Unit 1.5: Entity framework

Lecturer: Le Thi Bich Tra

Content

What is ORM?

- ORM stands for Object Relational Mapping
  - Your application is object-oriented (O)
  - Your database is relational (R)
  - Your application maps to your database (M)
- Bridging the database with your application
  - Tables = Classes, Entities represent your domain objects
  - Columns = Properties
  - Relationships = Collections or association
  - Stored Procedures = Functions
- ORM is a framework automatically creates classes based on database tables, and the vice versa is also true, that is, it can also automatically generate necessary SQL to create database tables based on classes.

What is Entity Framework?

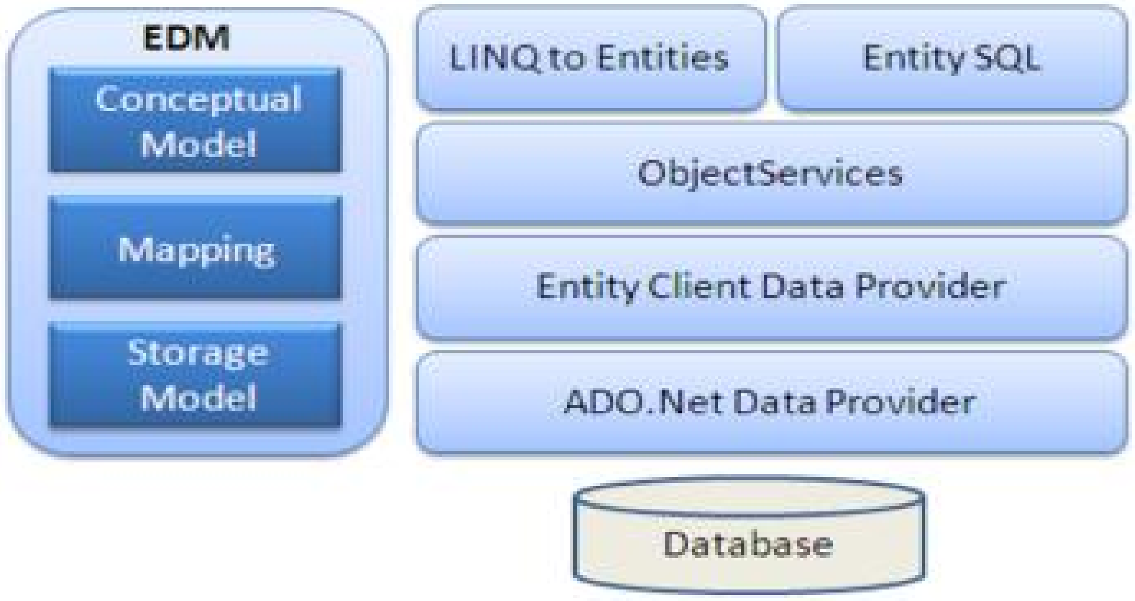
- Entity Framework (EF) is an ORM framework that enables .NET developers to work with relational data has a domain specific object, eliminating the use of most of the data access plumbing code that developers usually need to write.
- "...object – relational mapper..": This is a programming technique used to convert "incompatibles types" into object that can used by programming languages.
- "..domain-specific object..": Specific to a particular area. EX: Accounting system->invoices, products, vendor objects,..
- "..eliminates...data-access code..": connection strings, data adapters, datasets, tables objects,...

What is Entity Framework?

- Entity Framework (EF) is an enhancement to ADO.NET that gives developers an automated mechanism for accessing and storing the data.
- Developers can write LINQ queries for getting the data from the database using EF.
- EF helps us to retrieve and manipulate data as strongly typed objects.

Entity Framework Architecture

› Architect of EF:
  - EDM (Entity Data Model): is a model that describes entities and the relationships between them.



  - Add EDM:

Install Entity Framework

- Install Entity Framework 6:
- Go to Tools -> Library Package Manager -> Package Manager Console:

  Install-package entityframework
- ConnectionString

```
<connectionstring>
        <add name="ProductContext" connectionString="data source=.\SQLExpress;
        initial catalog=ProductDB;integrated security=True;MultipleActiveResultSets=True;
        App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

DbContext

- The DBContext is the backbone of the Entity Framework.
- The DbContext is often referred as the context is the class which is responsible for interacting with the entity model and the database. It allows you to query, insert, update and delete operations on the entities.
- Using DbContext:

```
public class EFContext : DbContext
   {
       public EFContext() : base("EFDatabase")
       {
       }
       public DbSet<Employee> Employees { get; set; }
   }
```

```
using (var context = new ProductContext())
  {
     // Perform data access using the context
  }
```

DbSet

- DBSet class represents an entity set that is used for CRUD operations.
- The DBContext class exposes the DBSet Property for the each entity in the model. Some of the important methods of DBSet class:
  - Add
  - Attach(Entity)
  - Create
  - Find
  - Include
  - Remove

Approaches in Entity Framework

- Database First
- Model First
- Code First

Database First

- The Database First approach enables us to create an entity model from the existing database. This approach helps us to reduce the amount of code that we need to write. The following procedure will create an entity model using the Database First approach.
- Step 1: Create ADO.NET Entity Data Model using “Generate from Database”.
- Step 2: Select an existing Connection or create new connection.
- Step 3: Select database objects (table, view, stored procedure)
- This will generate with selected entities and association.
- To update the model (in case of database change), right-click on the model and select “Update Model from Database”.

Model First

- Model classes and their relation is created first using the ORM designer and the physical database will be generated using this model. The Model First approach means we create a diagram of the entity and relation that will be converted automatically into a code model.
- Step 1: Create the ADO.net entity data model using the "Empty EF Designer Model" option.
- Step 2: Create the Entity by right-click on the diagram, Add new.
- Step 3: Add properties to the entity.
- Step 4: Add associations (relation among entities) if any.
- Step 5: Generate the database from this model using “"Generate Database from model“, we can select any existing database connection or create a new database connection.

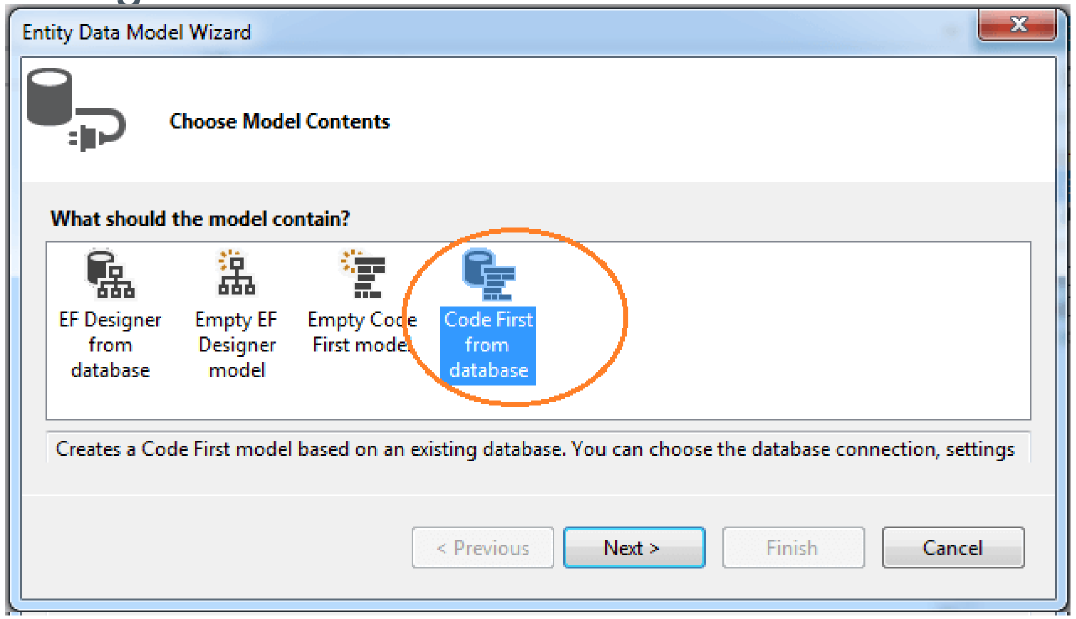
Step 5 will generate the DDL script and this generated file will be added to the solution as a script file.

Code First

- The Code First approach enables us to create a model and their relation using classes and then create the database from these classes. It enables us to work with the Entity Framework in an object-oriented manner. Here we need not worry about the database structure.
- Step 1: Create a class that represents the database table.
- Step 2: Create DbContext class and define a DbSet properties type of entity classes that are represented as a table in the database.
- Step 3: Create the database from the model, open Package Manager Console:
  - Enable Migration
  - Add Migration
  - Update database

Code First

- Code First with Existing Database
- Install EF
- Add EDM


Entity Data Model Wizard

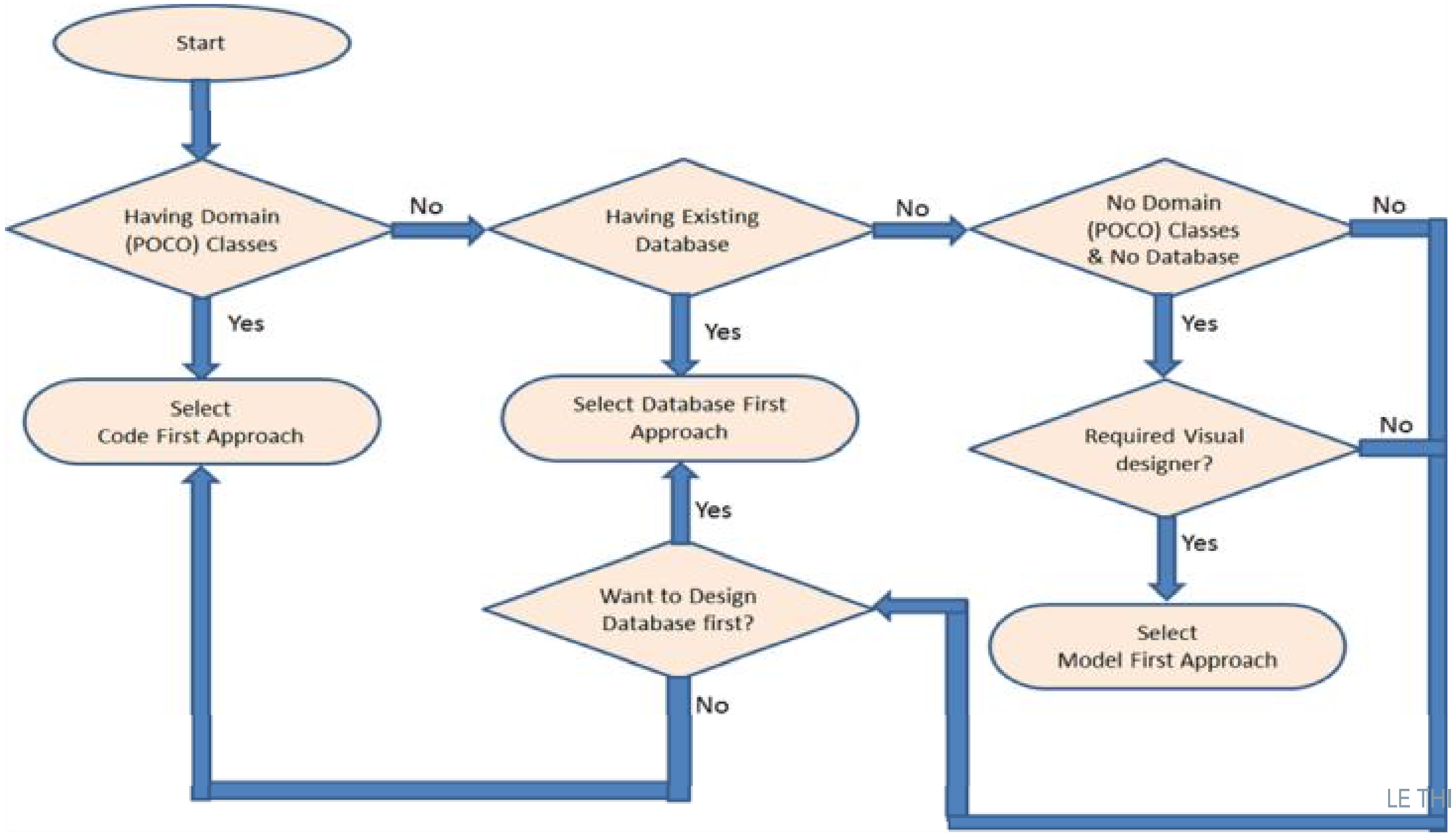
Choose Model Contents

What should the model contain?

Creates a Code First model based on an existing database. You can choose the database connection, settings

Approaches in Entity Framework

› Selecting Right Approach?



Start
Having Domain (POCO) Classes
No
Having Existing Database
No
No Domain (POCO) Classes & No Database
No
Yes
Select Code First Approach
Yes
Select Database First Approach
Yes
Required Visual designer?
No
Yes
Want to Design Database first?
Yes
Select Model First Approach
No

Relationship in Entity Framework using Code First

› There are several types of database relationships:
  - One-to-One Relationships
  - One-to-Many or Many to One Relationships
  - Many-to-Many Relationships

Relationship in Entity Framework using Code First

- One-to-One

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string Name { get; set; }
    //
    //Navigation property Returns the Employee Address
    public virtual EmployeeAddress EmployeeAddress { get; set; }
}

public class EmployeeAddress
{
[Key, ForeignKey("Employee")]
    public int EmployeeID { get; set; }
    public string Address { get; set; }
    //
    //Navigation property Returns the Employee object
    public virtual Employee Employee { get; set; }
}
```

Relationship in Entity Framework using Code First

› One-to-Many

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string Name { get; set; }
    [ForeignKey("Dept")]
    public int DeptID { get;set;}
    public virtual Department Dept { get; set; }
}
public class Department
{
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Employee> Employees { get; set; }
}
```

Relationship in Entity Framework using Code First

› Many-To-Many

```
public class Employee
{
   public int EmployeeID { get; set; }
   public string Name { get; set; }
   public virtual ICollection<EmployeesInProject> EmployeesInProject { get; set; }
}
public class Project
{
   public int ProjectID { get; set; }
   public string Name { get; set; }
   public virtual ICollection<EmployeesInProject> EmployeesInProject { get; set; }
}
public class EmployeesInProject
{
   [Key]
   [Column(Order=1)]
   public int EmployeeID { get; set; }
   [Key]
   [Column(Order = 2)]
   public int ProjectID { get; set; }
   [ForeignKey("EmployeeID")]
   public Employee Employee { get; set; }
   [ForeignKey("ProjectID")]
   public Project Project { get; set; }  }
```

Query in Entity Framework

- Entity framework supports three types of queries:
  - LINQ to Entities: operates on entity framework entities to access the data from the underlying database. You can use LINQ method syntax or query syntax when querying with EDM
  - Entity SQL: It is processed by the Entity Framework's Object Services directly. It returns ObjectQuery instead of IQueryable. You need ObjectContext to create a query using Entity SQL. EntitySQL looks similar to standard SQL Queries
  - Native SQL:ou can execute native SQL queries for a relational database
- In Entity Framework Querying the database is done using the DBContext class exposes the DSet Property for the each of the entities in the model. The queries are written against the DBSet.

Query with LINQ to Entities

- Uses familiar LINQ syntax
- LINQ to Entities is a subset of LINQ which allows us to write queries against the Entity Framework conceptual models.
- LINQ to Entities translates the results to an IQueryable collection of anonymous type.

Query with LINQ to Entities

- Query syntax:
- Method syntax:

```
using (ProductContext db = new ProductContext())
{
//Query syntax
   var products = from e in db.Products
                        select e;
//Or Method syntax
var products = db.Products;
   foreach (Product p in products)
   {
      Console.WriteLine("{0} {1} ", p.ProductID, p.Name);
   }
}
```

Query with LINQ to Entities

› Use common LINQ queries
  – Select
  – ToList()
  – Where
  – Find
  – Sort
  – Contain
  – Single/ SingleOrDefault
  – First /FirstOrDefault

Loading Related Entities

- Lazy loading is the process where the Entity Framework delays the loading of an entity or collection of entities until the time application actually needs it.
  - The lazy loading is the default behavior in Entity Framework
  - You can disable the Lazy Loading:
    - for a selected Navigation Property: by making the selected property as non virtual (remove the virtual keyword)

      public ProductModel ProductModel { get; set; }
    - for the entire context: by making LazyLoadingEnabled property to false in the constructor of the DbContext class:

      this.Configuration.LazyLoadingEnabled = false;
- Rules for lazy loading:
  - *context.Configuration.ProxyCreationEnabled* should be true.
  - *context.Configuration.LazyLoadingEnabled* should be true.

Loading Related Entities

› Lazy loading:

Eg: The Product & Product model has a one to Many Relationship between them. We can navigate to the Product model from the Product using the navigation property.

```
using (ProductContext db = new ProductContext())
   {
       db.Database.Log = Console.Write;
           var product = (from p in db.Products
                   where p.ProductID == 5
                   select p).ToList();
        foreach (var p in product)
       {
          Console.WriteLine("{0} {1} {2}", p.ProductID, p.Name, p.ProductModel.Name);
       }
       Console.ReadLine();
   }
```

Loading Related Entities

- Eager loading is a process where related entities are loaded along with the target entity.
- Eager loading does this by using the Include method.
- Disable Lazing loading before doing Eager loading

Loading Related Entities

› Eg:

```
using (ProductContext db = new ProductContext())
{
    //Disable Lazy Loading
    db.Configuration.LazyLoadingEnabled = false;
    //Log SQL Command to Console
    db.Database.Log = Console.Write;
     var product = (from p in db.Products
                  .Include("ProductModel")    //ProductModel table to be included in the result
                  where p.ProductID == 5
                  select p).ToList();
     foreach (var p in product)
    {
        Console.WriteLine("{0} {1} {2}", p.ProductID, p.Name, p.ProductModel.Name);
    }
}
```

```
var product = (from p in db.Products
              .Include(p => p.ProductModel) // Using Lambda Expression instead of a string
              where p.ProductID == 5
              select p).ToList();
```

Loading Related Entities

› Eg: Eager loading multiple tables

```
using (ProductContext db = new ProductContext())
{
    db.Configuration.LazyLoadingEnabled = false;
     var product = (from p in db.Products
                  .Include(p => p.ProductModel)
                  .Include(p => p.ProductVendors)
                  where p.ProductID == 931
                  select p).ToList();
}
```

› Load multiple levels of related entities:

```
var product = (from p in db.SalesPersons
                  .Include(p => p.Employee.Person)
                  select p).ToList();
```

Loading Related Entities

- Explicit Loading is a process where, related entities are loaded with an explicit call.
- Explicit loading works very similar to Lazy Loading, but the loading of the related entities happens only after an explicit call.
- Explicit loading is done using the Load method on the related entity’s DBEntityEntry object. This object is returned by the DBContext.Entry() method.
- DbEntityEntry object provides access to information about the entity and the ability to perform actions on the Entity. Load method of the DbEntityEntry object
- With explicit loading you are not required to mark your navigation properties as virtual.

Loading Related Entities

- Eg: Explicit Loading

```
using (ProductContext db = new ProductContext())
{
    //Disable Lazy Loading
    db.Configuration.LazyLoadingEnabled = false;
    //Log Database
    db.Database.Log = Console.Write;
     //List of Products queried here.
    var product = (from p in db.Products
                orderby p.ProductID descending
                select p).Take(5).ToList();
     foreach (var p in product)
    {
       //Product model is retrieved here
       db.Entry(p).Reference(m => m.ProductModel).Load();
       Console.WriteLine("{0} {1} Product Model => {2}", p.ProductID, p.Name,
             ( p.ProductModel==null) ? "" : p.ProductModel.Name );
       Console.ReadKey();
        }
}
```

Loading Related Entities

› Eg: Loading Collections

```
using (ProductContext db = new ProductContext())
{
   var productModel = (from pm in db.ProductModels
                  select pm).Take(5).ToList();
    foreach (var pm in productModel)
   {
      db.Entry(pm).Collection(p => p.Products).Load();
      foreach (var p in pm.Products)
      {
         Console.WriteLine(" {0}", p.Name);
      }
   }
}
```

› You can filter the output of load method by using Query and Load methods

```
db.Entry(pm).Collection(p => p.Products).Query().Where(p => p.ListPrice > 0).Load();
```

Persistence in Entity Framework

- The Persistence in Entity Framework is handled by DbContext Class.
- There are two Scenario's arises when you persist the data to the database
  - Connected: This is when the context is aware of the entity is being modified.
  - Disconnected: This is when Entity is being added or edited is not attached to any context.

Persistence in Entity Framework

› Connected Scenario

```
using (EFContext db = new EFContext())
{
   var e = db.Employees;
   //Make Changes to the Employee

   //Call db.SaveChanges
   db.SaveChanges();
}
```

› Disconnected Scenario

```
using (EFContext db = new EFContext())
{
   //Query for the entity and then dispose the entity
   Var e = db.Employees;
}
 //Make Changes to the Entity
 //Create a new context
using (EFContext db = new EFContext())
(
   // Add modifed entity
   db.AddRange(e);
   //Instruct the Context about the changes that have been made to
   //Entity like Add,Edit or delete
   //Call db.SaveChanges
   db.SaveChanges();
)
```

Entities States

› An entity is always in any one of the following states.
  - Added: Added entity state indicates that the entity exists in the context, but does not exist in the database.
  - Deleted: The Deleted entity state indicates that the entity is marked for deletion, but not yet deleted from the database. It also indicates that the entity exists in the database.
  - Modified: The Modified entity state indicates that the entity is modified but not updated in the database. It also indicates that the entity exists in the database.
  - Unchanged: The property values of the entity have not been modified since it was retrieved from the database.
  - Detached: The Detached entity state indicates that the entity is not being tracked by the context.

Adding new entities

- Adding single Entity: Using Add() method
- The Add method adds the new entity in Added State. Finally, the SaveChanges method is called to insert the new entity into the database.

```
using (EFContext db = new EFContext())
{
   Department dep = new Department();
   dep.Name = "Secuirty";
   db.Departments.Add(dep);
   db.SaveChanges();
   Console.WriteLine("Department {0} ({1}) is added ", dep.Name, dep.DepartmentID);
}
```

Adding new entities

› Adding multiple entities: using the AddRange() method

```
using (EFContext db = new EFContext())
{
    db.Database.Log = Console.WriteLine;

    List<Department> deps = new List<Department>();

    deps.Add(new Department { Name = "Dept1", Descr = "" });
    deps.Add(new Department { Name = "Dept2", Descr = "" });

    db.Departments.AddRange(deps);
    db.SaveChanges();

    Console.WriteLine("{0} Departments added ", deps.Count);
    Console.ReadKey();
}
```

Adding new entities

› Adding Related Entities

```
// Adding Department & Employee
using (EFContext db = new EFContext())
{
   db.Database.Log = Console.WriteLine;
    // Add a Department
   Department dep = new Department();
   dep.Name = "Production";
   db.Departments.Add(dep);

   //Add Employee with Deparment
   Employee emp = new Employee();
   emp.FirstName = "Rahul";
   emp.LastName = "Drvid";
   emp.Department = dep;
    db.Employees.Add(emp);
    //Save
   db.SaveChanges();
    Console.WriteLine("Department {0} ({1}) is added ", dep.Name,dep.De
partmentID);
   Console.WriteLine("Employee {0} ({1}) is added in the department {2} "
, emp.FirstName, emp.EmployeeID, emp.Department.Name);
   Console.ReadKey();
}
```

Updating the Entity

› Updating the entity involves getting the entity from the database, make the necessary changes and then call the SaveChanges to persist the changes in the database.

› Updating in Connected Scenario

```
Department dep;

//Connected Scenario
using (EFContext db = new EFContext())
{
   db.Database.Log = Console.WriteLine;
   dep = db.Departments.Where(d=>d.Name=="Accounts").First();
   dep.Descr = "This is Accounts Department";
   db.SaveChanges();
   Console.WriteLine("Department {0} ({1}) is modified", dep.Name, dep.DepartmentID);
   Console.ReadKey();
}
```

Updating the Entity

› Updating in Disconnected Scenario

```
Department dep;
 //Disconnected Scenario
using (EFContext db = new EFContext())
{
    Console.Clear();
    db.Database.Log = Console.WriteLine;
    dep = db.Departments.Where(d => d.Name == "Accounts").First();
}
 dep.Descr = "Accounts Department-Disconnected Scenario";
using (EFContext db = new EFContext())
{
    db.Database.Log = Console.WriteLine;
    db.Departments.Add(dep);
    db.Entry(dep).State = System.Data.Entity.EntityState.Modified;
    db.SaveChanges();
}
```

Deleting the entity

› Deleting Entity in Connected Scenario

Call Remove() method

```
Department dep;

//Connected Scenario
using (EFContext db = new EFContext())
{
   db.Database.Log = Console.WriteLine;
   dep = db.Departments.Where(d => d.Name == "Accounts").First();
   db.Departments.Remove(dep);
   db.SaveChanges();

   Console.WriteLine("Department {0} ({1}) is Deleted ", dep.Name, dep.DepartmentID);
   Console.ReadKey();
}
```

Deleting the entity

› Deleting Entity Disconnected Scenario

```
Department dep;

//Disconnected Scenario
using (EFContext db = new EFContext())
{
    db.Database.Log = Console.WriteLine;
    dep = db.Departments.Where(d => d.Name == "Production").First();
}

using (EFContext db = new EFContext())
{
    db.Database.Log = Console.WriteLine;
    db.Departments.Add(dep);
    db.Entry(dep).State = System.Data.Entity.EntityState.Deleted;
    db.SaveChanges();
}
```

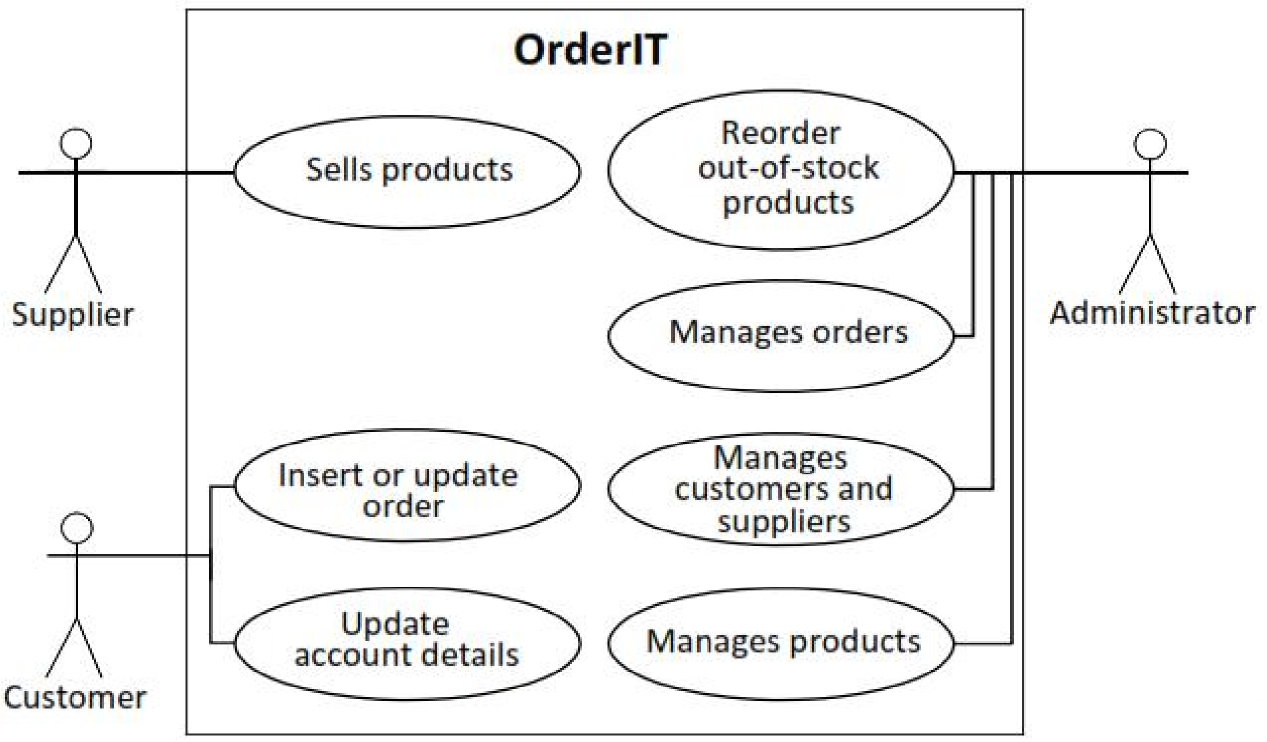
Deleting the entity

› Deleting Multiple Records

```
//Deleting Multiple Records
using (EFContext db = new EFContext())
{
   db.Database.Log = Console.WriteLine;
   List<Department> deps= db.Departments.Take(3).ToList();
   db.Departments.RemoveRange(deps);
   db.SaveChanges();
}
```

Exercises

› Query with LinQ to Entities



OrderIT
Sells products
Reorder out-of-stock products
Manages orders
Insert or update order
Manages customers and suppliers
Update account details
Manages products
Supplier
Customer
Administrator