# Session: 6

# Data Access

# Objectives

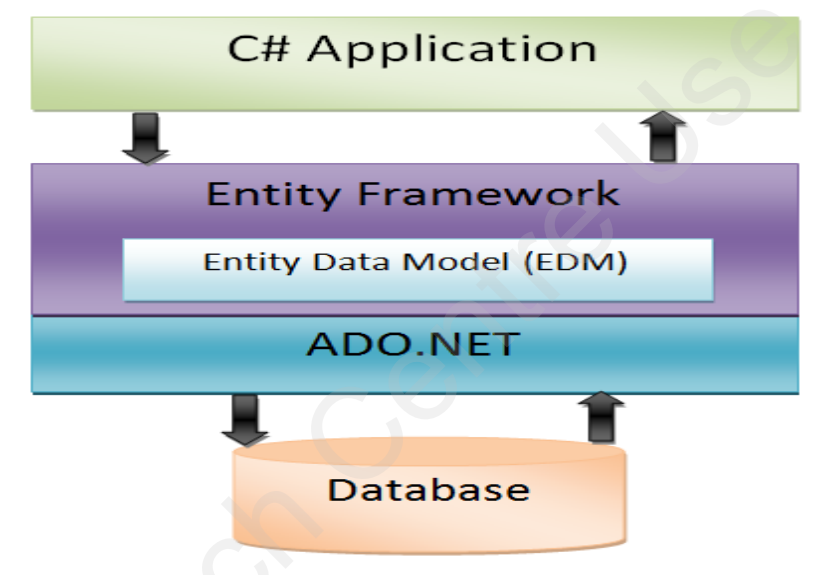
- Define and describe the Entity Framework
- Explain how to work with the Entity Framework
- Define and describe how to initialize a database with sample data
- Explain how to use LINQ queries to perform database operations

# Entity Framework 1-2

- To address the data access requirements of ASP.NET MVC application, you can use an ORM framework.
- An ORM framework:
  - Simplifies the process of accessing data from applications.
  - Performs the necessary conversions between incompatible type systems in relational databases and object-oriented programming languages.
- The Entity Framework is an ORM framework that ASP.NET MVC applications can use.
- The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they 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 2-2

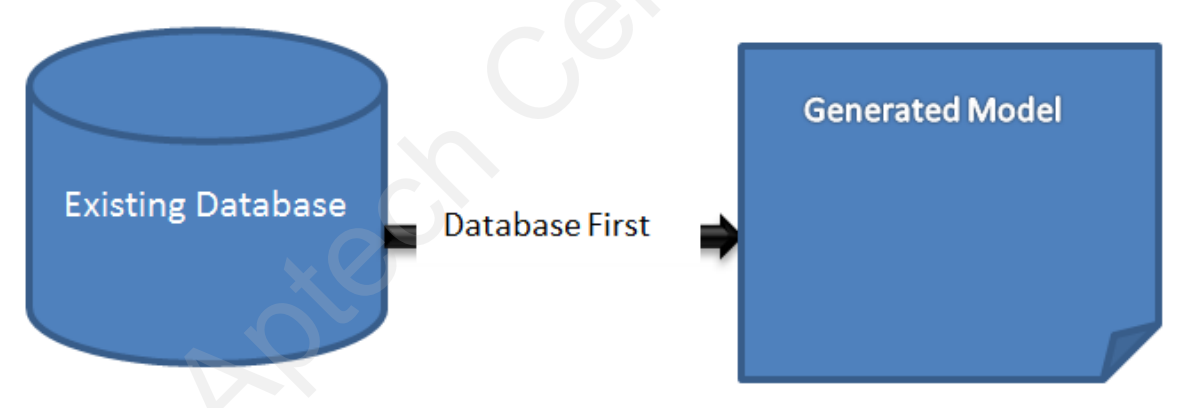
- Following figure shows the role of EDM in the Entity Framework architecture:




- Entity Framework:
  - Eliminates the need to write most of the data-access code that otherwise need to be written.
  - Uses different approaches, such as database-first and code-first to manage data related to an application.

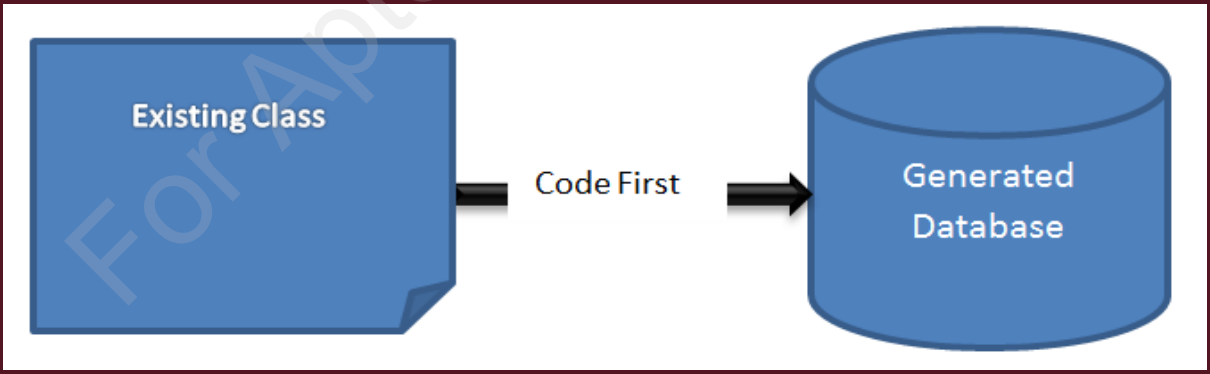
# Database-first Approach

- In the database-first approach the Entity Framework creates model classes and properties corresponding to the existing database objects, such as tables and columns.
- The database-first approach is applicable in scenario where a database already exists for the application.
- Following figure shows the database-first approach:



# Code-first Approach

- In the code-first approach the Entity Framework creates database objects based on model classes that you create to represent application data.
- The code-first approach:
  - Is the most common approach implemented in ASP.NET MVC Framework.
  - 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.
- Following figure shows the code-first approach:



# Working with Entity Framework

- The Entity Framework uses different approaches to manage data related to objects.
- The code-first approach allows you to define your own model by creating custom classes.
- Then, you can create database based on the models.
- There are certain conventions that you should follow for the code-first approach.

# Implementing Code-first Approach 1-3

- The 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.
- Some of the code-first conventions that enable automatic configuration of a model are as follows:
  - Table naming convention: When you have created an object of the User model and need to store its data in the database, Entity Framework by default creates a table named Users.
  - Primary key convention: When you create a property named UserId in the User model, the property is accepted as a primary key. In addition, the Entity Framework sets up an auto-incrementing key column to hold the property value.
  - Relationship convention: Entity Framework provides different conventions to identify a relationship between two models.

# Implementing Code-first Approach 2-3

- Consider a scenario of an online shopping store where:
  - You have created two model classes named Customer and Order.
  - Then you need to declare properties in each class that allows navigating to the properties of another class.
  - Finally, define the relationship between these two classes.
- Following code snippet creates a Customer model class:

Snippet

```
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.

# Implementing Code-first Approach 3-3

- Following code snippet creates a Order model class:

Snippet

```
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; }
}
```

- In this 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 because they allow us to navigate to the properties of another class.

# The DbContext Class 1-2

- The DbContext class:
  - Is provided by the System.Data.Entity namespace of the ASP.NET MVC Framework.
  - Can be used to define the database context class after creating a model class.
  - Coordinates with Entity Framework and allows you to query and save the data in the database.
  - Uses the DbSet <T> type to define one or more properties where, T represents the type of an object that needs to be stored in the database.

# The DbContext Class 2-2

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

Snippet

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

    }
```

- In this code:
  - A database context class named OLShopDataContext is created that derives from the DbContext class.
  - The DbContext class creates the DBSet property for both, the Customer class and the Product class.

# Initializing a Database with Test Data 1-6

- While developing an ASP.NET MVC Web application, you might need to change the model classes to implement various new features.
- This 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.

# Initializing a Database with Test Data 2-6

- The Entity Framework provides the following two classes under the System.Data.Entity namespace that allows recreating databases:
  - DropCreateDatabaseAlways: Allows recreating an existing database whenever the application starts.
  - DropCreateDatabaseIfModelChanges: 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.
- You can use one of these two classes while calling the SetInitializer() method of the System.Data.Entity.Database namespace.

# Initializing a Database with Test Data 3-6

- Following code snippet shows using the DropCreateDatabaseAlways class inside the Application_Start() method of the Global.asax.cs file:

Snippet

```
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.
- On the other hand, you can use the DropCreateDatabaseIfModelChanges class to recreate a database only when the model changes.

# Initializing a Database with Test Data 4-6

- Following code snippet shows using the DropCreateDatabaseIfModelChanges class inside the Application_Start() method:

Snippet

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

- In this code, the DropCreateDatabaseIfModelChanges class is used while calling the SetInitializer() method to ensure that the existing database is recreated whenever the model changes.
- You can also instruct the MVC Framework to populate a database with sample data for an application.
- This can be achieved by creating a class that derives from either the DropCreateDatabaseIfModelChanges class or the DropCreateDatabaseAlways class.

# Initializing a Database with Test Data 5-6

- Following code snipept shows the MyDbInitializer model class that uses the Seed() method to insert some sample data in the Customers database:

Snippet

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

- In this 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.

# Initializing a Database with Test Data 6-6

- Once you have defined 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.
- Following code snipept shows using the SetInitializer() method inside the Application_Start() method:

Snippet

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

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

# LINQ Query

- LINQ:
  - Is a set of APIs that allows you to create data-source-independent queries to implement data access in an application.
  - Has a programming model that provides the standard query syntax to query different types of data sources.
  - Queries are written in any .NET Framework supported programming language, such as VB or C#.
  - Query acts on a strongly-typed collection of objects with the help of language keywords and common operators.
  - Helps you to quickly manipulate data from the various data sources without requiring them to learn a new query language.

# Using a Simple LINQ Query 1-3

- In an ASP.NET MVC application:
  - A LINQ query operation starts by creating a query.
  - Once the query is created, the next operation is to execute the query.
  - To execute a LINQ query, you need to first specify the data source.
  - LINQ queries operate on objects that implement either the IEnumerable<T> or IQueryable<T> interface.
  - Both these interfaces enable you to create queries to retrieve data from a specific data source.

# Using a Simple LINQ Query 2-3

- Following code snippet shows creating a query that retrieves customer details from the Customer data source:

Snippet

```
IQueryable<Customer> q = from s in db.customers select s;
```

where,

- from: Clause specifies the data source that contains the data.
- db: Is an instance of the data context class provides access to the data source.
- s: Is the range variable.
- select: Clause specifies that each element in the result will consist of a customer object.

# Using a Simple LINQ Query 3-3

- Once you have created a LINQ query:
  - You need to execute it to retrieve the data.
  - Then iterate over the result set using a foreach loop.
- Following code snippet shows using foreach loop to retrieve customer details:

Snippet

```
string names = "";
IQueryable<Customer> q = from s in db.customers
                         select s;
                        foreach (var cust in q)
                        {
                           names = names+" "+cust.Name;
                        }
      ViewBag.Name = names;
```

- In this code:
  - The foreach loop retrieves the query results and the cust variable stores each value one at a time.
  - Then, the name of each customer is appended to the string variable, names.
  - Finally, the names variable is assigned to the Name property using ViewBag to display the information in a view.

# Using Advance LINQ Queries 1-7

- In addition to retrieve data from data sources, you can also use LINQ queries to perform various operations on data stored in a data source.
- Some of the common operations that you perform on data using LINQ queries include:
  - Forming Projections
  - Filtering the data
  - Sorting the data
  - Grouping the data

# Using Advance LINQ Queries 2-7

- Sometimes, you might only need to retrieve specific properties of a model from the data store, for example, only the Name property of the Customer model.
- You can achieve this by forming projections in the select clause.
- Following code snippet shows a LINQ query that retrieves only the customer names of the Customer model class:

Snippet

```
public static void DisplayCustomerNames() {
using (Model1Container dbContext = new Model1Container()) {
IQueryable<String> query = from c in dbContext.Customers
                                        select c.Name;
 Console.WriteLine("Customer Names:");
 foreach (String custName in query)
 {
Console.WriteLine(custName);

            }
          }
        }
```

# Using Advance LINQ Queries 3-7

- In the preceding code:
  - The select clause retrieves a sequence of customer names as an IQueryable<String> object.
  - The foreach loop iterates through the result to print out the names.
- Following is the output of the preceding code:

```
Customer Names:
Alex Parker
Peter Milne
```

# Using Advance LINQ Queries 4-7

- The where clause in a LINQ query enables filtering data based on a Boolean condition, known as the predicate.
- The where clause applies the predicate to the range variable that represents the source elements and returns only those elements for which the predicate is true.
- Following code snippet uses the where clause to filter customer records:

Snippet

```
public static void DisplayCustomerByName() {
using (Model1Container dbContext = new Model1Container()) {
        IQueryable<Customer> query = from c in dbContext.Customers
where c.Name == "Alex Parker"select c;
             Console.WriteLine("Customer Information:");
            foreach (Customer cust in query)
              {
              Console.WriteLine("Customer ID: {0}, Name: {1},
Address: {2}", cust.CustomerId, cust.Name, cust.Address);
               }
            }
        }
```

# Using Advance LINQ Queries 5-7

- In the preceding code:
  - The where clause to retrieve information of the customer with the name Alex Parker.
  - The foreach statement iterate through the result to print the information of the customer.
- Following is the output of the preceding code:

```
Customer Information:
Customer ID: 1, Name: Alex Parker, Address: 10th Park Street,
Leo Mount
```

# Using Advance LINQ Queries 6-7

- You can also use LINQ queries to retrieve data and then, display it in sorted manner by using the orderby clause.
- This clause specifies whether the result should be displayed in either ascending or descending order.
- While using the orderby clause the result is displayed in the ascending order by default.
- Following code snippet shows using the ascending keyword to display the customer name in ascending order:

Snippet

```
IQueryable<Customer> q = from s in dbContext.customers
                         where s.City == "New Jersey"
                         orderby s.Name ascending
                         select s;
```

- This code will retrieve and then, displays the customer name who lives in New Jersey in the ascending order.

# Using Advance LINQ Queries 7-7

- You can also use LINQ query to retrieve and display the data as a group.
- For this, you need to use the group clause that allows you to group the results based on a specified key.
- Following code snippet shows how to group the customers according to their cities:

Snippet

```
var q = from s in dbContext.customers
        groups by s.City;
```

- This code retrieves the customer records and groups these records based on the city where the customers lives.

# Using LINQ Method-Based Queries 1-3

- The LINQ queries used so far are created using query expression syntax.
- Such queries are compiled into method calls to the standard query operators, such as select, where, and orderby.
- Another way to create LINQ queries is by using method-based queries where you can directly make method calls to the standard query operator, passing lambda expressions as the parameters.

# Using LINQ Method-Based Queries 2-3

- Following code snippet shows the use of Select clause to project the Name and Address properties of Customer model class into a sequence of anonymous types:

Snippet

```
public static void DisplayPropertiesMethodBasedQuery() {
     using (Model1Container dbContext = new Model1Container()) {
        var query =dbContext.Customers.Select(c=> new
                {     CustomerName = c.Name,
                      CustomerAddress = c.Address
                 });
              Console.WriteLine("Customer Names and Addresses:");
              foreach (var custInfo in query)
               {
                 Console.WriteLine("Name: {0}, Address: {1}",
    custInfo.CustomerName, custInfo.CustomerAddress);
                 }
            }
        }
```

# Using LINQ Method-Based Queries 3-3

- The preceding code uses the Select clause to project the Name and Address properties of Customer model class into a sequence of anonymous types.
- Following is the output of the preceding code:

  Customer Names and Addresses:

  Name: Alex Parker, Address: 10th Park Street, Leo Mount

  Name: Peter Milne, Address: Lake View Street, Cheros Mount
- Similarly, you can use the other operators such as Where, GroupBy, Max, and so on through method-based queries.

# LINQ Data Providers

- LINQ provides a consistent programming model to create standard query expression syntax to query different types of data sources.
- However, different data sources accept queries in different formats.
- To solve this problem, there are several LINQ providers, such as LINQ to SQL, LINQ to Objects, and LINQ to XML.
- The LINQ queries, that you have learned till now uses the LINQ to SQL provider.
- This provider allows you to access SQL-complaint databases and makes data available as objects in an application.

# LINQ to Objects

- LINQ to Objects refers to the use of LINQ queries with enumerable collections, such as List<T> or arrays.
- To use LINQ to query an object collection, you need to declare a range variable.
- Following code snippet shows how to access data from a string array that contains the names of customers in a class:

Snippet

```
string products= "";
string[] arr = new string[] { "Laptop", "Mobile", "Jewellery"};
var query = from string production arr
select product;
foreach (var pin query)
{
products= products+ " " + p;
}
```

- This code accesses the data from a string array that contains the names of customers.

# LINQ to XML 1-2

- You can use the LINQ to XML queries to work with XML in an application.
- LINQ to XML query allows accessing data that is stored in XML files.
- Following code snippet shows the content of an XML file named, CustomersRecord.xml:

Snippet

```
<?xml version="1.0" encoding="utf-8"?>
<CustomersDetails>
<Customer>
<CustID>CId1001</CustID>
<Name>Peter Jones</Name>
<City>New York</City>
</Customer>
<Customer>
<CustID>CId1002</CustID>
<Name>Jessica Parker</Name>
<City>London</City>
</Customer>
</CustomersDetails>
```

- This code shows an XML document, named CustomersRecord.Xml that contains customer details.

# LINQ to XML 2-2

- Now, to access the data from CustomersRecord.Xml you can use LINQ.
- Following code snippet shows how to retrieve data from an XML file by using LINQ query:

Snippet

```
string result = "";
 XDocument xmlDoc = XDocument.Load("E:\\CustomerRecord.xml");
       var q = from c in xmlDoc.Descendants("Customer")
                   select (string)c.Element("CustID") + "-" +
                  (string)c.Element("Name") + "-" + (string)c.Element("City");
              foreach (string entry in q)
              {
                  result += entry + " | ";
              }
```

- This code will retrieve all the customer details from the CustomersRecord.Xml file.
- The xmlDoc.Descendants() method returns a collection of the descendant elements for the specified element in the order they are present in the XML file.

# Summary

- The Entity Framework is an ORM framework that ASP.NET MVC applications can use.
- In the database-first approach, the Entity Framework creates model classes and properties corresponding to the existing database objects.
- In the code-first approach, the Entity Framework creates database objects based on model classes that you create to represent application data.
- The System.Data.Entity namespace of the ASP.NET MVC Framework provides a DbContext class that coordinates with Entity Framework and allows you to query and save the data in the database.
- LINQ is a set of APIs that allows you to create data-source-independent queries to implement data access in an application.
- LINQ queries operate on objects that implement either the IEnumerable<T> or IQueryable<T> interface.
- LINQ queries can also be created as method-based queries where you can directly make method calls to the standard query operator, passing lambda expressions as the parameters.