Questions for full detailed answers

1. **What are the different types of table relations?**

In relational databases, relationships between the tables are defined through foreign keys. A foreign key (FK) is a column or combination of columns that is used to establish and enforce a link between the data in two tables. There are three types of relationships: one-to-one, one-to-many, and many-to-many. In a one-to-many relationship, the foreign key is defined on the table that represents the many end of the relationship. The many-to-many relationship involves defining a third table (called a junction or join table), whose primary key is composed of the foreign keys from both related tables. In a one-to-one relationship, the primary key acts additionally as a foreign key and there is no separate foreign key column for either table. In Entity Framework, an entity can be related to other entities through an association (relationship). Each relationship contains two ends that describe the entity type and the multiplicity of the type (one, zero-or-one, or many) for the two entities in that relationship. The relationship may be governed by a referential constraint, which describes which end in the relationship is a principal role and which is a dependent role.

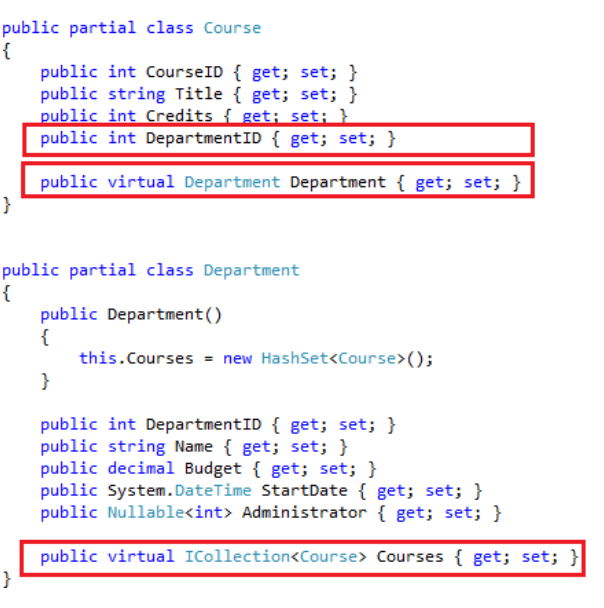
Navigation properties provide a way to navigate an association between two entity types. Every object can have a navigation property for every relationship in which it participates. Navigation properties allow you to navigate and manage relationships in both directions, returning either a reference object (if the multiplicity is either one or zero-or-one) or a collection (if the multiplicity is many). You may also choose to have one-way navigation, in which case you define the navigation property on only one of the types that participates in the relationship and not on both.

It is recommended to include properties in the model that map to foreign keys in the database. With foreign key properties included, you can create or change a relationship by modifying the foreign key value on a dependent object. This kind of association is called a foreign key association.

When foreign key columns are not included in the model, the association information is managed as an independent object. Relationships are tracked through object references instead of foreign key properties. This type of association is called an independent association. The most common way to modify an independent association is to modify the navigation properties that are generated for each entity that participates in the association.

You can choose to use one or both types of associations in your model. However, if you have a pure many-to-many relationship that is connected by a join table that contains only foreign keys, the EF will use an independent association to manage such many-to-many relationship.

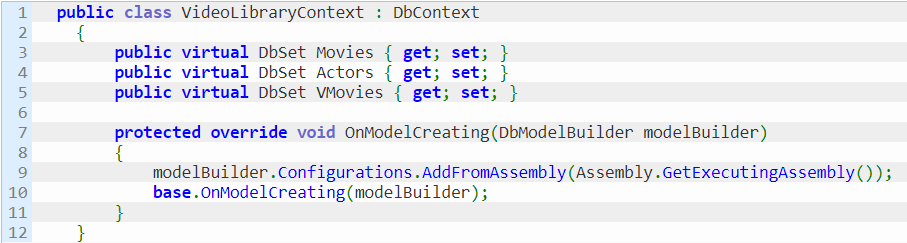
The following image shows a model containing two entities that participate in one-to-many relationship. Both entities have navigation properties. Course is the depend entity and has the DepartmentID foreign key property defined.

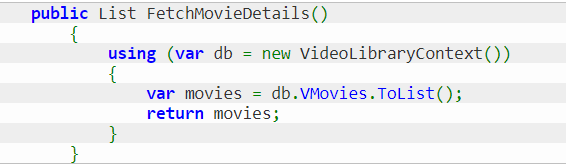


1. **When and why do we use select, where, order by and etc. on database sets?**

Collection of single entity type. Set operations: Add, Attach, Remove, Find. Use with DbContext to query database

We use linq queries to fetch data from our database and display it for the client. Based on the particular query we can take different types of data from the Db sets.





1. **What are transactions and why do we use them?**

A transaction is a group of one or more database statements that are either wholly committed or wholly rolled back. Each transaction is atomic, consistent, isolated, and durable (ACID). If the transaction succeeds, all statements within it are committed. If the transaction fails, that is at least one of the statements in the group fails, then the entire group is rolled back.

1. **How do we achieve transactions in Entity Framework?**

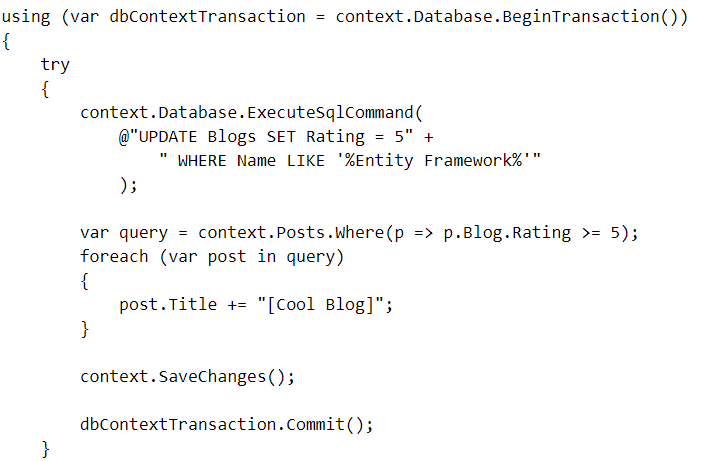
In all versions of Entity Framework, whenever you execute SaveChanges() to insert, update or delete on the database the framework will wrap that operation in a transaction. This transaction lasts only long enough to execute the operation and then completes. When you execute another such operation a new transaction is started.

Starting with EF6 Database.ExecuteSqlCommand() by default will wrap the command in a transaction if one was not already present. There are overloads of this method that allow you to override this behavior if you wish. Also in EF6 execution of stored procedures included in the model through APIs such as ObjectContext.ExecuteFunction() does the same (except that the default behavior cannot at the moment be overridden).

In either case, the isolation level of the transaction is whatever isolation level the database provider considers its default setting. By default, for instance, on SQL Server this is READ COMMITTED.

Entity Framework does not wrap queries in a transaction.

* Database.BeginTransaction() : An easier method for a user to start and complete transactions themselves within an existing DbContext – allowing several operations to be combined within the same transaction and hence either all committed or all rolled back as one. It also allows the user to more easily specify the isolation level for the transaction. It has two overrides – one which takes an explicit IsolationLevel and one which takes no arguments and uses the default IsolationLevel from the underlying database provider. Both overrides return a DbContextTransaction object which provides Commit() and Rollback() methods which perform commit and rollback on the underlying store transaction. The DbContextTransaction is meant to be disposed once it has been committed or rolled back. One easy way to accomplish this is the using(…) {…} syntax which will automatically call Dispose() when the using block completes:



Beginning a transaction requires that the underlying store connection is open. So calling Database.BeginTransaction() will open the connection if it is not already opened. If DbContextTransaction opened the connection then it will close it when Dispose() is called.

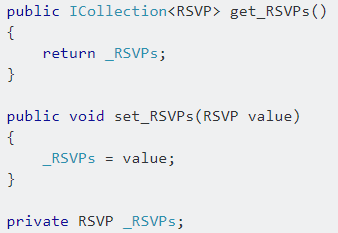
* Database.UseTransaction() : which allows the DbContext to use a transaction which was started outside of the Entity Framework. Sometimes you would like a transaction which is even broader in scope and which includes operations on the same database but outside of EF completely. To accomplish this you must open the connection and start the transaction yourself and then tell EF a) to use the already-opened database connection, and b) to use the existing transaction on that connection. To do this you must define and use a constructor on your context class which inherits from one of the DbContext constructors which take i) an existing connection parameter and ii) the contextOwnsConnection boolean. Then you are free to execute database operations either directly on the SqlConnection itself, or on the DbContext. All such operations are executed within one transaction. You take responsibility for committing or rolling back the transaction and for calling Dispose() on it, as well as for closing and disposing the database connection. You can pass null to Database.UseTransaction() to clear Entity Framework’s knowledge of the current transaction. Entity Framework will neither commit nor rollback the existing transaction when you do this, so use with care and only if you’re sure this is what you want to do.

1. **Where do we use virtual classes in Entity Framework DbContext Models and why?**

It allows the Entity Framework to create a proxy around the virtual property so that the property can support lazy loading and more efficient change tracking. By "create a proxy around" I'm referring specifically to what the Entity Framework does. The Entity Framework requires your navigation properties to be marked as virtual so that lazy loading and efficient change tracking are supported. The Entity Framework uses inheritance to support this functionality, which is why it requires certain properties to be marked virtual in your base class POCOs. It literally creates new types that derive from your POCO types. So your POCO is acting as a base type for the Entity Framework's dynamically created subclasses. That's what I meant by "create a proxy around". The dynamically created subclasses that the Entity Framework creates become apparent when using the Entity Framework at runtime, not at static compilation time. And only if you enable the Entity Framework's lazy loading or change tracking features. If you opt to never use the lazy loading or change tracking features of the Entity Framework (which is not the default) then you needn't declare any of your navigation properties as virtual. You are then responsible for loading those navigation properties yourself, either using what the Entity Framework refers to as "eager loading", or manually retrieving related types across multiple database queries. You can and should use lazy loading and change tracking features for your navigation properties in many scenarios though.

If you were to create a standalone class and mark properties as virtual, and simply construct and use instances of those classes in your own application, completely outside of the scope of the Entity Framework, then your virtual properties wouldn't gain you anything on their own. Properties such as: 

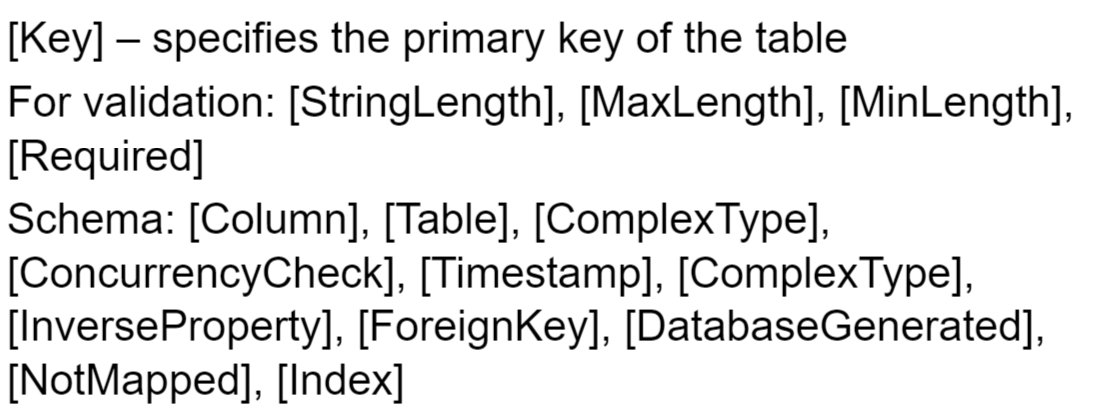
Are not fields and should not be thought of as such. These are called getters and setters and at compilation time, they are converted into methods.



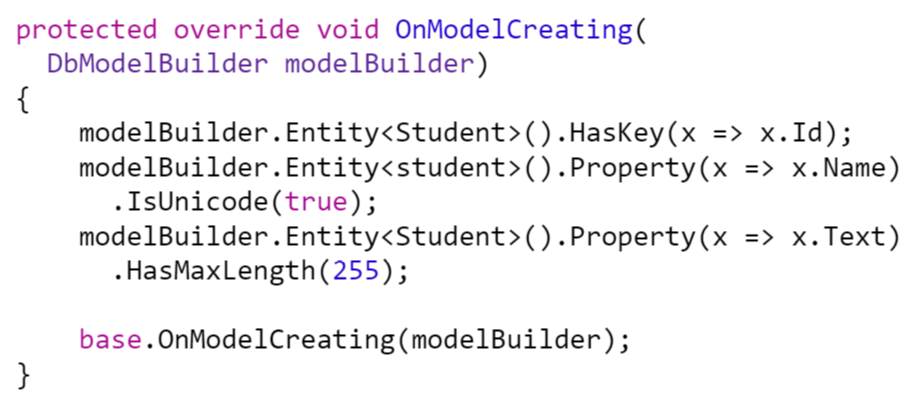
That's why they're marked as virtual for use in the Entity Framework, it allows the dynamically created classes to override the internally generated get and set functions. If your navigation property getter/setters are working for you in your Entity Framework usage, try revising them to just properties, recompile, and see if the Entity Framework is able to still function properly:

1. **How can we enforce number of characters (minimum or maximum) for a field, or any other validation?**

There is a bunch of data annotation attributes in System.ComponentModel.DataAnnotations



Or we can use FluentAPI - By overriding OnModelCreating method in DbContext class we can specify mapping configurations.



1. **Can we do an insertion and deletion at the same block of code and then apply save changes to them in a statement?**

No, because when we insert data in the DB we call on SaveChanges() which causes the Entity Framework to create SQL statements to update the database row. If we first insert and then delete than actually the data that we should have inserted isn’t really in the DB so it cannot be deleted.

1. **If we have not followed the proper conventions set by EF, then how we can mark a property as primary key?**

The proper conventions are naming the property Id, using an attribute [Key] or using Fluent API.

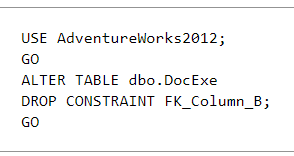
1. **What is the proper way of handing data deletion and why?**

The proper way is marking the data as “deleted” and not really deleting it since it could be useful or wanted after a certain period of time.

1. **How do we delete entities with foreign keys or used primary keys?**

**First we delete the children possessing the foreign key as a primary key and then we proceed with the current entity.**

The constraint for the foreign key should be dropped.



Questions you should have general knowledge of:

1. What is is a Trigger?

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)

A database definition (DDL) statement (CREATE, ALTER, or DROP).

A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

Generating some derived column values automatically

Enforcing referential integrity

Event logging and storing information on table access

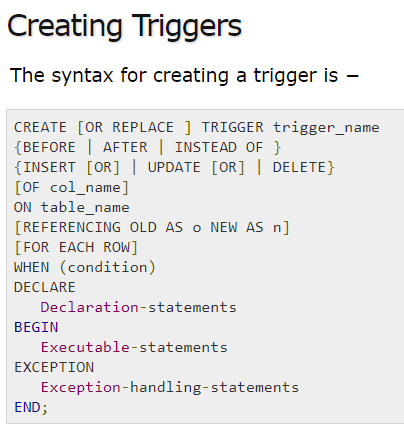
Auditing

Synchronous replication of tables

Imposing security authorizations

Preventing invalid transactions

1. What types of Triggers do you know?



CREATE [OR REPLACE] TRIGGER trigger\_name − Creates or replaces an existing trigger with the trigger\_name.

{BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.

{INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.

[OF col\_name] − This specifies the column name that will be updated.

[ON table\_name] − This specifies the name of the table associated with the trigger.

[REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

[FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

1. What are the benefits of Stored Procedures in MSSQL?

Stored procedures are a batch of SQL statements that can be executed in a couple of ways. A benefit of stored procedures is that you can centralize data access logic into a single place that is then easy for DBA's to optimize. Stored procedures also have a security benefit in that you can grant execute rights to a stored procedure but the user will not need to have read/write permissions on the underlying tables. This is a good first step against SQL injection.

Stored procedures do come with downsides, basically the maintenance associated with your basic CRUD operation. Let's say for each table you have an Insert, Update, Delete and at least one select based on the primary key, that means each table will have 4 procedures. Now take a decent size database of 400 tables, and you have 1600 procedures! And that's assuming you don't have duplicates which you probably will.

1. What are the different types of locking we can experience in MSSQL?

Execution plan retention and reuse

Query auto-parameterization

Encapsulation of business rules and policies

Application modularization

Sharing of application logic between applications

Access to database objects that is both secure and uniform

Consistent, safe data modification

Network bandwidth conservation

Support for automatic execution at system start-up

Enhanced hardware and software capabilities

Improved security

Reduced development cost and increased reliability

Centralized security, administration, and maintenance for common routines

1. What are complex types in Entity Framework? Complex types are non-scalar properties of entity types that enable scalar properties to be organized within entities. Like entities, complex types consist of scalar properties or other complex type properties. When you work with objects that represent complex types, be aware of the following: Complex types do not have keys and therefore cannot exist independently. Complex types can only exist as properties of entity types or other complex types.

Complex types cannot participate in associations and cannot contain navigation properties.

Complex type properties cannot be null. An \*\*InvalidOperationException \*\*occurs when DbContext.SaveChanges is called and a null complex object is encountered. Scalar properties of complex objects can be null.

Complex types cannot inherit from other complex types.

You must define the complex type as a class.

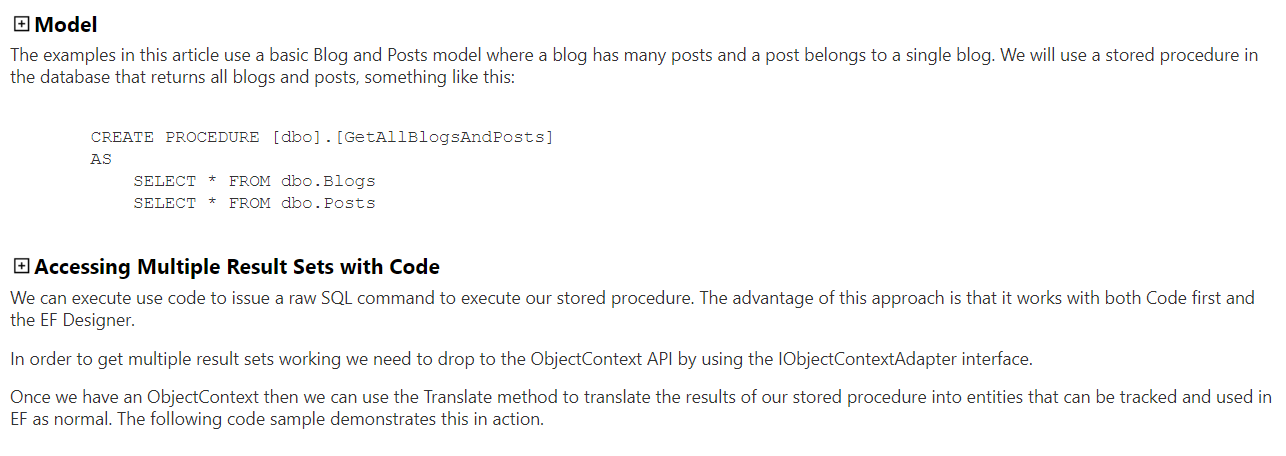
EF detects changes to members on a complex type object when DbContext.DetectChanges is called. Entity Framework calls DetectChanges automatically when the following members are called: DbSet.Find, DbSet.Local, DbSet.Remove, DbSet.Add, DbSet.Attach, DbContext.SaveChanges, DbContext.GetValidationErrors, DbContext.Entry, DbChangeTracker.Entries.

1. Can we access a model in Entity Framework without primary key defined?

No you can't because Entity Framework needs to know the key to keep track on the object when you make an update or delete operation.

1. How can we use Stored Procedures in Entity Framework?You can call a stored procedure in your DbContext class as follows.





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

hey are somewhat similar, and can be used in a very similar way, code-wise, but they have some important differences. Note that "LINQ" is not the same thing as "LINQ to SQL"; the EF also uses LINQ. Some notable differences are:

LINQ to SQL is largely SQL Server only, not so much by design as by implementation. The EF is designed to support, and does support, multiple DBs, if you have a compatible ADO.NET provider.

Out of the box, LINQ to SQL has a very poor story for DB metadata changes. You have to regenerate parts of your model from scratch, and you lose customizations.

The EF supports model features like many-to-many relationships and inheritance. LINQ to SQL does not directly support these.

In .NET 3.5, LINQ to SQL had much better support for SQL-Server-specific functionality than the EF. This is mostly not true in .NET 4; they're fairly similar in that respect.

The EF lets you choose Model First, DB First, or Code First modeling. LINQ to SQL, out of the box, really only supports DB First.

1. How can we do pessimistic locking in Entity Framework?

use transactions (not necessarily serializable, cause it will lead to poor perf.) - readcommitted is ok to use...but depends...

do your changes, call dbcontext.savechanges()

do lock your table - execute T-SQL manually, or feel free to use the code att. below.

the given T-SQL command with the hints will keep that database locked until the duration of the given transaction.

there's one thing you need to take care: your loaded entities might be obsolete at the point you do the lock, so all entities from the locked table should be re-fetched (reloaded).

515

down vote

accepted

Optimistic Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.

If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.

This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks. To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection. Optimistic assumes that nothing's going to change while you're reading it.

Pessimistic assumes that something will and so locks it.

If it's not essential that the data is perfectly read use optimistic. You might get the odd 'dirty' read - but it's far less likely to result in deadlocks and the like.

Most web applications are fine with dirty reads - on the rare occasion the data doesn't exactly tally the next reload does.

For exact data operations (like in many financial transactions) use pessimistic. It's essential that the data is accurately read, with no un-shown changes - the extra locking overhead is worth it.

Oh, and Microsoft SQL server defaults to page locking - basically the row you're reading and a few either side. Row locking is more accurate but much slower. It's often worth setting your transactions to read-committed or no-lock to avoid deadlocks while reading.

1. What is the difference between DbContext and ObjectContext?

https://www.c-sharpcorner.com/UploadFile/ff2f08/objectcontext-vs-dbcontext/