**What is Entity Framework?**

ADO.NET entity is an ORM (object relational mapping) which creates a higher abstract object model over ADO.NET components. So rather than getting into dataset, datatables, command, and connection objects as shown in the below code, you work on higher level domain objects like customers, suppliers, etc.

DataTable table = adoDs.Tables[0];

for (int j = 0; j < table.Rows.Count; j++)

{

DataRow row = table.Rows[j];

// Get the values of the fields

string CustomerName =

(string)row["Customername"];

string CustomerCode =

(string)row["CustomerCode"];

}

Below is the code for Entity Framework in which we are working on higher level domain objects like customer rather than with base level ADO.NET components (like dataset, datareader, command, connection objects, etc.).

foreach (Customer objCust in obj.Customers)

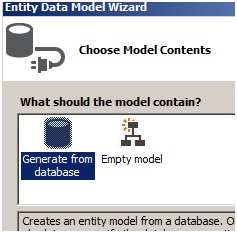
{}

**What are the benefits of using EF?**

The main and the only benefit of EF is it auto-generates code for the Model (middle layer), Data Access Layer, and mapping code, thus reducing a lot of development time.

**What are the different ways of creating these domain / entity objects?**

Entity objects can be created in two ways: from a database structure, or by starting from scratch by creating a model.

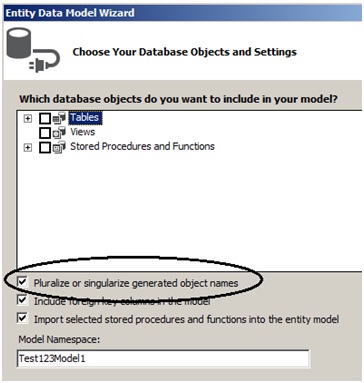


**What is pluralize and singularize in the Entity Framework dialog box?**

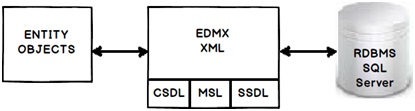
“Pluralize” and “Singularize” give meaningful naming conventions to objects. In simple words it says do you want to represent your objects with the below naming convention:

* One Customer record means “Customer” (singular).
* Lot of customer records means “Customer’s” (plural, watch the “s”)

If you select the below checkbox, Entity Framework generates a naming convention which adheres to plural and singular coding conventions.



**What is the importance of EDMX file in Entity Framework?**



EDMX (Entity Data Model XML) is an XML file which contains all the mapping details of how your objects map with SQL tables. The EDMX file is further divided into three sections: CSDL, SSDL, and MSL.

**Can you explain CSDL, SSDL and MSL sections in an EDMX file?**

* CSDL (Conceptual Schema definition language) is the conceptual abstraction which is exposed to the application.
* SSDL (Storage Schema Definition Language) defines the mapping with your RDBMS data structure.
* MSL (Mapping Schema Language) connects the CSDL and SSDL.

CSDL, SSDL and MSL are actually XML files.

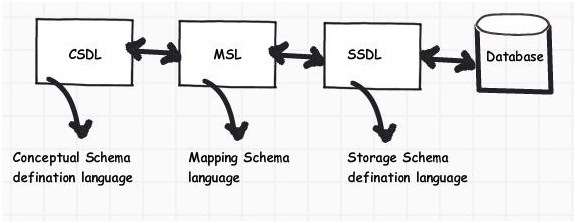


Figure: CSDL, MSL, and SSDL

**What are T4 templates?**

T4 (Text Template Transformation Toolkit) is a template based code generation engine. You can go and write C# code in T4 templates (*.tt* is the extension) files and those C# codes execute to generate the file as per the written C# logic.

For instance, the below T4 C# code:

<#@ template language="C#" #>

Hello <# Write("World!") #>

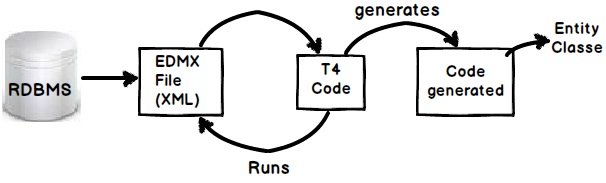
Will generate the following C# output:

Hello

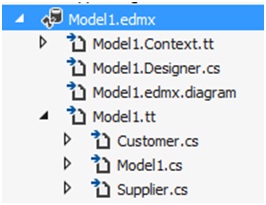
World !

**What is the importance of T4 in Entity Framework?**

T4 files are the heart of EF code generation. The T4 code templates read the EDMX XML file and generate C# behind code. This C# behind code is nothing but your entity and context classes.



If you create a project using VS 2012, you will see the following hierarchy. At the top we have the EDMX file, followed by the TT or T4 file, and then the .CS code file.



**How can we read records using Entity Framework classes?**

In order to browse through records you can create the object of the context class and inside the context class you will get the records.

For instance, in the below code snippet we are looping through a customer object collection. This customer collection is the output given by the context class CustomermytextEntities.

CustomermytestEntities obj = new CustomermytestEntities();

foreach (Customer objCust in obj.Customers)

{}

**How can we add, update, and delete using EF?**

Create the object of your entity class, add it to the data context using AddObject method, and then call the SaveChanges method.

CustomermytestEntities obj = new CustomermytestEntities();

Customer objCust = new Customer();

objCust.CustomerCode = "1001";

obj.Customers.AddObject(objCust);

obj.SaveChanges();

If you want to update, select the object, make changes to the object, and call AcceptAllChanges.

CustomermytestEntities objContext = new CustomermytestEntities();

Customer objCustomer = (Customer)objContext.Customers.FirstOrDefault();

objCustomer.CountryCode = "NEP";

objContext.AcceptAllChanges();

If you want to delete, call the DeleteObject method as shown in the below code snippet:

CustomermytestEntities objContext = new CustomermytestEntities();

Customer objCustomer = (Customer)objContext.Customers.FirstOrDefault();

objContext.DeleteObject(objCustomer);

You can see the following YouTube video which shows a simple insert, update, and delete example using Entity Framework:

**People say Entity Framework runs slow**

By default EF has lazy loading behavior. Due to this default behavior if you are loading a large number of records and especially if they have foreign key relationships, you can have performance issues. So you need to be cautious if you really need lazy loading behavior for all scenarios. For better performance, disable lazy loading when you are loading a large number of records or **use stored procedures**.

**Can you explain lazy loading in a detailed manner?**

Lazy loading is a concept where we load objects on demand rather than loading everything in one go. Consider a situation where you have 1 to many relationships between the Customer and Address objects. Now let’s say you are browsing the customer data but you do not want address data to be loaded at that moment. But the time you start accessing the address object you would like to load address data from the database.

Entity Framework has lazy loading behavior by default enabled. For instance, consider the below code. When we are doing a foreach on the Customer object, the Address object is not loaded. But the time you start doing foreach on the address collection, the Address object is loaded from SQL Server by firing SQL queries.

So in simple words, it will fire a separate query for each address record of the customer, which is definitely not good for a large number of records.

MyEntities context = new MyEntities();

var Customers = context.Customers.ToList();

foreach (Customercust in Customers) // In this line no address object loaded

{

foreach(Address add in cust.Addresses){}// Address object is loaded here

}

**How can we turn off lazy loading?**

The opposite of lazy loading is eager loading. In eager loading we load the objects beforehand. So the first thing is we need to disable lazy loading by settingLazyLoadingEnabled to false.

context.ContextOptions.LazyLoadingEnabled = false;

Now we have to explicitly tell EF what objects we want to load by using the include function. Below is a simple sample code where we tell EF to load customer as well as address objects by using the include function.

Now the customer object and the related address objects will be loaded in one query rather than multiple queries.

var employees = context.Customers.Include("Addresses").Take(5);

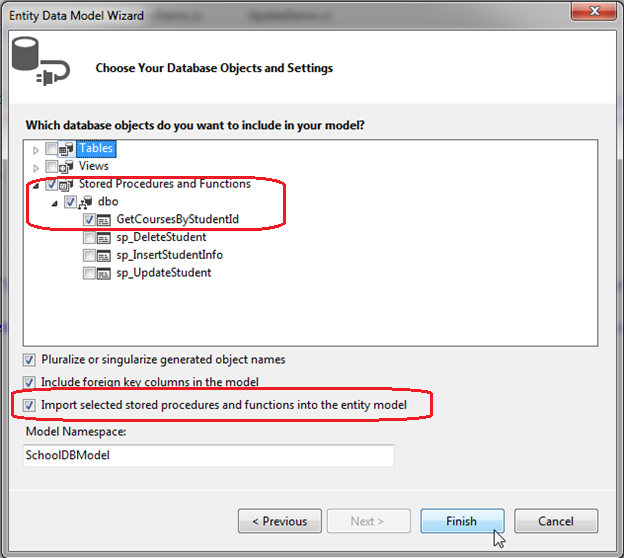
**Rules for lazy loading**

1. *context.Configuration.ProxyCreationEnabled* should be true.
2. *context.Configuration.LazyLoadingEnabled* should be true.
3. Navigation property should be defined as public, virtual. Context will **NOT** do lazy loading if the property is not defined as virtual.

**How can we use stored procedures in Entity Framework?**

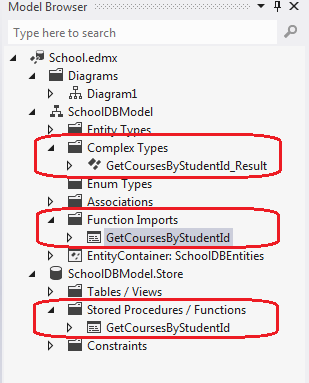
* 1. **With Database First Approach:**

**Import selected stored procedures and functions into the entity model**checkbox is selected and then click **Finish.**

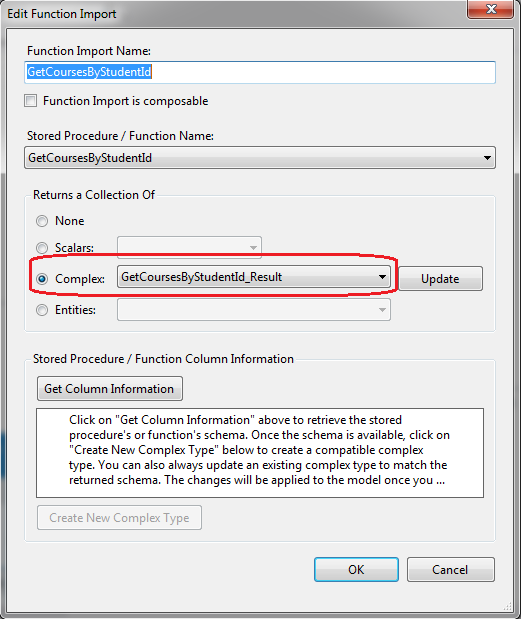


You can use stored procedure mapping details in EDMX file.

You will see the GetCoursesByStudentId stored procedure added in **Stored Procedures/Functions** and **Function Imports** shown below, with the new complex type GetCoursesByStudentId\_Result in the Model Browser. Whenever you import a stored procedure or UDF into an EDM, it creates a new complex type with the name {sp name}\_Result by default.



GetCoursesByStudentId returns the same fields defined in the Course entity. So, we don't need to add a new complex type for the GetCoursesByStudentId. You can change it by right clicking on GetCoursesByStudentId in Function Imports and selecting **Edit**. This will open Edit Function Import popup as shown below.

[](http://www.entityframeworktutorial.net/images/EF5/stored-proc-fg4.PNG)

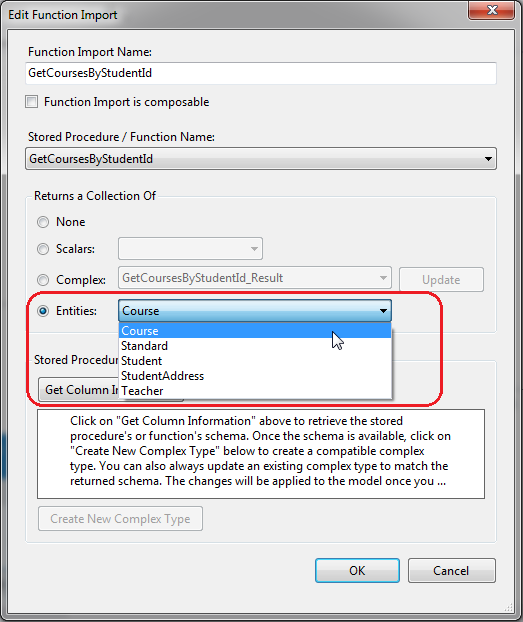
To set the Course entity as result type, select Entities and select Course from dropdown in the popup window and click OK, as shown below:

**None:** it means stored procedure will not return any value.

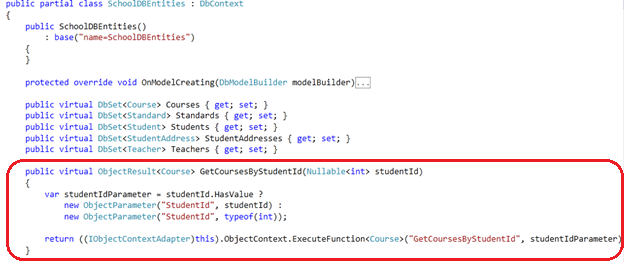
**Scalars:**it means stored procedure will return single value of selected type like binary, Boolean, byte etc.

**Complex:** It means stored procedure will return complex type which is only on conceptual model but not in database table. You can create complex type here only by first clicking on ‘Get Column Information’ which will get the schema of stored procedure and then click on ‘Create New Complex Type’ which will generate complex type.

**Entities:** it means stored procedure will return collection of selected entities.

[](http://www.entityframeworktutorial.net/images/EF5/stored-proc-fg5.PNG)

This will add the function named GetCoursesByStudentId in the context (derived from DbContext) class as shown below:

[](http://www.entityframeworktutorial.net/images/EF5/stored-proc-fg6.PNG)

Now, you can use GetCoursesByStudentId as a function and get the data, as shown below:

using (var context = new SchoolDBEntities())

{

var courses = context.GetCoursesByStudentId(1);

foreach (Course cs in courses)

Console.WriteLine(cs.CourseName);

}

The above example will execute the following statement in the database:

exec [dbo].[GetCoursesByStudentId] @StudentId=1

* 1. **With Code First Approach**

Entity Framework 6 Code-First provides the ability to create and use a stored procedure for add, update, and delete operations when the SaveChanges() method is called.

Let's use stored procedures for the CUD (Create, Update, Delete) operations for the following Student entity.

class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

public DateTime DoB { get; set; }

}

Use the MapToStoredProcedures() method to map to Student entity. EF API will create three rocedures Student\_Insert, Student\_Update and Student\_Delete for the above Student entity.

EF 6 allows you to use your own custom stored procedures and map them to an entity. You can also configure parameters mapping with entity properties.

The following example maps custom stored procedures with the Student entity.

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<Student>()

.**MapToStoredProcedures**(p=>p.Insert(sp=>sp.HasName("sp\_InsertStudent")

.Parameter(pm => pm.StudentName, "name")

.Result(rs => rs.StudentId, "Id"))

.Update(sp=>sp.HasName("sp\_UpdateStudent")

.Parameter(pm => pm.StudentName, "name"))

.Delete(sp => sp.HasName("sp\_DeleteStudent")

.Parameter(pm => pm.StudentId, "Id"))

);

}

**1 - Use Stored procedure mapping**

Below query will save data into database using stored procedure “sp\_InsertStudent”

Employee dept = new Employee() { EmpID = 1, EmpName = "Test",DepartmentID=1 };

context.employee.Add(dept);

context.SaveChanges();

# 2 - Call the stored procedure directly

List<Employee> empList = d.Database.SqlQuery<Employee>("Select \* from Employees").ToList();

context.Database.ExecuteSqlCommand("exec dob.procGetEmployees");

**What are POCO classes in Entity Framework?**

POCO means Plain Old C# Object. When EDMX creates classes, they are cluttered with a lot of entity tags. For instance, below is a simple customer class generated using Entity Framework. Many times we would like to use simple .NET classes and integrate them with Entity Framework.

Entity Framework allows this. In other words you can create a simple .NET class and use the entity context object to load your simple .NET classes.

Below is a simple class generated by EF which is cluttered with a lot of EF attributes.

[EdmEntityTypeAttribute(NamespaceName="CustomermytestModel", Name="Customer")]

[Serializable()]

[DataContractAttribute(IsReference=true)]

public partial class Customer : EntityObject

{

#region Factory Method

/// <summary>

/// Create a new Customer object.

/// </summary>

/// <param name="id" />Initial value of the Id property.

/// <param name="customerCode" />Initial value of the CustomerCode property.

/// <param name="customername" />Initial value of the Customername property.

public static Customer CreateCustomer(global::System.Int32 id,

global::System.String customerCode, global::System.String customername)

{

Customer customer = new Customer();

customer.Id = id;

customer.CustomerCode = customerCode;

customer.Customername = customername;

return customer;

}

#endregion

#region Primitive Properties

**How do we implement POCO in Entity Framework?**

To implement POCO is a three step process:

* Go to the designer and set the code generation strategy to NONE. This step means that you would be generating the classes on your own rather than relying on EF auto code generation.
* Now that we have stopped the auto generation of code, we need to create the domain classes manually. Add a class file and create the domain classes like the Customer class we created.

public class Customer

{

private string \_customerName;

public string CustomerName

{

get { return \_customerName; }

set { \_customerName = value; }

}

private int \_Customerid;

public int Customerid

{

get { return \_Customerid; }

set { \_Customerid = value; }

}

}

* Write your Context layer code inheriting from ObjectContext. This code you can copy paste from the behind code of EF, also before disabling auto-generation.

public partial class Test123Entities : ObjectContext

{

public Test123Entities()

: base("name=Test123Entities", "Test123Entities")

{

this.ContextOptions.LazyLoadingEnabled = true;

OnContextCreated();

}

partial void OnContextCreated();

public ObjectSet<Customer> Customers

{

get

{

if ((\_Customers == null))

{

\_Customers = base.CreateObjectSet<Customer>("Customers");

}

return \_Customers;

}

}

private ObjectSet<Customer> \_Customers;

public void AddToCustomers(Customer customer)

{

base.AddObject("Customers", customer);

}

}

And finally you can use the above code in your client as if you where using EF normally.

Test123Entities oContext = new Test123Entities();

List<Customer> oCustomers = oContext.Customers.ToList<Customer>();

**In POCO classes do we need EDMX files?**

Yes, you will still need EDMX files because the context object reads the EDMX files to do the mapping.

**What is Code First approach in Entity Framework?**

In Code First approach we avoid working with the Visual Designer of Entity Framework. In other words the EDMX file is excluded from the solution. So you now have complete control over the context class as well as the entity classes.

**What is the difference between POCO, Code First, and simple EF approach?**

All these three approaches define how much control you want on your Entity Framework code. Entity Framework is an OR mapper, it generates a lot of code, it creates your middle tier (Entity), and Data Access layer (Context).

But a lot of times you want to enjoy the benefits of both worlds, you want the auto-generation part to minimize your development time and you want control on the code so that you can maintain code quality.

Below is the difference table which defines each of the approaches. In simple Entity Framework, everything is auto generated and so you need the EDMX XML file as well. POCO is semi-automatic so you have full control on the entity classes but then the context classes are still generated by the EDMX file.

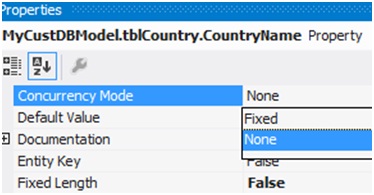
In Code First, you have complete control on how you can create the entity and context classes. Because you are going to manually create these classes, you do not have dependency on the EDMX XML file. Below is a simple table which shows the cross comparison.

|  | **EDMX** | **Entity** | **Context** |
| --- | --- | --- | --- |
| **Simple entity framework** | Needed | Auto | Auto |
| **POCO approach** | Needed | Manual | Auto |
| **Code First** | Not Needed | Manual | Manual |

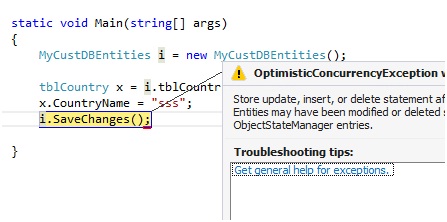
**How can we handle concurrency in Entity Framework?**

**Note**: Before this question, the interviewer can ask you about concurrency and what is pessimistic and optimistic locking. Please do refer to the ADO.NET chapter for those.

In EF, concurrency issue is resolved by using optimistic locking. Please refer to the ADO.NET chapter for what is optimistic locking and pessimistic locking? To implement optimistic locking, right click on the EDMX designer and set the concurrency mode to Fixed, as shown in the below figure.



Now whenever we have concurrency issues you should get an OptimisticConcurrencyException error as shown in the below figure. You can then put a try / catch to handle this situation.



**How can we do pessimistic locking in Entity Framework?**

We cannot do pessimistic locking using Entity Framework. You can invoke a stored procedure from Entity Framework and do pessimistic locking by setting the isolation level in the stored procedure. But directly, Entity Framework does not support pessimistic locking.

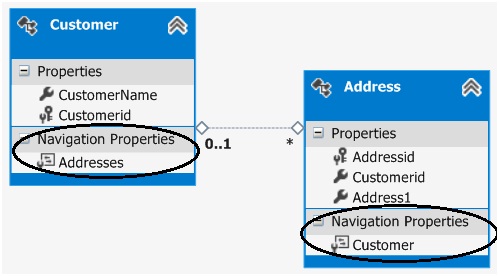
**What is client wins and store wins mode in Entity Framework concurrency?**

Client wins and store wins are actions which you would like to take when concurrency happens. In store wins / database wins, the data from the server is loaded into your entity objects. Client wins is opposite to stored wins, data from the entity object is saved to the database.

We need to use the Refresh method of the Entity Framework context and provide the RefreshMode enum values. Below is a simple code snippet which executes ClientWins.

Context.Refresh(System.Data.Objects.RefreshMode.ClientWins,Obj);

**What are scalar and navigation properties in Entity Framework?**



Scalar properties are those where actual values are contained in the entities. For example, in the above customer entity, customername and customerid are scalar properties. Normally a scalar property will map to a database field.

Navigation properties help to navigate from one entity to another entity. For instance, consider the below example in which we have two entities: Customer and Address, and a customer has multiple address objects.

Now we would like to have a facility where at any given moment we would like to browse from a given customer object to the addresses collection and from the address object to the customer.

If you open the Entity Designer, you would notice navigation properties as shown below. The navigation properties are automatically created from the primary and foreign key references.

So now because of those navigation properties, we can browse from the Customer to the Addresses object, look at the below code:

Customer Cust = oContext.Customers.ToList<Customer>()[0];

// From customer are browsing addresses

List<Address> Addresses = Cust.Addresses.ToList<Address>();

You can also do vice versa. In other words, from the Address object, you can reference the Customer object, as shown in the below code.

Address myAddress = Addresses[0];

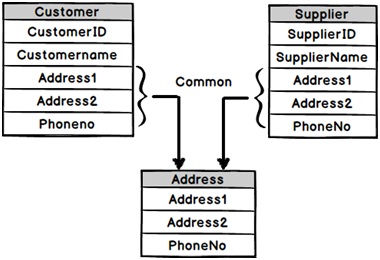
// From address we can browse customer

Customer cus = myAddress.Customer;

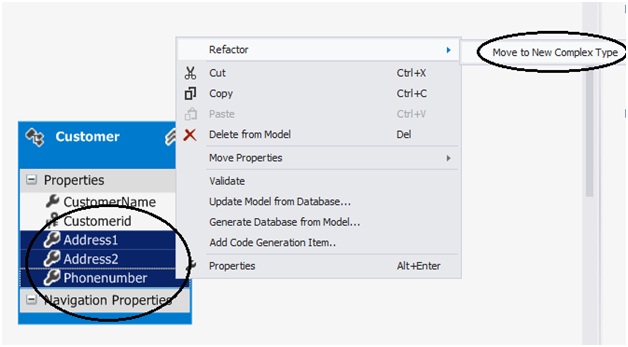
**What are complex types in Entity Framework?**

There can be situations where you have common properties across entities. For example, consider the below figure where we have Customer and Supplier entities. They have three fields in common: Address1, Address2, and PhoneNo. These fields have been duplicated both in the Customer and Supplier entities.

So to remove these duplicate and redundant fields, we can move them to a common complex type called Address. Complex types group common fields so that they can be reused across entities.



To create a complex type, select the fields which you want to group in a complex type, click on Refactor, and create the complex type. Below is a figure which shows this. Once the complex type is created, you can then reuse the complex type with other entities as well.

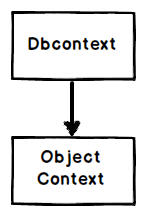


**What’s the difference between LINQ to SQL and Entity Framework?**

* LINQ to SQL is good for rapid development with SQL Server. EF is for enterprise scenarios and works with SQL Server as well as other databases.
* LINQ maps directly to tables. One LINQ entity class maps to one table. EF has a conceptual model and that conceptual model maps to the storage model via mappings. So one EF class can map to multiple tables, or one table can map to multiple classes.
* LINQ is more targeted towards rapid development while EF is for enterprise level where the need is to develop a loosely coupled framework.

**What is the difference between DbContext and ObjectContext?**

DbContext is a wrapper around ObjectContext, it’s a simplified version of ObjectContext.



As a developer you can start with DbContext as it’s simple to use. When you feel that some of the operations cannot be achieved by DbContext, you can then access ObjectContext from DbContext, as shown in the below code:

((IObjectContextAdapter)dbContext).ObjectContext

**Note**: If the interviewer asks what kind of operations are not supported in DbContext, you can excuse by saying you do not remember them up front. Wonder why sometimes interviewers ask API level questions?

**How can you enhance the performance of Entity Framework?**

To enhance the performance of Entity Framework, you have to follow the following steps

* Try to avoid to put all the DB objects into one single entity model
* Disable change tracking for entity if not needed
* Reduce response time for the first request by using pre-generating Views
* If not required try to avoid fetching all the fields
* For data manipulation select appropriate collection
* Wherever needed use compiled query
* Avoid using Views and Contains
* While binding data to grid or paging, retrieve only required no of records
* Debug and Optimize LINQ query

**Explain how you can load related entities in EF (Entity Framework)?**

You can load related entities or data in EF in three ways

* Eager Loading
* Lazy Loading
* Explicit Loading

**Explain Lazy loading, Eager Loading, and Explicit Loading?**

* **Lazy Loading:** It is a process to delay the loading of related objects until it is required.
* **Eager Loading:** It occurs when you query for an object and all of the related objects are also returned. In eager loading, related objects are loaded automatically with its parent object
* **Explicit Loading:** Explicitly loading takes place when you have disabled Lazy loading, and you still want to lazy loading. For this, we have to call the load method on the related entities.

**What is DbSet**

The DbSet class represents an entity set that can be used for create, read, update, and delete operations. The context class (derived from DbContext ) must include the DbSet type properties for the entities which **map** to**database tables** and views