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.

Hide   Copy Code

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.).

Hide   Copy Code

foreach (Customer objCust in obj.Customers)

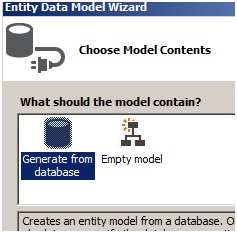
{}

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.

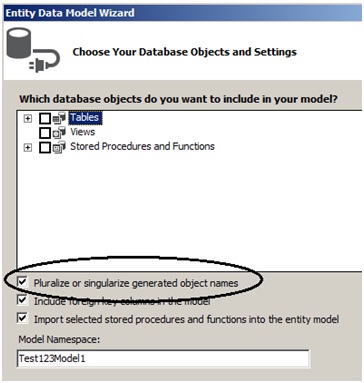


What is pluralize and singularize in the Entity Framework dialog box?

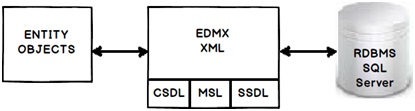
“Pluralize” and “Singularize” give meaningful naming conventions to objects. In simple words it says do you want to represent your objects with the below naming convention:

* 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?

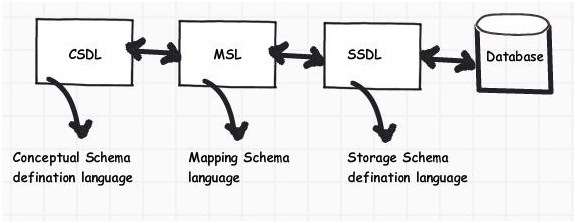


EDMX (Entity Data Model XML) is an XML file which contains all the mapping details of how your objects map with SQL tables. The EDMX file is further divided into three sections: CSDL, SSDL, and MSL.

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:

Hide   Copy Code

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

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

Will generate the following C# output:

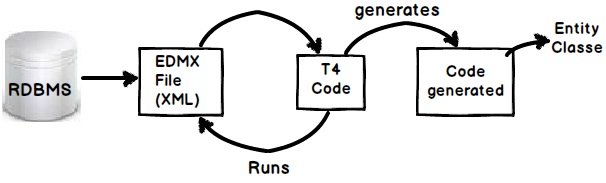
Hide   Copy Code

Hello

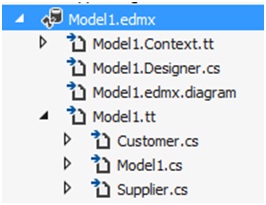
World !

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.

Hide   Copy Code

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.

Hide   Copy Code

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.

Hide   Copy Code

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:

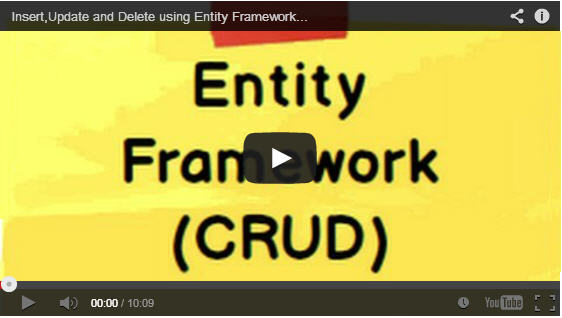
Hide   Copy Code

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:

[](http://www.youtube.com/v/b6vTIiBNcJ0)

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 doingforeach 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.

Hide   Copy Code

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 setting LazyLoadingEnabled to false.

Hide   Copy Code

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.

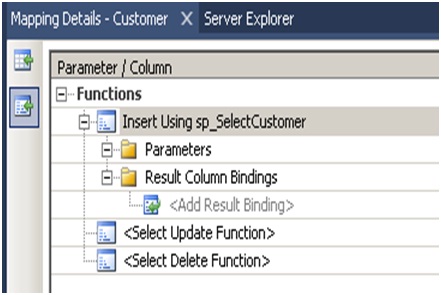
Now the customer object and the related address objects will be loaded in one query rather than multiple queries.

Hide   Copy Code

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

How can we use stored procedures in Entity Framework?

You can use stored procedure mapping details in EDMX as shown in the below figure.



***Figure: Specify stored procedures***

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.

Hide   Copy Code

[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.

Hide   Copy Code

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.

Hide   Copy Code

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.

Hide   Copy Code

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.

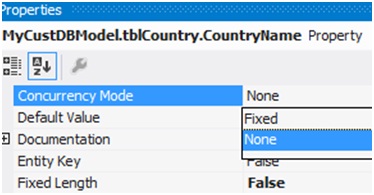
In Code First, you have complete control on how you can create the entity and context classes. Because you are going to manually create these classes, you do not have dependency on the EDMX XML file. Below is a simple table which shows the cross comparison.

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

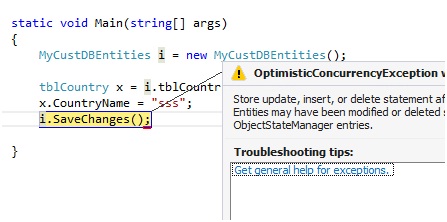
How can we handle concurrency in Entity Framework?

**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?

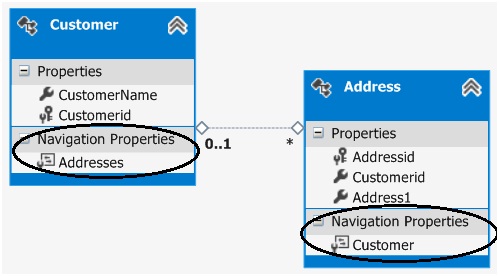
Client wins and store wins are actions which you would like to take when concurrency happens. In store wins / database wins, the data from the server is loaded into your entity objects. Client wins is opposite to stored wins, data from the entity object is saved to the database.

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.

Hide   Copy Code

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:

Hide   Copy 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.

Hide   Copy Code

Address myAddress = Addresses[0];

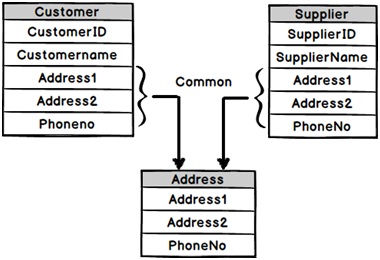
*// From address we can browse customer*

Customer cus = myAddress.Customer;

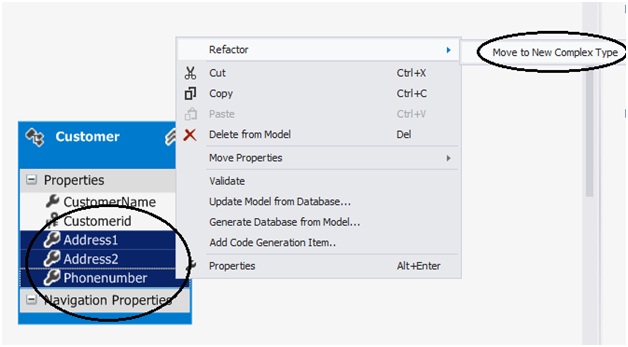
What are complex types in Entity Framework?

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 calledAddress. 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.

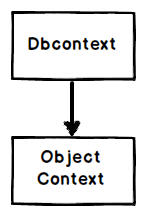


What’s the difference between LINQ to SQL and Entity Framework?

* 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:

Hide   Copy 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?

Please do visit my site [www.questpond.com](http://www.questpond.com/) which has a lot of videos on C#, .NET, EF, MVC, Design Patterns, etc.

Q1. **EDMX PART :**

SSDL (Store schema definition language):

Storage Models are our database design models, which contains database tables, views, stored procs and keys with relationships

CSDL (Conceptual schema definition language):

Conceptual Models are the model classes which contain the relationships. These are independent of the database design.

MSL (Mapping specification language):

The Mapping will have the information on how the Conceptual Models(csdl) are mapped to Storage Models.(SSDL)

**What is DB Context:** When we create a edmx file, it will have the list of entities and context class which will be derived from DB Context class.

DBContext will be responsible for Insert, Update and Delete functionalities of the entities. It acts like a bridge between the database and entity classes.

**Object Context:**

Object Context manages all the database operations, like database connection, and manages various entities of the Entity Model. DB Context is a wrapper around Object Context.

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

Conceptually, both these are here for the same reason.

* DBContext is wrapper around Object Context.
* Prior to EF 4.1, EDM used to use Object Context as base class for Context classes.
* There were some difficulties been observed in Object Context so now Db Context is introduced in EF 6.0 and this is used for all the development models –
* Database First, Model First and Code First.
* Object Context supports complied queries, but DB Context not.
* Object Context supports self-tracking entities, but DB Context does not.
* DB Context is thread safe, but Object Context not.

**T4 Templates:**

T4 Template (Text Template Transformation Toolkit) will generate the C# code based on edmx XML file. (.tt extension)

**different approaches supported in the Entity Framework to create Entity Model?**

* Database First
* Model First
* Code First

**28) What is the Database First Approach?**

This approach is suitable, if we have a database already created and ready to use it. Using the existing database, we can create the Entity Models.

**29) What is the Model First Approach?**

This approach is suitable, when we prefer to create the Entity Models first and derive the database from the Entity Models.

**30) What is the Code First Approach?**

This approach is suitable, when we prefer to create the Domain classes first and derive the database from the Domain classes.

**31) What are the advantages of Database First Approach?**

Below are the advantages of Database First Approach –

* Easy to create entity models if there is an existing database.
* Preferred approach for data intensive applications.

**32) What are the disadvantages of Database First Approach?**

Below are the disadvantages of Database First Approach –

* Once we create a edmx file from an existing database, huge pile of code is generated.
* If we want to add the additional functionality to the models generated, we need to extend the models.

**33) What are the advantages of Model First Approach?**

Below are the advantages of Model First Approach –

* Model first approach gives the flexibility to design the Entity Models independently and gives an option to improve at later stages.
* Model classes can be created by drawing it in the edmx designer, so no much of database is required.

**34) What are the advantages of Code First Approach?**

Below are the advantages of Code First Approach –

* Based on business objects we can decide the database structure.
* We can decide which classes need to be serialized and can specify the collection to eager load.
* Good for smaller applications.

**35) What are the disadvantages of Code First Approach?**

Below are the disadvantages of Code First Approach –

* All database related stuffs should be included in the code.
* For Stored Procs, we need to use the Fluent APIs to write it in code.
* Not good for data intensive application

**What are the Entity States supported in Entity Framework?**

Below are the States supported in Entities during lifetime –

* Added
* Deleted
* Modified
* Un Changed
* Detached

**How we can increase the performance of Entity Framework?**

* Disable the Change Tracking if it’s not required.
* Use the compiled query whenever required.
* Avoid using Views
* Fetch the required data from database.

\***What are the ways we can load the related entities in Entity Framework?**

**1. LAZY LOADING**

**2. Eager loading**

**3. Explicit loading**

**Lazy Loading in Entity Framework?** Lazy Loading is the default behavior of Entity Framework, wherein the dependent/related entities are loaded once they are accessed for the first time.

**example for Lazy Loading?**

Filter the Student based on Student Id.

Student s = dbContext.Students.FirstOrDefault(a => a. StudentId == sId);

Load all the Courses related to Student.

List<Course> courses = s.Courses

**Eager Loading in Entity Framework?** Eager Loading will load the dependent/related entities at once. Unlike Lazy loading, Eager loading will do only one database call and get all the dependent entities.

**Example for Eager Loading?**

Consider a same Student scenario -

Student s = dbContext. Students.Include(s => s. Courses).FirstOrDefault(s => s. StudentId == sid).

Here we are loading all data at a time

**Explicit Loading in Entity Framework:**

Be default Entity Framework supports Lazy loading, but we can disable the Lazy loading and we can still load the dependent/related entities by calling “Load” method.

Disable lazy Loading in DBCONTEXT class:

public SchoolDBEntities(): base("name=SchoolDBEntities")

{

this.Configuration.LazyLoadingEnabled = false;

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

}

**example for Explicit Loading?**

Student s = dbContext. Students.FirstOrDefault(a => a. StudentId == sId);

dbContext.Entry(s).Reference(s => s. Courses).Load();

**Which type of loading is good in which scenario?**

**Which type of loading is good in which scenario?**

* Use Eager Loading, when the relations are not too much so that we can get the data in one database query.
* Use Eager Loading, if you are sure that we are going to use the dependent/related entities.
* If there are one-to-many relations use Lazy loading and if the dependent/related entities are not required down the line.
* When Lazy Loading is turned off, use Explicit loading when you are not sure whether dependent/related entities are not required down the line

**How Entity Framework handles the Change Tracking?**

Entity Framework supports automatic change tracking of all the loaded entities through context class.

If Entity Framework need to handle Change Tracking, each entity should have Entity Key (Primary Key).

\*DbContext :

*System.Data.Entity.DbContext* DbContext is an important class in Entity Framework API. It is a bridge between your domain or entity classes and the database.

DbContext is the primary class that is responsible for interacting with the database. It is responsible for the following activities:

* **Querying:** Converts LINQ-to-Entities queries to SQL query and sends them to the database.
* **Change Tracking:** Keeps track of changes that occurred on the entities after querying from the database.
* **Persisting Data:** Performs the Insert, Update and Delete operations to the database, based on entity states.
* **Caching:** Provides first level caching by default. It stores the entities which have been retrieved during the life time of a context class.
* **Manage Relationship:** Manages relationships using CSDL, MSL and SSDL in Db-First or Model-First approach, and using fluent API configurations in Code-First approach.
* **Object Materialization:** Converts raw data from the database into entity objects

## DbContext Methods: like SaveChanges

|  |  |
| --- | --- |
| **Method** | **Usage** |
| Entry | Gets an DbEntityEntry for the given entity. The entry provides access to change tracking information and operations for the entity. |
| SaveChanges | Executes INSERT, UPDATE and DELETE commands to the database for the entities with Added, Modified and Deleted state. |
| SaveChangesAsync | Asynchronous method of SaveChanges() |
| Set | Creates a DbSet<TEntity> that can be used to query and save instances of TEntity. |
| OnModelCreating | Override this method to further configure the model that was discovered by convention from the entity types exposed in DbSet<TEntity> properties on your derived context. |

# DbSet in Entity Framework 6:

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.

The following table lists important methods of the DbSet class:

| Method Name | Return Type | Description |
| --- | --- | --- |
| Add | Added entity type | Adds the given entity to the context with the Added state. When the changes are saved, the entities in the Added states are inserted into the database. After the changes are saved, the object state changes to Unchanged.  Example:  dbcontext.Students.Add(studentEntity) |
| AsNoTracking<Entity> | DBQuery<Entity> | Returns a new query where the entities returned will not be cached in the DbContext. (Inherited from DbQuery.)  **Entities returned as AsNoTracking will not be tracked by DBContext. This will be a significant performance boost for read-only entities.**   Example:  var studentList = dbcontext.Students.AsNoTracking<Student>().ToList<Student>(); |
| Attach(Entity) | Entity which was passed as parameter | Attaches the given entity to the context in the Unchanged state   Example:  dbcontext.Students.Attach(studentEntity); |
| Create | Entity | Creates a new instance of an entity for the type of this set. This instance is not added or attached to the set. The instance returned will be a proxy if the underlying context is configured to create proxies and the entity type meets the requirements for creating a proxy.  Example:  var newStudentEntity = dbcontext.Students.Create(); |
| Find(int) | Entity type | Uses the primary key value to find an entity tracked by the context. If the entity is not in the context, then a query will be executed and evaluated against the data in the data source, and null is returned if the entity is not found in the context or in the data source. Note that the Find also returns entities that have been added to the context but have not yet been saved to the database.   Example:  Student studEntity = dbcontext.Students.Find(1); |
| Include | DBQuery | Returns the included non-generic LINQ to Entities query against a DbContext. (Inherited from DbQuery)  Example: var studentList = dbcontext.Students.Include("StudentAddress").ToList<Student>(); var studentList = dbcontext.Students.Include(s => s.StudentAddress).ToList<Student>(); |
| Remove | Removed entity | Marks the given entity as Deleted. When the changes are saved, the entity is deleted from the database. The entity must exist in the context in some other state before this method is called.  Example: dbcontext.Students.Remove(studentEntity); |
| SqlQuery | DBSqlQuery | Creates a raw SQL query that will return entities in this set. By default, the entities returned are tracked by the context; this can be changed by calling AsNoTracking on theDbSqlQuery<TEntity> returned from this method.  Example: var studentEntity = dbcontext.Students.SqlQuery("select \* from student where studentid = 1").FirstOrDefault<Student>(); |

# Execute Raw SQL Queries in Entity Framework 6

* DbSet.SqlQuery()
* DbContext.Database.SqlQuery()
* DbContext.Database.ExecuteSqlCommand()

Example: using (var ctx = new SchoolDBEntities())

{

var studentList = ctx.Students

.SqlQuery("Select \* from Students")

.ToList<Student>();

}

**Call stored Procurure from code:**

using (var db = DepartmentContext() )

{

var name = new SqlParameter("@name, txtDepartment.text.trim());

//to get this to work, you will need to change your select inside dbo.insert\_department to include name in the resultset

var department = db.Database.SqlQuery<Department>("dbo.insert\_department @name", name).SingleOrDefault();

//alternately, you can invoke SqlQuery on the DbSet itself:

//var department = db.Departments.SqlQuery("dbo.insert\_department @name", name).SingleOrDefault();

int departmentID = department.DepartmentId;

}

}

DbContext generally represents a database connection and a set of tables.

**DbSet in Entity Framework** 6. 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

**Fluent Api:**

Code first add column:

public class Empoptum

{

public int id { get; set; }

public string name { get; set; }

}

public virtual DbSet<Empoptum> Empoptums { get; set; }

static void Main(string[] args)

{

Empcodefirst str;

using (str=new Empcodefirst())

{

str.Empoptums.Add(new Empoptum { id = 1, name="sujeet1" });

str.SaveChanges();

}

Console.ReadLine();

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<Empoptum>().MapToStoredProcedures();

}

* Enable code migration updating exitimng column:
* Enable-Migrations -ContextTypeName EXEXAMPLE.Code\_first.Empcodefirst
* Or
* enable-migrations

<https://www.codeproject.com/Articles/801545/Code-First-Migrations-With-Entity-Framework>

In generated Configuration file update this.

AutomaticMigrationsEnabled = true;

AutomaticMigrationDataLossAllowed = false;

Then enter :

Update-Database -Verbose

The "-Verbose" flag specifies to show the SQL Statements being applied to the target database in the console. You get results as in the following figure in the Package Manager Console.

**Model First Example:**

<https://www.tutorialspoint.com/entity_framework/entity_model_first_approach.htm>

<http://www.entityframeworktutorial.net/model-first-with-entity-framework.aspx>

\*ObjectContext and DbContext have the capability to query and work with data as objects. Additionally Dbcontext can be represented as a combination of the Unit of Work and Repository patterns. Conceptually DbContext is the same as ObjectContext.  
  
**ObjectContext**  
  
ObjectContext is a class that manages all the database operations, like database connection, and manages various entities of the Entity Model. We can say that ObjectContext is the primary class for accessing or working together with entities that are defined in the conceptual model.  
  
ObjectContext is responsible for:

* Database connection
* It provides builtin Add, Update and Delete functions
* Object Set of every entity
* Provide State of pending changes
* It holds the changes done in entities

ObjectContext also encapsulates a few of things; they are:

* Connection to the Data Store or Database
* Metadata in the Entity Data Model (EDM)
* ObjectStateManager to track changes to the objects

**DbContext**  
DbContext is conceptually similar to ObjectContext. DbContext is nothing but a ObjectContext wrapper, we can say it is a lightweight alternative to the ObjectContext. DbContext can be used for DataBase first, code first and model first development. DbContext mainly contains a set of APIs that are very easy to use. The API is exposed by ObjectContext. These APIs also allow us to use a Code First approach that ObjectContext does not allow.   
  
**ObjectContext VS DbContext**

|  |  |
| --- | --- |
| **ObjectContext** | **DbContext** |
| ObjectContext class is part of the core Entity Framework API, that allows us to perform queries, change and track updates of the Database by using strongly typed entity classes. | The DbContext class can be described as a wrapper of ObjectContext. It exposes the most commonly used features of ObjectContext. |
| ObjectContext supports Compiled Queries. | DbContext does not support Compiled Queries. |
| ObjectContext supports self-tracking of Entities | DbContext does not support self-tracking of Entities. |
| ObjectContext is only useful in Model First and Database First approaches | DbContext is useful in Model First, Database First approach as well as Code First approach. |
| ObjectContext can be used by Entity Framework 4.0 and below. | DBContext can be used by Entity Framework 4.1 and above. |
| The ObjectContext class is not thread-safe. | Any public static (C#) or Shared (Visual Basic) members of DbContext are thread-safe. Any instance members are not guaranteed to be thread safe. |

There are also many methods in common in both ObjectContext and DbContext. For example ObjectContext has a direct method to execute a query against a database whereas the DbContext.DataBase class contains the SQLQuery method to obtain the same result.  
  
**Example**

// using ObjectContext (EF4.0)  
using (Entities context = new Entities())  
{  
        IEnumerable<EmployeeDetails> empDetails  =  context.ExecuteStoreQuery<EmployeeDetails>      
                                                                                            ("exec GetEmployeeData").ToList();  
}

// using DBContext (EF 4.1 and above)  
using (Entities context = new Entities())  
{  
        IEnumerable<EmployeeDetails> empDetails  =  context. Database.SqlQuery  
                                                                      < EmployeeDetails >("exec GetEmployeeData ", null).ToList();  
}  
  
DbContext also has some sets of methods that are not available with ObjectContext, like Dbset.Find, DbSet.Local etcetera.  
  
Some of the ObjectContext methods are also renamed. For example ObjectContext has methods like AddObject, DeleteObject And Attech on ObjectSet<T>, in DbContext, they are named Add, Remove and Attach on DbSet<TEntity>.  
  
**Conclusion**  
  
Dbcontext can be defined as a lightweight version of the ObjectContext or we can say Dbcontext is a wrapper of ObjectContext and exposes only the common features that are really required in programming.  
We can also get a reference to the ObjectContext from then DbContext to use those features that are only supported in ObjectContext.  
  
The following code could help to get an ObjectContext Object from an existing DbContext Object.  
  
public class EntityDBContext: DbContext, IObjectContextAdapter  
{  
   ObjectContext IObjectContextAdapter.ObjectContext  
   {  
        get   
        {   
              var objectContext = (this as IObjectContextAdapter)  
              if(objectContext != null)  
                return (this as IObjectContextAdapter).ObjectContext;   
              else   
                return null;  
        }  
   }  
}  
  
Finally, DbContext is not a replacement of ObjectContext, but it is a simple alternative that builds on ObjectContext.   
  
Hope this will help to understand the differences between ObjectContext and DbContext.

\*

\*

**DbContext corresponds to your database (or a collection of tables and views in your database) whereas a DbSet corresponds to a table or view in your database. So it makes perfect sense that you will get a combination of both!**