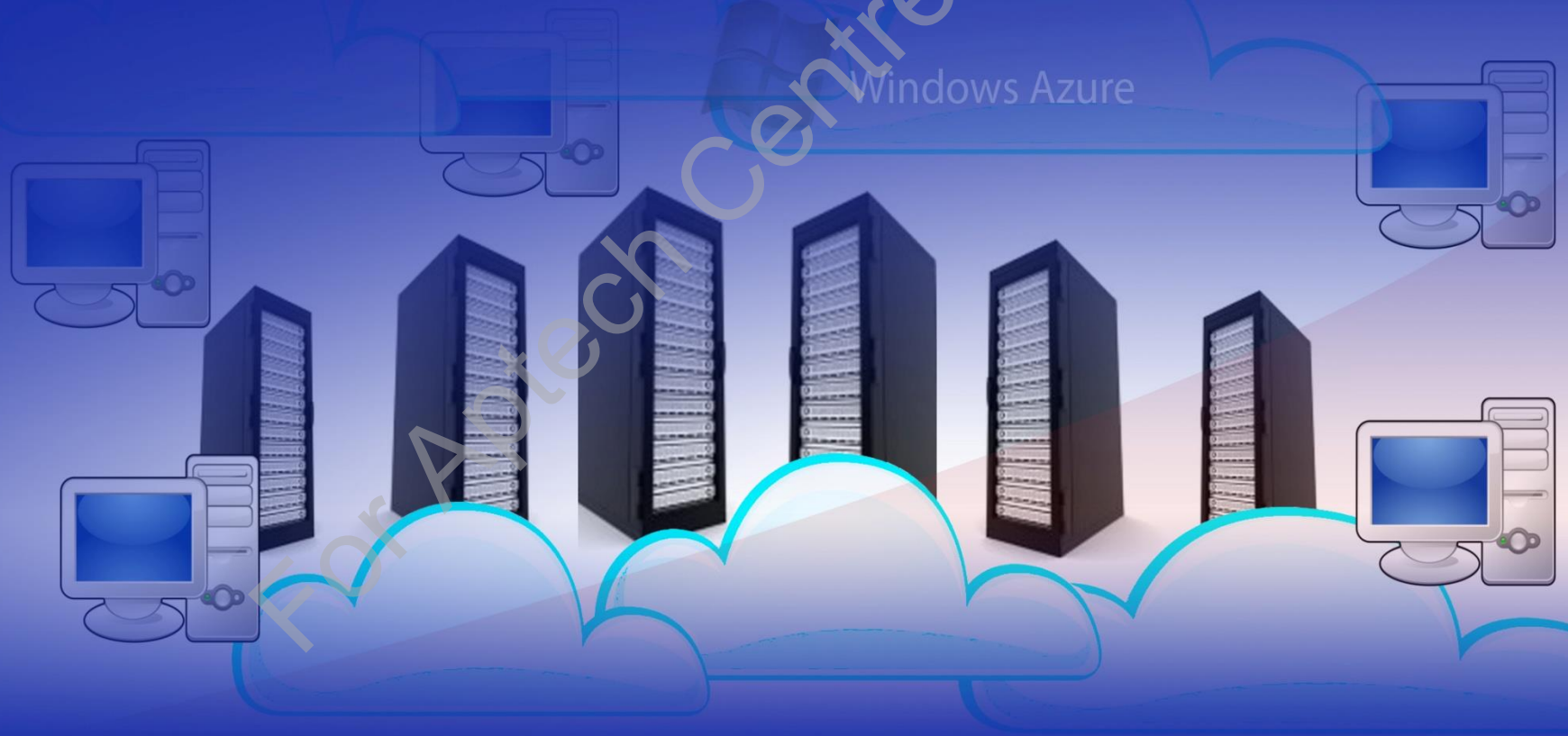
Enterprise Application Development Using Windows Azure and Web Services

Session 6

Advanced Concept of Data Access

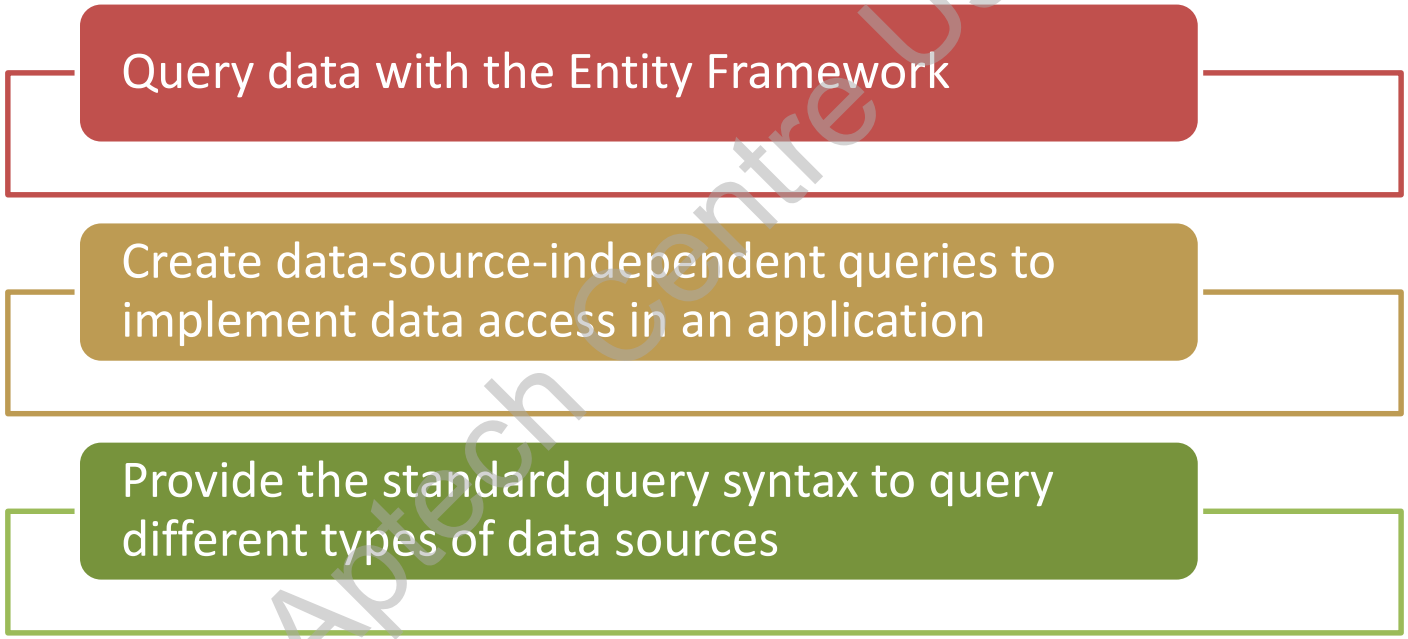


Learning Objectives

- Define and describe how to query data with Entity Framework
- Explain how to work with transactions
- Explain how to create and use Entity Framework data model

Querying Data with Entity Framework

- LINQ technologies are used to:



- Before using LINQ, you need access to a database context object that the `DbContext` class represents.

DbContext and Lazy Loading 1-3

- The Entity Framework:
  - Provides a `DbContext` class in the `System.Data.Entity` namespace that implements a database context.
  - Provides another database context implementation known as `ObjectContext`.
- The `DbContext` class:
  - Is a wrapper around the `ObjectContext` class.
  - Is responsible for coordinating with the 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. In the type, `DbSet<T>`, `T` represents the type of an object that needs to be stored in the database.

DbContext and Lazy Loading 2-3

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

```
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. This class creates the `DbSet` property for both, the `Customer` class and the `Product` 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. The `DbContext` class supports lazy loading that delays the loading of data until a specific request for the data arrives.
  - You can configure lazy loading by setting the `DbContext.Configuration.LazyLoadingEnabled` property to `True`.

DbContext and Lazy Loading 3-3

- Following code uses lazy loading to retrieve the category of a product:

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

- In this code:
  - Lazy loading is used as a result of which two SQL queries will get executed.
  - The first query will retrieve all products, while the second query will only execute to retrieve the Category property of the referred product, which in this case, is the first product of the product list.

Querying with LINQ to SQL 1-2

- LINQ to SQL is used to:
  - Access SQL-compliant databases.
  - Make data available as objects in an application to map with the database objects to enable working with data.
  - Map objects help to perform all the database operations, such as select, insert, update, and delete on a database.
  - Retrieve the data from a database, the LINQ to SQL query are converted into SQL queries.
  - Send the query to the database where the database query execution engine executes the query and returns results.
  - Use an instance of the database context class of the application to cache the data retrieved from the database and send the data to the application.
  - Use the instances of the model class to represent the tables of the database in the application.

Querying with LINQ to SQL 2-2

- Following code describes how to access the names of the customers of an online shopping store from the `Customer` table:

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

- This code retrieves the names of the customers from the `Customers` table.

Querying with Entity SQL 1-2

- Entity SQL is a query language that you can use to:
  - Query entity models in the Entity Framework. Entity SQL is similar to SQL.
  - Query data with Entity SQL against the Entity Framework without requiring a steep learning curve.
  - Query data either as objects or in a tabular form.
  - Construct a query statement dynamically as a string, instead of committing to its structure in code.

Entity SQL resembles Structured Query Language (SQL), but instead of using table or view names inside a statement, Entity SQL uses domain class names.

Querying with Entity SQL 2-2

- Following code uses Entity SQL to query a database:

```
var context =new AppDbContext();
var ObjectContext = (context as
IObjectContextAdapter).ObjectContext;
string eSql= "SELECT VALUE prod FROM AppDbContext.Product AS
prod";
var query = objectContext.CreateQuery <Product>(eSql);
List<Product> products =query.ToList();
```

- In this code:
  - An `ObjectContext` instance is obtained from the `DbContext` object.
  - Then, the `CreateQuery<T>` generic method of the `ObjectContext` class is called to execute the Entity SQL query and retrieve objects from the database.

Querying with LINQ to Entities 1-3

- In LINQ to Entities, a programmer:
  - Creates a query that returns a collection of zero or more typed entities.
  - Needs a data source against which the query will execute.
  - An instance of the `ObjectQuery` class represents the data source.
  - Stores a query in a variable.
  - When the query is executed, it is first converted into a command tree representation that is compatible with the Entity Framework.

Querying with LINQ to Entities 2-3

- Following code creates and executes a query to retrieve the records of all `Customer` entities along with the associated `Order` entities:

```
public static void DisplayAllCustomers()
{
    using (Model1Container dbContext = new Model1Container())
    {
        IQueryable<Customer> query = from c in dbContext.Customers
        select c;
        Console.WriteLine("Customer Order Information:");
        foreach (var cust in query)
        {
            Console.WriteLine("Customer ID: {0}, Name: {1}, Address:
              {2}", cust.CustomerId, cust.Name, cust.Address);
            foreach (var cst in cust.Orders)
            {
                Console.WriteLine("Order ID: {0}, Cost: {1}",
                    cst.OrderId, cst.Cost);
            }
        }
    }
}
```

Querying with LINQ to Entities 3-3

- In this code:
  - The `from` operator specifies the data source from where the data has to be retrieved.
  - `dbContext` is an instance of the `DbContext` class that provides access to the **Customers** data source, and **c** is the range variable.
  - When the query is executed, the `range` variable acts as a reference to each successive element in the data source.
  - The `select` operator in the LINQ query specifies the type of the returned elements as an `IQueryable<Customer>` object. The `foreach` loops iterate through the results of the query returned as an `IQueryable<Customer>` object to print the customer and order details.
- **Output:**

```
Customer Order Information:
Customer ID: 1, Name: Alex Parker, Address: 10th Park Street,
Leo Mount
Order ID: 1, Cost: 575
Customer ID: 2, Name: Peter Milne, Address: Lake View Street,
Cheros Mount
Order ID: 2, Cost: 800
```

LINQ to SQL and the Entity Framework 1-3

- LINQ to SQL and the Entity Framework:
  - Provides support for precompiled queries.
  - Performs the query transformation when the query is first executed.
  - Reuses the transformed query each time it executes.
- Following code shows the LINQ query that you can use to address this requirement:

```
ctx.Students.Where(s=>s.StudentID==_studentID)
```

- In this code,
  - You can convert the preceding query into a precompiled query by using the `CompiledQuery` class of the Entity Framework.
  - This class present in the `System.Data.Objects.CompiledQuery` namespace provides a `Compile()` method.
  - This method takes the current `DataContext` or `ObjectContext` object as parameter.
  - This method also accepts any parameters that need to be passed to the query and the type that the query returns. This method returns a `Func (delegate)` that can be later invoked.

LINQ to SQL and the Entity Framework 2-3

- Following code shows a query that will be precompiled by the Entity Framework:

```
var _studentRec = CompiledQuery.Compile<StudentsEntities, int,
Student>   ((ctx, id)
=>ctx.Students.Where(s=>s.StudentID==_studentID).Single());
```

- The `Compile()` method when executed will generate the following result:

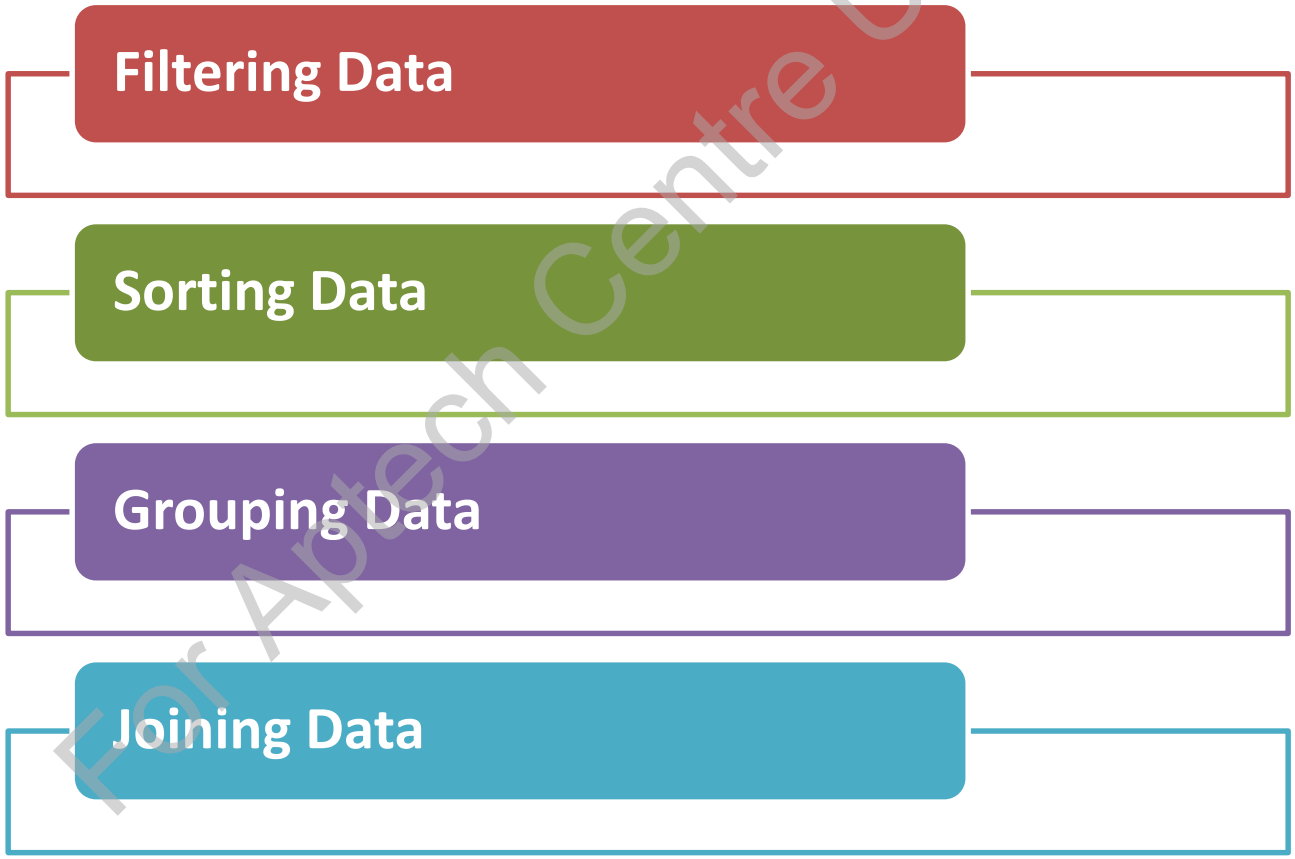
```
private System.Func<StudentsEntities, int, Student> studentRec
```

- Once the query is compiled, it can be invoked from the application, as shown in the following code:

```
Student student = studentRec.Invoke(ctx, studentID);
```

LINQ to SQL and the Entity Framework 3-3

- LINQ provides several operators, which are methods that you can use to query data. These operators support functionalities such as:



Filtering Data 1-2

- You can use the `where` operator in a LINQ query to perform data filtering based on a `Boolean` condition, known as the **predicate**. The `where` operator 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 uses the where operator to filter customer records:

```
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);
      }
    }
}
```

Filtering Data 2-2

- This code uses the `where` operator to retrieve information of the customer with the name `AlexParker`. The `foreach` statement iterates through the result to print the information of the customer.

- Output:

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

Sorting Data

- The `orderby` operator is used to:

Retrieve data and display it in sorted manner.

Specify the results in either ascending or descending order.

Retrieve the records from the Customer table and use the `ascending` keyword along with the `orderby` operator.

- Following code shows the use of the `ascending` keyword to display the customer name in ascending order:

```
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, display the customer name who lives in New Jersey in the ascending order.

Grouping Data

- The `group` operator is used to:

Retrieve and display the data as a group.	Group the results based on a specified key.

- Following code shows how to group the customers according to their cities:

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

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

Joining Data 1-2

- The `join` operator is used to:
  - Associate the elements in one table with the elements in another table, where the two tables share a common attribute.
  - Retrieve the elements from different tables on the basis of a common attribute.

Joining Data 2-2

- Following code shows usage of the join operator to retrieve records of all the male customers who have placed an order online:

```
var customers = db.customers;
var orderdetails = db.orderdetails;
var list = (from s in customers
join t in orderdetails on
s.Name equals t.Customer
where s.Gender == "M"
select new {Customer=s.Name,
Product=t.Product}).ToList();
var orders = "";
foreach (var order in list)
{
     orders += order.Customer + " : " + order.Product + " ";
}
```

- This code retrieves the details of the customers from the `Customer` and `orderdetails` tables by joining the tables using the `join` operator.

CRUD with LINQ to SQL 1-2

- To perform CRUD operations using the LINQ to SQL queries:
  - Log the query that is executed to database out of the LINQ query.
  - Use the `Log` property of the `DataContext` class.
  - Analyze and identify any problems in the LINQ query you executed.

- Following code uses the `Log` property of the `DataContext` class to log the queries that are executed:

```
using (System.IO.StreamWriter sw = new
System.IO.StreamWriter(@"C:\tempdatacontext.log"))
{
    AppDataContext ctx = new AppDataContext();
    ctx.Log = sw;
    var cust = from c in ctx.Customers
    select c;
    List<customer> custList = cust.ToList();
}
```

CRUD with LINQ to SQL 2-2

- The code creates a `StreamWriter` object to write to the `C:\tempdatacontext.log` file.
- An `AppDataContext` object that is an implementation of the `DataContext` class is created and the `Log` property is used to set the `StreamWriter` object as the destination to write the SQL query.
- Finally, a LINQ to SQL query is executed.
- At runtime, when the LINQ to SQL query executes, the `tempdatacontext.log` file stores the SQL code.

Implementing Query Boundaries 1-3

- While using LINQ to query data from database and collections:
  - Use both `IEnumerable` and `IQueryable` objects for data manipulation.
  - Understand their differences and when to use either of the `IQueryable` or `IEnumerable` objects in an application.
  - The `IEnumerable` interface is available in the `System.Collections` namespace and is suitable for LINQ to Object and LINQ to XML queries.
  - Use an `IEnumerable` object to query data from collections such as `List` and `Array`.
  - Use an `IEnumerable` object to query data from a database, the object executes the query on the database server, loads the retrieved results, and performs any specified data filtration.

Implementing Query Boundaries 2-3

- Following code uses an `IEnumerable` object to query data from a `Students` table for students whose name starts with the letter `E`:

```
AppDataContext ctx = new AppDataContext ();
IEnumerable <Student> list = ctx.Students.Where(s
     =>s.Name.StartsWith("E"));
```

Implementing Query Boundaries 3-3

- The `IQueryable` interface:
  - Is available in the `System.Linq` namespace and derives from the `IEnumerable` interface.
  - Should be used for LINQ to SQL queries to query data sources, such as remote databases or Web services.
  - Provides methods, such as `CreateQuery()` and `Execute()` to create custom queries.
  - When you use an `IQueryable` object to query data from a database, the object executes the query along with any specified filters on the database server and then loads the filtered records.

- Following code uses an `IQueryable` object to query data from a `Students` table for students whose name starts with the letter `E`:

```
AppDataContext ctx = new AppDataContext ();
IQueryable<Student> list = ctx.Students.Where(s
=>s.Name.StartsWith("E"));
```

Working with Transactions 1-9

- When developing an enterprise application:

Identify the related data access operations that implement a specific functionality of the application.

Execute related data access operations together as a single unit to address the business requirement of the application.

Ensure that the set of operations will not be executed even if a single operation of the unit fails to execute.

Address such requirements using a transaction. A transaction is a single atomic unit of work where multiple operations occur in sequence.

In a transaction, all the operations defined for the transaction will either successfully complete or fail even if one of the defined operations of the unit fails.

Working with Transactions 2-9

- Following are the four key transaction properties:

Atomicity	• Specifies that a transaction should be considered a single unit of operation consisting of a sequence of independent operations, where all the operations complete either successfully or unsuccessfully.
Consistency	• Ensures that a transaction ends by leaving the database in a valid state. This means that even if the transaction fails, the state of the database remains the same as it was before the transaction started.
Isolation	• Ensures that multiple transactions occurring simultaneously remain isolated from each other to prevent data corruption.
Durability	• Ensures that the results of a successful transaction are permanent, regardless of any system failure.

Working with Transactions 3-9

System.Transactions API

- The `System.Transactions` namespace:
  - Contains classes that can be used to implement transactions in applications.
  - Implements transaction programmatically in an application.
  - Uses the classes of the `System.Transactions` namespace for implicit transaction management, where transaction is managed by the infrastructure.
- Following table lists the key classes of the `System.Transactions` namespace:

Class	Description
`Transaction`	Represents a transaction.
`TransactionInformation`	Provides additional information regarding a transaction.
`TransactionManager`	Contains methods used for transaction management. This class cannot be inherited.
`TransactionScope`	Makes a code block transactional. This class cannot be inherited.

- When Entity Framework is used in an enterprise application, the framework handles transaction management.

Working with Transactions 4-9

A transaction that updates data on two or more computers in a network is known as distributed transaction.

- Distributed transactions:
  - Involve multiple resources, such as database servers and messaging systems.
  - Involve a transaction manager that controls and coordinates the transaction on the resources.
  - If any failure occurs in any resource, the transaction must roll back in all other resources participating in the transaction.

Working with Transactions 5-9

TransactionScope Class

- Enterprise applications require multiple connections to multiple databases:

Step 1

- Use the `TransactionScope` class of the `System.Transactions` namespace to combine multiple operations into a single transaction that may be distributed across multiple systems.

Step 2

- Create the `TransactionScope` object.

Step 3

- Create multiple `DbContext` objects that may be performing different operations on different database servers.

Working with Transactions 6-9

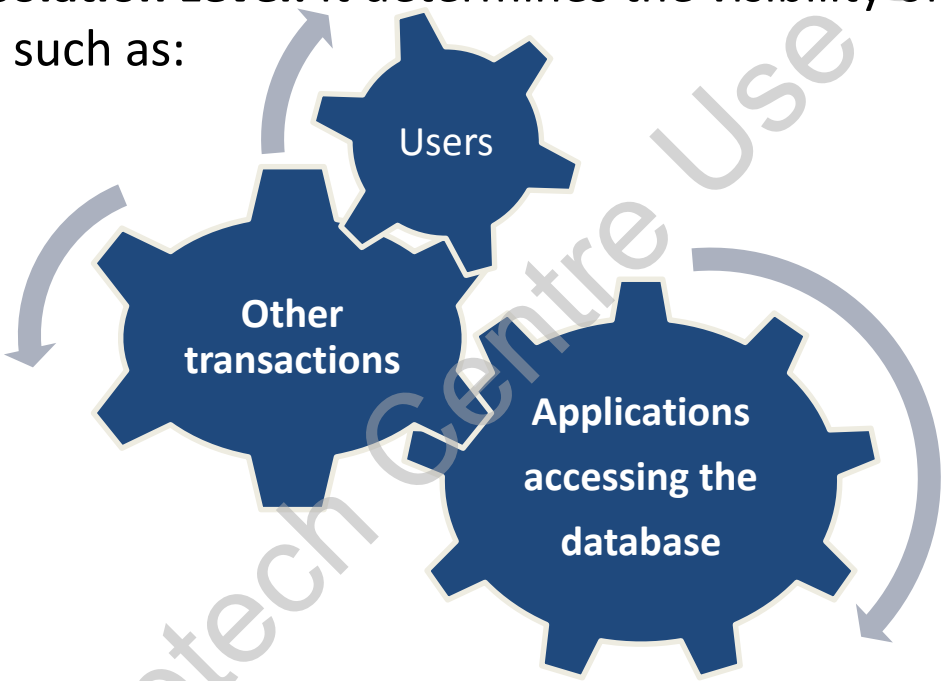
- Following code uses the `TransactionScope` class to execute multiple operations of `DbContext` objects in a single transaction:

```
using(var transactionScope = new TransactionScope())
{
     AppDbContext1 appCtx1= new AppDbContext1();
     appCtx1.saveChanges();
     AppDbContext2 appCtx2= new AppDbContext2();
     appCtx2.saveChanges();
     AppDbContext3 appCtx3= new AppDbContext3();
     appCtx3.saveChanges();
     transactionScope.Complete();
}
```

- In this code, the `TransactionScope` class is created to execute different operations performed by three `DbContext` objects in a single transaction.

Working with Transactions 7-9

- **Transaction Isolation Level:** It determines the visibility of transactions to other entities such as:



- Consider an example:
  - A transaction is storing invoice records in a database and the transaction has inserted the invoice headers and data of five invoices.
  - It is yet to insert all the invoices in the system. At this stage, transaction isolation determines whether or not the partial data entered is accessible to other entities.

Working with Transactions 8-9

A lower isolation level will enable other entities to access the partial data being inserted by the current transaction, but at the risk of data inconsistency.

A higher isolation level will prevent other entities from accessing the data until the current transaction completes successfully.

Working with Transactions 9-9

- The ISO standard defines the following isolation levels, all of which are supported by the SQL Server Database Engine:

Read uncommitted	Read committed	Repeatable read	Serializable
• Lowest level of transaction isolation. • Ensures that corrupt data is not being made accessible to the entities accessing it.	• Intermediate isolation level at a higher level of the Read uncommitted mode. • Ensures that a transaction may read only the data that has been committed in the database.	• Intermediate isolation level after the Read committed level. • Ensures that a transaction may read only the data that has been committed in the database.	• Highest level of transaction isolation. • One transaction is completely isolated from one another based on the ACID principles.

Creating and Using Entity Data and Model

- Entity Data Model (EDM) is a conceptual model that:
  - Describes the entities and the associations they participate in an application.
  - Allows a programmer to handle data access logic by programming against entities.

- Plain Old CLR Objects (POCO):
  - Is an object of a normal class that remains independent and has no concern about the infrastructure, such as the Entity Framework to execute as a part of the infrastructure.

Implementing a Data Model 1-4

- Steps to implement a data model in Visual Studio 2013:

Step 1	Open **Visual Studio 2013**.
Step 2	Create an ASP.NET Web Application project named **EDMDemo** with the MVC template.
Step 3	Right-click **EDMDemo** in the **Solution Explorer** window, and select **Add → New Item**.
Step 4	Select **Data** from the left menu and then select **ADO.NET Entity Data Model**.
Step 5	Click **Add**. The **Entity Data Model Wizard** is displayed.
Step 6	Select **Empty model**.

Implementing a Data Model 2-4

Step 7	Click **Finish**. The **Entity Data Model Designer** is displayed.
Step 8	Right-click the **Entity Data Model Designer**, and select **Add New Entity**.
Step 9	Enter **Student** in the **Entity name** field.
Step 10	Click **OK**.
Step 11	Right-click the **Student** entity and select **Add New → Scalar Property.**
Step 12	Enter **Name** as the name of the property.
Step 13	Similarly, add a **Term** property to the **Student** entity.
Step 14	Add another **Address** property to the **Student** entity.

Implementing a Data Model 3-4

Step 15	Right-click the **Entity Data Model Designer**, and select **Generate Database from Model**.
Step 16	Click **New Connection**.
Step 17	Enter **(localdb)\v11.0** in the **Servername** field and **EDMDEMO.StudentDB** in the **Select or enter a database name** field.
Step 18	Click **OK**. Visual Studio will prompt whether to create a new database.
Step 19	Click **Yes**.
Step 20	Click **Next**. The **Generate Database Wizard** window displays the generated DLL scripts required to create the database objects.
Step 21	Click **Finish**. Visual Studio 2013 opens the file containing the SQL scripts to create the database objects.
Step 22	Right-click the file and select **Execute**. The **Connect to Server** dialog box is displayed.

Implementing a Data Model 4-4

Step 23	Click **Connect**. Visual Studio will create the database objects.
Step 24	Click **Build → Build Solution**. This will add the **Student.cs** class.
Step 25	Expand the **Controllers** folder in the **Solution Explorer** window.
Step 26	Double-click the **HomeController** controller class. The Code Editor displays the code of the **HomeController** class.
Step 27	Add the `AddStudent()` method after the default `Index()` method.
Step 28	Right-click inside the `AddStudent()` method and select **Add View** from the context menu. The **Add View** dialog box is displayed.
Step 29	Click **Add**. The Code Editor displays the code of the **AddStudent.cshtml** view.
Step 30	Add the code to display a message in the **AddStudent.cshtml** view.
Step 31	Click **Debug → Start Without Debugging**. The browser displays the home page of the application.
Step 32	Type the following URL in the address bar of the browser: `http://localhost:12719/Home/AddView`

Summary 1-2

- You can use LINQ to query data with the Entity Framework that allows you to create data-source-independent queries to implement data access in an application.
- The Entity Framework provides a DbContext class that coordinates with the Entity Framework and allows you to query and save the data in the database.
- Entity SQL is query language similar to Structured Query Language (SQL) that you can use to query entity models in the Entity Framework.
- LINQ provides several operators, which are methods that you can use to query data that support functionalities to filter, sort, group, and join data.

Summary 2-2

- The DataContext class provides the Log property that allows you to log the queries that are executed in a file.
- The IEnumerable interface is suitable for LINQ to Object and LINQ to XML queries while the IQueryable object should be used for LINQ to SQL queries to query data sources.
- A transaction is a single atomic unit of work where all the operations defined for the transaction will either successfully complete or fail even if one of the defined operations of the unit fails.