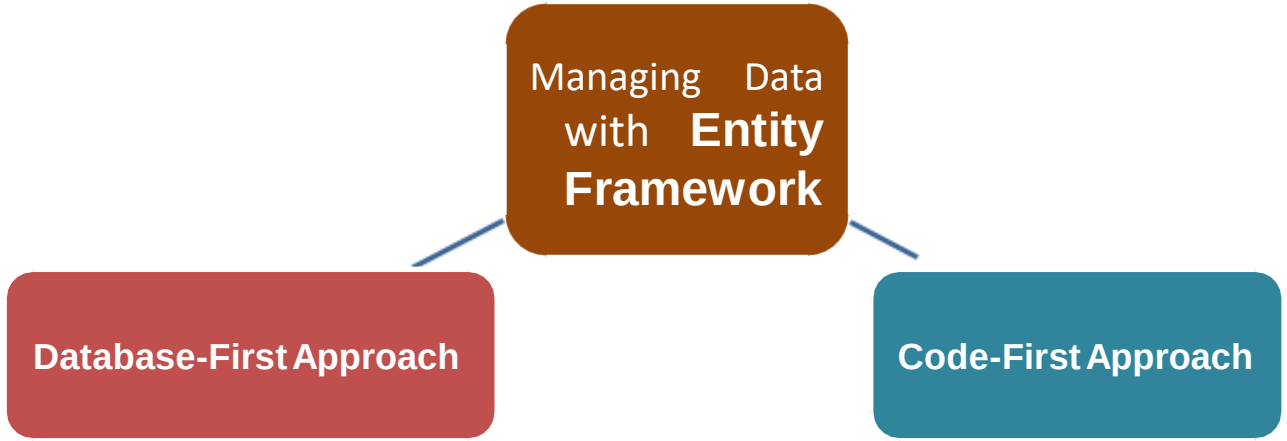
Entity Framework 2-3

- The `System.Data.Entity` namespace of the Entity Framework:
  - Provides classes that you can use to synchronize between the model classes and its associated database.
  - Also provides the **`DbContext`** class that coordinates with Entity Framework and allows you to query and save application data in the database.



Entity Framework 3-3

- **Entity Framework:**
  - Eliminates the need to write most of the data-access code that would otherwise need to be written.
  - Uses different approaches to manage data related to an application:



Database-First and Code-First

Database-First Approach

•The Entity Framework creates model classes and properties corresponding to the existing database objects, such as tables and columns.

- Applicable in scenarios where a database already exists for the application.

Code-First Approach

- The Entity Framework creates database objects based on custom classes that a programmer creates to represent the entities and their relationships in the application.
- It allows you to develop your application by coding model classes and properties and delegate the process of creating the database objects to the Entity Framework.
- The classes and properties will later correspond to tables and columns in the database.

Code-First Approach

- Allows you to provide the description of a model by using the C# classes.
- Based on the class definitions, the code-first conventions detect the basic structure for the model.
- The `System.Data.Entity.ModelConfiguration.Conventions` namespace provides several code-first conventions that enable automatic configuration of a model.

Conventions

- Some of these conventions are as follows:

Table Naming Convention

- Entity Framework by default creates a table named `Users` when you have created an object of the `User` model and need to store its data in the database.

Primary Key Convention

- When you create a property named `UserId` in the User model, the property is accepted as a primary key.
- The Entity Framework sets up an auto-incrementing key column to hold the property value.

Relationship Convention 1-5

Relationship Convention

- Entity Framework provides different conventions to identify a relationship between two models.
- You can use navigational properties in order to define relationship between two models.
- You should define a foreign key property on types that represents dependent objects.

Relationship Convention 2-5

- Consider a scenario:

  - You are developing an online shopping store.
  - For the application, you have created two model classes named `Customer` and `Order`.
  - Now, you need to declare properties in each class that allows navigating to the properties of another class.
  - You can then, define the relationship between these two classes.

Online Shopping Store

Relationship Convention 3-5

- Following code snippet creates the `Customer` model class:

```
public class Customer
{
  public int CustId { get; set; }
  public string Name { get; set; }
  // Navigation property
  public virtual ICollection<Order> Orders { get; set; }
}
```

  - This code creates a model named `Customer` that contains two properties named `CustId` and `Name`.

Relationship Convention 4-5

- Following code snippet creates the `Order` model class:

```
public class Order
{
    public int Id { get; set; }

    public string ProductName { get; set; }

    public int Price { get; set; }

    // Foreign key

    public int CustId { get; set; }

    // Navigation properties
    public virtual Customer cust { get; set; }
}
```

Relationship Convention 5-5

- In the code:
  - **Orders**: Is the navigational property in the `Customer` class.
  - **Cust**: Is the navigational property in the `Order` class.
  - These two properties are known as navigational properties as they allow to navigate to the properties of another class.
- For example:
  - You can use the `cust` property to navigate to the orders associated with that customer.
  - In the `Customer` class, the `Orders` navigational property is declared as a collection, as one customer can place multiple orders.
  - This indicates a one-to-many relationship between the `Customer` and `Order` classes.
  - The `CustId` property in the `Order` class is inferred as the foreign key by Entity Framework.

Database Context 1-10

- The `System.Data.Entity` namespace provides a `DbContext` class.
- DbContext class is the central component that serves as the bridge between your application and the underlying database. It provides a set of methods and properties that allow you to interact with the database, including querying, saving, and managing data.
- After creating the model class, you can use the `DbContext` class to define the database context class.
- This class coordinates with Entity Framework and allows you to query and save the data in the database.
- The database context class uses the `DbSet<T>` type to define one or more properties.
- In the type, `DbSet<T>,T` represents the type of an object that needs to be stored in the database.

Database Context 2-10

- Following code snippet shows how to use the `DbContext` class:

```
public class ShopDataContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

- In the code:
  - A database context class named `ShopDataContext` is created that derives from the `DbContext` class.
  - This class creates the `DbSet` property for both the `Customer` class and the `Product` class.

Database Context 3-10

- In a Web based cloud application:
  - You might need to change the model classes to implement various new features.
  - This also requires maintaining the database related to the model based on the changes made in the model class.
  - So while modifying the model classes, you should ensure that any changes in the model are reflected back in the database.
  - To maintain the synchronization between the model classes and its associated database, you need to recreate databases.

Database Context 4-10

- Entity Framework provides the `System.Data.Entity` namespace that contains the following two classes to recreate databases:

`DropCreateDatabaseAlways`	`DropCreateDatabaseIfModelChanges`
• Allows recreating an existing database whenever the application starts.	• Allows recreating an existing database whenever the associated model class changes.

- Based on your requirements, you can use one of these two classes in your application:
  - To recreate a database.
  - While calling the `SetInitializer()` method of the Database class defined in the `System.Data.Entity` namespace.

Database Context 5-10

- Following code snippet shows:
  - Creation of a new instance of the `DropCreateDatabaseAlways` class inside the `Application_Start()` method of the `Global.asax.cs` file:

```
. . .
Database.SetInitializer(new
        DropCreateDatabaseAlways<ShopDataContext>());
```

- In this code:
  - The `DropCreateDatabaseAlways` class is used while calling the `SetInitializer()` method to ensure that the existing database is recreated whenever the application starts.
  - You can use the `DropCreateDatabaseIfModelChanges` class to recreate a database only when the model changes.

```
Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<ShopDataContext>());
```

Database Context 6-10

- When you use the **DropCreateDatabaseAlways** database initializer, it drops and recreates the database every time the application starts. This means that any data that was previously stored in the database will be deleted.
- If you want to avoid losing data, you should consider using a different database initializer. For example, you can use the **CreateDatabaseIfNotExists** initializer, which creates the database if it does not exist but does not drop it if it already exists. You can also use the **MigrateDatabaseToLatestVersion** initializer, which applies any pending migrations to the database when the application starts.
- Alternatively, if you want to use **DropCreateDatabaseAlways** but still preserve some data, you can write code to seed the database with initial data after it is recreated. You can do this by overriding the **Seed method** of the database initializer and adding the necessary code to insert data into the database.

Database Context 7-10

- You can also populate a database with sample data for an application by creating a class that derives from either:
  - The `DropCreateDatabaseIfModelChanges` class or
  - The `DropCreateDatabaseAlways` class
- In this class:
  - You need to override the `Seed()` method that enables you to define the initial data for the application.

Database Context 8-10

- Following code snippet shows:
  - The **`MyDbInitializer`** class that uses the **`Seed()`** method to insert some sample data in the `Customers` database:

```
public class MyDbInitializer :
DropCreateDatabaseIfModelChanges<ShopDataContext>
{
    protected override void Seed(ShopDataContext context)
    {
        context.Customers.Add(new Customer() { Name = "John
        Parker", Address="Park Street", Email =
        "john@webexample.com" });
        base.Seed(context);
    }
}
```

Database Context 9-10

- In the code:
  - The `MyDbInitializer` class is derived from the `DropCreateDatabaseIfModelChanges` class.
  - Then, the `Seed()` method is overridden to define the initial data for the `Customer` model.
  - After defining the initial data for the customers, you need to register the `MyDbInitializer` model class in the `Global.asax.cs` file by calling the `SetInitializer()` method.

Database Context 10-10

- Following code snippet shows:
  - Use of the `SetInitializer()` method inside the `Application_Start()` method:

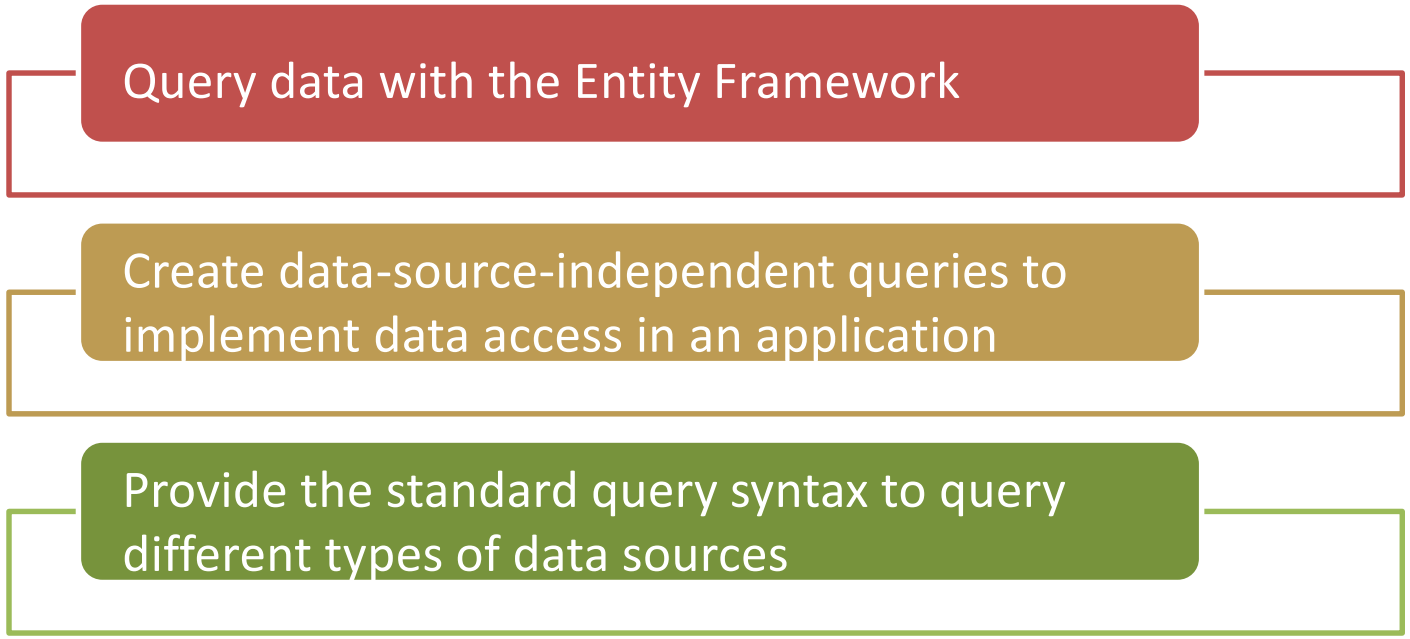
```
protected void Application_Start()
{
System.Data.Entity.Database.SetInitializer(new MyDbInitializer());
}
```

  - This code uses the `SetInitializer()` method to register the `MyDbInitializer` model class.

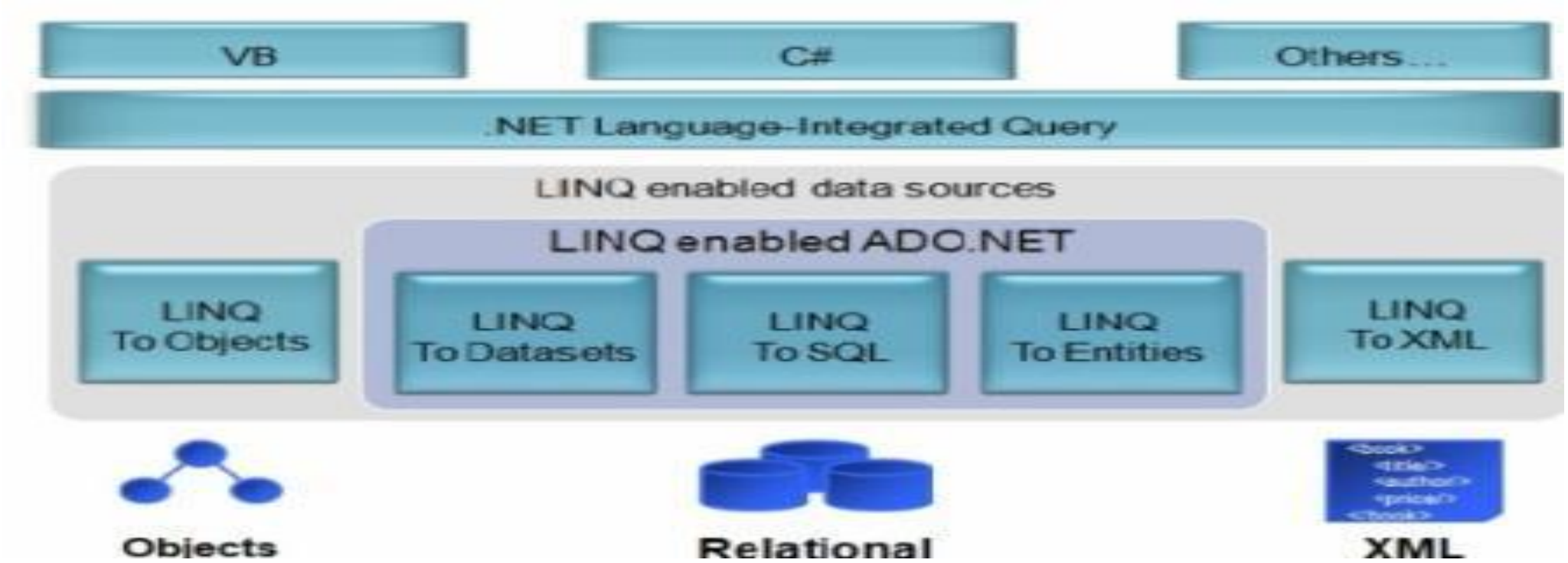
Query the Database Using LINQ to Entities

Querying Data with Entity Framework

- LINQ technologies are used to:



LINQ (Language Integrated Query)



DbContext

- Following code shows how to use the `DbContext` class:

```
public class ShopDataContext : DbContext
{
    public DbSet<Product> Products{ get; set;}
    public DbSet<Employee> Employees { get; set; }
}
```

- In this code:
  - A database context class named `OLShopDataContext` is created that derives from the `DbContext` class. This class creates the `DbSet` property for both, the `Product` class and the `Employee` class.
  - The key method of the `DbContext` class that you will most commonly use is the `SaveChanges()` method.
  - This method saves all changes made in the `DbContext` object to the underlying database.

DbContext

- Following code use to retrieve the category of a product:

```
using (var ctx = new ProductDbContext())
{
    IList <Product> productList = ctx.Products.ToList<Product>();
    Product product = productList[0];
    Category cat = product.Category;

}
```

Read (Select)

- To read data from the database, you use LINQ queries to query the DbSet properties in your DbContext class.
- You can filter, sort, and project the data as needed.

```
using (var context = new ShopDataContext ())
{
   var employees = context.Employees
                  .Where(e => e.Department == "Sales")
                  .OrderBy(e => e.Name)
                  .ToList();
}
```

Create (Insert)

- To create a new record in the database, you first create an instance of the entity class (model) that represents the table you want to insert data into.
- You then add the new entity instance to the appropriate DbSet (a collection that represents the table) in your DbContext class.
- Finally, you call the SaveChanges() method on the DbContext to commit the changes to the database.

```
using (var context = new ShopDataContext ())
{
   var newEmployee = new Employee { Name = "John Doe", Department = "Sales" };
   context.Employees.Add(newEmployee);
   context.SaveChanges();
}
```

Update

- To update an existing record in the database, you first retrieve the entity instance you want to update from the DbSet.
- You then modify the properties of the entity instance and call the SaveChanges() method to commit the changes to the database

```
using (var context = new ShopDataContext ())
{
   var employee = context.Employees.Find(1);
   employee.Department = "Marketing";
   context.SaveChanges();
}
```

Delete

- To delete a record from the database, you first retrieve the entity instance you want to delete from the DbSet.
- You then call the Remove() method on the DbSet to mark the entity for deletion, and finally call SaveChanges() to commit the changes.

```
using (var context = new ShopDataContext ())
{
   var employee = context.Employees.Find(1);
   context.Employees.Remove(employee);
   context.SaveChanges();
}
```

Delete [Cont…]

```
using (var context = new ShopDataContext ())
{
var employeeToDelete = context.Employees
                        .Where(e => e.Name == "John")
                        .FirstOrDefault();
   if (employeeToDelete != null)
   {
      context.Employees.Remove(employeeToDelete);
      context.SaveChanges();
      Console.WriteLine("Employee with name 'John' has been deleted.");
   }
   else
   {
      Console.WriteLine("No employee with the name 'John' found.");
   }
}
```