**Note** It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about how to implement repositories, see the [ASP.NET Data Access Content Map](https://www.asp.net/whitepapers/aspnet-data-access-content-map).

**Security Note:** The ValidateAntiForgeryToken attribute helps prevent [cross-site request forgery](https://www.asp.net/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages) attacks. It requires a corresponding Html.AntiForgeryToken() statement in the view, which you'll see later.

The Bind attribute is one way to protect against *over-posting* in create scenarios. For example, suppose the Student entity includes a Secret property that you don't want this web page to set.

public class Student

{

public int ID { get; set; }

public string LastName { get; set; }

public string FirstMidName { get; set; }

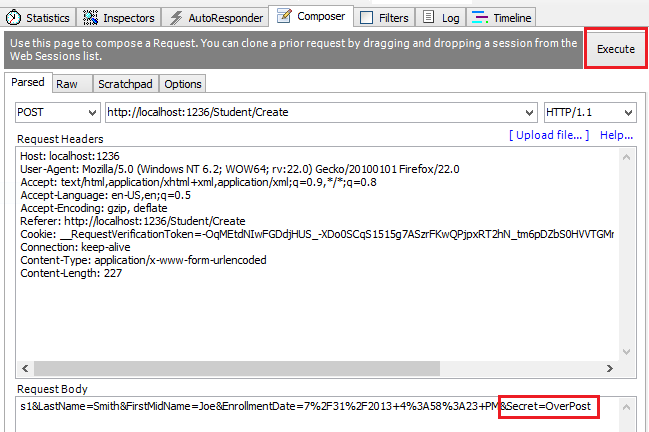
public DateTime EnrollmentDate { get; set; }

public string Secret { get; set; }

public virtual ICollection<Enrollment> Enrollments { get; set; }

}

Even if you don't have a Secret field on the web page, a hacker could use a tool such as [fiddler](http://fiddler2.com/home), or write some JavaScript, to post a Secret form value. Without the [Bind](http://msdn.microsoft.com/en-us/library/system.web.mvc.bindattribute(v=vs.108).aspx) attribute limiting the fields that the model binder uses when it creates a Student instance*,* the model binder would pick up that Secret form value and use it to create the Student entity instance. Then whatever value the hacker specified for the Secret form field would be updated in your database. The following image shows the fiddler tool adding the Secret field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the Secret property of the inserted row, although you never intended that the web page be able to set that property.

It's a security best practice to use the Include parameter with the Bind attribute to *whitelist*fields. It's also possible to use the Exclude parameter to *blacklist*fields you want to exclude. The reason Include is more secure is that when you add a new property to the entity, the new field is not automatically protected by an Exclude list.

You can prevent overposting in edit scenarios is by reading the entity from the database first and then calling TryUpdateModel, passing in an explicit allowed properties list. That is the method used in these tutorials.

An alternative way to prevent overposting that is preferrred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. **Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as [AutoMapper](http://automapper.org/). Use db.Entry on the entity instance to set its state to Unchanged, and then set Property("PropertyName").IsModified to true on each entity property that is included in the view model. This method works in both edit and create scenarios.**

Entity States and the Attach and SaveChanges Methods

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the SaveChanges method. For example, when you pass a new entity to the [Add](http://msdn.microsoft.com/en-us/library/system.data.entity.dbset.add(v=vs.103).aspx) method, that entity's state is set to Added. Then when you call the [SaveChanges](http://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext.savechanges(v=VS.103).aspx) method, the database context issues a SQL INSERTcommand.

An entity may be in one of the[following states](http://msdn.microsoft.com/en-us/library/system.data.entitystate.aspx):

* Added. The entity does not yet exist in the database. The SaveChanges method must issue an INSERT statement.
* Unchanged. Nothing needs to be done with this entity by the SaveChanges method. When you read an entity from the database, the entity starts out with this status.
* Modified. Some or all of the entity's property values have been modified. The SaveChanges method must issue an UPDATE statement.
* Deleted. The entity has been marked for deletion. The SaveChanges method must issue a DELETE statement.
* Detached. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. In a desktop type of application, you read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to Modified. Then when you call SaveChanges, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

The disconnected nature of web apps doesn't allow for this continuous sequence. The [DbContext](http://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext(v=VS.103).aspx) that reads an entity is disposed after a page is rendered. When the HttpPost Edit action method is called,  a new request is made and you have a new instance of the[DbContext](http://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext(v=VS.103).aspx), so you have to manually set the entity state to Modified. Then when you call SaveChanges, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want the SQL Update statement to update only the fields that the user actually changed, you can save the original values in some way (such as hidden fields) so that they are available when the HttpPost Edit method is called. Then you can create a Student entity using the original values, call the Attach method with that original version of the entity, update the entity's values to the new values, and then call SaveChanges. For more information, see [Entity states and SaveChanges](http://msdn.microsoft.com/en-us/data/jj592676) and [Local Data](http://msdn.microsoft.com/en-us/data/jj592872) in the MSDN Data Developer Center.

## Closing Database Connections

To close database connections and free up the resources they hold as soon as possible, dispose the context instance when you are done with it. That is why the scaffolded code provides a [Dispose](http://msdn.microsoft.com/en-us/library/system.idisposable.dispose(v=vs.110).aspx) method at the end of the StudentController class in StudentController.cs, as shown in the following example:

protected override void Dispose(bool disposing)

{

db.Dispose();

base.Dispose(disposing);

}

The base Controller class already implements the IDisposable interface, so this code simply adds an override to the Dispose(bool) method to explicitly dispose the context instance.

## Handling Transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call SaveChanges, the Entity Framework automatically makes sure that either all of your changes succeed or all fail. If some changes are done first and then an error happens, those changes are automatically rolled back.

IQueryable vs IEnumerable

Yes, both will give you deferred execution.

The difference is that IQueryable<T> is the interface that allows LINQ-to-SQL (LINQ.-to-anything really) to work. So if you further refine your query on an IQueryable<T>, that query will be executed in the database, if possible.

For the IEnumerable<T> case, it will be LINQ-to-object, meaning that all objects matching the original query will have to be loaded into memory from the database.

In code:

IQueryable<Customer> custs = ...;

// Later on...

var goldCustomers = custs.Where(c => c.IsGold);

That code will execute SQL to only select gold customers. The following code, on the other hand, will execute the original query in the database, then filtering out the non-gold customers in the memory:

IEnumerable<Customer> custs = ...;

// Later on...

var goldCustomers = custs.Where(c => c.IsGold);

This is quite an important difference, and working on IQueryable<T> can in many cases save you from returning too many rows from the database. Another prime example is doing paging: If you use Take and Skip on IQueryable, you will only get the number of rows requested; doing that on an IEnumerable<T> will cause all of your rows to be loaded in memory.

 If you want repeated filtering on your original result (several end results). Doing that on the IQueryable interface will make several roundtrips to the database, where as doing it on IEnumerable will do the filtering in the memory, making it faster (unless the amount of data is HUGE) –

The top answer is good but it doesn't mention expression trees which explain "how" the two interfaces differ. Basically, there are two identical sets of LINQ extensions. Where(), Sum(), Count(), FirstOrDefault(), etc all have two versions: one that accepts functions and one that accepts expressions.

* The IEnumerable version signature is: Where(Func<Customer, bool> predicate)
* The IQueryable version signature is: Where(Expression<Func<Customer, bool>> predicate)

You've probably been using both of those without realizing it because both are called using identical syntax:

e.g. Where(x => x.City == "<City>") works on both IEnumerable and IQueryable

* When using Where() on an IEnumerable collection, the compiler passes a compiled function to Where()
* When using Where() on an IQueryable collection, the compiler passes an expression tree to Where(). An expression tree is like the reflection system but for code. The compiler converts your code into a data structure that describes what your code does in a format that's easily digestible.

**Why bother with this expression tree thing? I just want Where() to filter my data.** The main reason is that both the EF and Linq2SQL ORMs can convert expression trees directly into SQL where your code will execute much faster.

**Oh, that sounds like a free performance boost, should I use AsQueryable() all over the place in that case?** No IQueryable is only useful if the underlying data provider can do something with it. Converting something like a regular List to IQueryable will not give you any benefit.

Both will give you deferred execution, yes.

As for which is preferred over the other, it depends on what your underlying datasource is.

Returning an IEnumerable will automatically force the runtime to use LINQ to Objects to query your collection.

Returning an IQueryable (which implements IEnumerable, by the way) provides the extra functionality to translate your query into something that might perform better on the underlying source (LINQ to SQL, LINQ to XML, etc.).

IEnumerable<T> and IQueryable<T> are the two most used terms of any LINQ discussion. What I am trying to here is that I am trying to simplify the two interfaces depending on their behavior. In LINQ world we generally have few providers available within .NET Framework, like LINQ to Object, LINQ to SQL, LINQ to XML.

It is a statement that every LINQ statement returns IEnumerable<T>. IEnumerable works in steps. Meaning, when you write,

var q = from a in b

where a > 5

select a;

It creates a list out “b” depending on “where” then it creates another list for “select”. This is the behavior of LINQ to Object and LINQ to XML.

[](https://msdnshared.blob.core.windows.net/media/TNBlogsFS/BlogFileStorage/blogs_msdn/wriju/WindowsLiveWriter/LINQIEnumerableTandIQueryableT_A403/image_4.png)

When you use LINQ to SQL it uses IQueryable<T>. This interface inherits from IEnumerable<T> but typically any LINQ to SQL generates T-SQL at the backend to be able to get the data for us. This evaluate and generates the query at one shot and gives us the whole data.

[](https://msdnshared.blob.core.windows.net/media/TNBlogsFS/BlogFileStorage/blogs_msdn/wriju/WindowsLiveWriter/LINQIEnumerableTandIQueryableT_A403/image_2.png)

**Note** In many cases you can call the same method either on an Entity Framework entity set or as an extension method on an in-memory collection. The results are normally the same but in some cases may be different.

For example, the .NET Framework implementation of the Contains method returns all rows when you pass an empty string to it, but the Entity Framework provider for SQL Server Compact 4.0 returns zero rows for empty strings. Therefore the code in the example (putting the Where statement inside an if statement) makes sure that you get the same results for all versions of SQL Server. Also, the .NET Framework implementation of the Contains method performs a case-sensitive comparison by default, but Entity Framework SQL Server providers perform case-insensitive comparisons by default. Therefore, calling the ToUpper method to make the test explicitly case-insensitive ensures that results do not change when you change the code later to use a repository, which will return an IEnumerable collection instead of an IQueryable object. (When you call the Contains method on an IEnumerable collection, you get the .NET Framework implementation; when you call it on an IQueryable object, you get the database provider implementation.)

Null handling may also be different for different database providers or when you use an IQueryable object compared to when you use an IEnumerable collection. For example, in some scenarios a Where condition such as table.Column != 0 may not return columns that have null as the value. For more information, see [Incorrect handling of null variables in 'where' clause](http://data.uservoice.com/forums/72025-entity-framework-feature-suggestions/suggestions/1015361-incorrect-handling-of-null-variables-in-where-cl).

The default [BeginForm](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.formextensions.beginform(v=vs.108).aspx) submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The [W3C guidelines for the use of HTTP GET](http://www.w3.org/2001/tag/doc/whenToUseGet.html)recommend that you should use GET when the action does not result in an update.

# [Should Repositories return IQueryable?](http://softwareengineering.stackexchange.com/questions/192044/should-repositories-return-iqueryable)

|  |  |
| --- | --- |
|  | I have been seeing a lot of projects that have repositories that return instances of IQueryable. This allows additional filters and sorting can be performed on the IQueryable by other code, which translates to different SQL being generated. I am curious where this pattern came from and whether it is a good idea.  My biggest concern is that an IQueryable is a promise to hit the database some time later, when it is enumerated. This means that an error would be thrown outside of the repository. This could mean an Entity Framework exception is thrown in a different layer of the application.  I have also run into issues with Multiple Active Result Sets (MARS) in the past (especially when using transactions) and this approach sounds like it would lead to this happening more often.  I have always called AsEnumerable or ToArray at the end of each of my LINQ expressions to make sure the database is hit before leaving the repository code.  I am wondering if returning IQueryable could be useful as a building block for a data layer. I have seen some pretty extravagant code with one repository calling another repository to build an even bigger IQueryable. |

Returning IQueryable will definitely afford more flexibility to the consumers of the repository. It puts the responsibility of narrowing results off to the client, which naturally can both be a benefit and a crutch.

On the good side, you won't need to be creating tons of repository methods (at least on this layer) — GetAllActiveItems, GetAllNonActiveItems, etc — to get the data you want. Depending on your preference, again, this could be good or bad. You **will** (/should) need to define behavioral contracts which your implementations adhere to, but where that goes is up to you.

So you could put the gritty retrieval logic outside the repository and let it be used however the user wants. So exposing IQueryable gives the most flexibility and allows for efficient querying as opposed to in-memory filtering, etc, and could reduce the need for making a ton of specific data fetching methods.

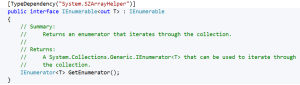
On the other hand, now you have given your users a shotgun. They can do things which you may not have intended (overusing .include(), doing heavy heavy queries and doing in-memory filtering in their respective implementations, etc), which would basically side-step the layering and behavioral controls because you have given full access.

So depending on the team, their experience, the size of the app, the overall layering and architecture … it depends :-\

A C-Sharp developer is always puzzled about usage of the following interfaces:

* IEnumerable
* IQueryable
* ICollection
* IList

I see developers are not that confident about usage of the mentioned interfaces. In this post, I am going to discuss basic differences among IEnumerable<T> and IQueryable<T> and in which scenario to use what.  
**IEnumerable<T>**  
The IEnumerable<T> is used to iterate a read only collection. It has only one method GetEnumeartor() which allows you to iterate the read only collection using a foreach loop. It only iterates in the forward direction.  IEnumerable <T> is defined in as follows:

[](https://i0.wp.com/blog.falafel.com/wp-content/uploads/2014/09/IEnumerable.png)  
As you see it does not contain any other functions like add, remove, count etc. So you cannot add, remove objects in IEnumerable collection. It is a read only collection and even to find the number of objects in collection, you need to iterate through entire collection.  Some of the important points about IEnumerable<T> is as follows:

* It is a read only collection.
* It iterates only in forward direction.
* It does not support adding, removing objects on collection.
* It provides enumerator to iterate collection in forward direction.

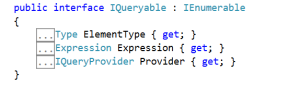
It is the base interface for any generic collection that can be enumerated using a foreach statement. To use iteration using a foreach the generic collection must implement the IEnumerable <T> and define GetEnumerator() method.  
GetEnumerator() method is not thread safe. Enumerator reads collection by positioning itself at the first position of the collection. You need to call MoveNext() method to read next object in collection.  
**When to use IEnumerable<T>**  
Try answering the following questions,

* Working with the read only collection
* Need to read the objects in forward direction only
* Not concerned about thread safety
* Want to iterate the collection’s objects using foreach

If answer to any of the above questions are **YES**, then consider using IEnumerable <T>

**IQueryable<T>**

As of the MSDN documentation, the IQueryable<T> allows you to execute a query against a specific data source wherein type of data is not specified.  
The IQueryable is defined as below:

[](https://i2.wp.com/blog.falafel.com/wp-content/uploads/2014/09/IQueryable.png)  
It extends the IEnumerable, hence it allows you to iterate over collection using the foreach statement. All the properties of IQueryable are read only.  
IQueryable<T> extends IQueryable and the IEnumerable. It is defined as follows:

[iq3](https://i2.wp.com/blog.falafel.com/wp-content/uploads/2014/09/iq3.png)  
If you are fetching records from remote databases IQueryable<T> should be used. It constructs the query using an Expression Tree. On the other hand in the IEnumerable <T> the query is constructed using delegates. Both IQueryable<T> and IEnumerable <T> support lazy loading of data from remote database servers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | DataClasses1DataContext context = new DataClasses1DataContext();              IEnumerable<Order> result = context.Orders;              var product = result.Where(x => x.OrderID == 10248);              context.Log = Console.Out;              foreach(var r in product)              {                    Console.WriteLine(r.ShipName);              } |

The generated SQL will be as follows:  
[IEnumerableSql](https://i0.wp.com/blog.falafel.com/wp-content/uploads/2014/09/IEnumerableSql.png)  
As you see even though we have applied filtering in query using where operator, IEnumerable<T> brings all data from database table and apply filter on returned data in memory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | IQueryable<Order> result1 = context.Orders;              var product1 = result1.Where(x => x.OrderID == 10248);              context.Log = Console.Out;              foreach (var r in product1)              {                  Console.WriteLine(r.ShipName);              }                Console.ReadKey(true); |

The generated SQL will be as follows:  
[IQueryableSQL](https://i2.wp.com/blog.falafel.com/wp-content/uploads/2014/09/IQueryableSQL.png)  
As you notice, the generated SQL has **where** clause and LINQ to SQL will (provider in this case) bring only filtered data from database. If you use IQueryable with foreach then query will be evaluated and executed at the same time.  
Some of the important points about IQueryable<T> are as follows:

* It implements IEnumerable <T> so the results can be iterated using foreach
* It is best suited to query against external data sources.
* It creates query using an Expression Tree
* It is used to query a queryable data source

When you use IQueryable<T>, the query gets created using an Expression Tree. Further the LINQ provider converts the Expression tree into the real SQL query. Always remember that IQueryable<T> does only construct query and it loads data in deferred way.  
**When to use IQueryable<T>**  
Try answering the following questions,

* Working with the queryable datasource
* Need to apply filter on data at the datasource
* Need to apply paging , composition
* Working with external data source
* Needs to load data in deferred way
* Need to use foreach to iterate collection

If answer to all of the above questions are **YES**, then consider using IQueryable<T>

Be careful that your use of IQueryable does not break the [encapsulation](http://deviq.com/encapsulation/)of your [repository](http://deviq.com/repository-pattern/). You do not want data-specific concerns to leak from your repository abstraction into the rest of your code. If what you really want is the ability to use a repository but with custom filters, consider adding additional query methods or passing a specification or predicate to one of the repository’s List methods (which would still return IEnumerable).

**Summary**  
The IEnumerable <T> works with collection in local memory whereas IQueryable<T> works with queryable data provider.  If you are working with LINQ to SQL then best is to work IEnumerable <T> since you don’t work with live data and to add or remove call functions from DataConext class.  IQueryable<T> does not return list of its own rather it constructs query. To return list from IQueryable<T> you need to use AsQueryable.

1. using System.Data.Entity.Infrastructure;
2. Change all of the catch blocks that catch DataException exceptions so that they catch RetryLimitExceededException exceptions instead. For example:
3. catch (RetryLimitExceededException /\* dex \*/)
4. {
5. //Log the error (uncomment dex variable name and add a line here to write a log.
6. ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists see your system administrator.");

}

We were using DataException to try to identify errors that might be transient in order to give a friendly "try again" message. But now that you've turned on a retry policy, the only errors likely to be transient will already have been tried and failed several times and the actual exception returned will be wrapped in the RetryLimitExceededException exception.

## Enable Command Interception

Now that you've turned on a retry policy, how do you test to verify that it is working as expected?  It's not so easy to force a transient error to happen, especially when you're running locally, and it would be especially difficult to integrate actual transient errors into an automated unit test. To test the connection resiliency feature, you need a way to intercept queries that Entity Framework sends to SQL Server and replace the SQL Server response with an exception type that is typically transient.

You can also use query interception in order to implement a best practice for cloud applications: [log the latency and success or failure of all calls to external services](https://www.asp.net/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/monitoring-and-telemetry#log) such as database services. EF6 provides a [dedicated logging API](http://msdn.microsoft.com/en-us/data/dn469464) that can make it easier to do logging, but in this section of the tutorial you'll learn how to use the Entity Framework's [interception feature](http://msdn.microsoft.com/en-us/data/dn469464) directly, both for logging and for simulating transient errors.

### Interceptor classes

Next you'll create the classes that the Entity Framework will call into every time it is going to send a query to the database, one to simulate transient errors and one to do logging. These interceptor classes must derive from the DbCommandInterceptor class. In them you write method overrides that are automatically called when query is about to be executed. In these methods you can examine or log the query that is being sent to the database, and you can change the query before it's sent to the database or return something to Entity Framework yourself without even passing the query to the database.

This code only overrides the ReaderExecuting method, which is called for queries that can return multiple rows of data. If you wanted to check connection resiliency for other types of queries, you could also override the NonQueryExecuting and ScalarExecuting methods, as the logging interceptor does.

When you run the Student page and enter "Throw" as the search string, this code creates a dummy SQL Database exception for error number 20, a type known to be typically transient. Other error numbers currently recognized as transient are 64, 233, 10053, 10054, 10060, 10928, 10929, 40197, 40501, abd 40613, but these are subject to change in new versions of SQL Database.

The code returns the exception to Entity Framework instead of running the query and passing back query results. The transient exception is returned four times, and then the code reverts to the normal procedure of passing the query to the database.

Because everything is logged, you'll be able to see that Entity Framework tries to execute the query four times before finally succeeding, and the only difference in the application is that it takes longer to render a page with query results.

The number of times the Entity Framework will retry is configurable; the code specifies four times because that's the default value for the SQL Database execution policy. If you change the execution policy, you'd also change the code here that specifies how many times transient errors are generated. You could also change the code to generate more exceptions so that Entity Framework will throw the RetryLimitExceededExceptionexception.

## Enable Code First Migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You have configured the Entity Framework to automatically drop and re-create the database each time you change the data model. When you add, remove, or change entity classes or change your DbContext class, the next time you run the application it automatically deletes your existing database, creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it is usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The [Code First Migrations](http://msdn.microsoft.com/en-us/data/jj591621) feature solves this problem by enabling Code First to update the database schema instead of dropping and re-creating the database. In this tutorial, you'll deploy the application, and to prepare for that you'll enable Migrations.

### Using code First Migrations to Deploy the Database

To deploy the database you'll use Code First Migrations.  When you create the publish profile that you use to configure settings for deploying from Visual Studio, you'll select a check box labeled **Execute Code First Migrations (runs on application start)**.  This setting causes the deployment process to automatically configure the application Web.config file on the destination server so that Code First uses the MigrateDatabaseToLatestVersion initializer class.

Visual Studio doesn't do anything with the database during the deployment process while it is copying your project to the destination server. When you run the deployed application and it accesses the database for the first time after deployment, Code First checks if the database matches the data model. If there's a mismatch, Code First automatically creates the database (if it doesn't exist yet) or updates the database schema to the latest version (if a database exists but doesn't match the model). If the application implements a Migrations Seed method, the method runs after the database is created or the schema is updated.

Your Migrations Seed method inserts test data. If you were deploying to a production environment, you would have to change the Seed method so that it only inserts data that you want to be inserted into your production database. For example, in your current data model you might want to have real courses but fictional students in the development database. You can write a Seed method to load both in development, and then comment out the fictional students before you deploy to production. Or you can write a Seed method to load only courses, and enter the fictional students in the test database manually by using the application's UI.

The Courses and OfficeAssignment properties are navigation properties. As was explained earlier, they are typically defined as

[virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4(v=vs.110).aspx) so that they can take advantage of an Entity Framework feature called

[lazy loading](http://msdn.microsoft.com/en-us/magazine/hh205756.aspx).

In addition, if a navigation property can hold multiple entities, its type must

implement the

[ICollection<T>](http://msdn.microsoft.com/en-us/library/92t2ye13.aspx) Interface. For example

[IList<T>](http://msdn.microsoft.com/en-us/library/5y536ey6.aspx)

qualifies but

not

[IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) because IEnumerable<T> doesn't implement

[Add](http://msdn.microsoft.com/en-us/library/63ywd54z.aspx).

If you define your navigation property virtual, Entity Framework will at runtime create a new class (dynamic proxy) derived from your class and uses it instead of your original class. This new dynamically created class contains logic to load the navigation property when accessed for the first time. This is referred to as "lazy loading". It enables Entity Framework to avoid loading an entire tree of dependent objects which are not needed from the database.

In some circumstances, it is best to use "Eager Loading" instead, especially if you know that you will be interacting with related objects at some point.

Julie Lerman really is the authority on all things Entity Framework, and she explains this process very well in her MSDN Article [Demystifying Entity Framework Strategies: Loading Related Data](http://msdn.microsoft.com/en-us/magazine/hh205756.aspx)

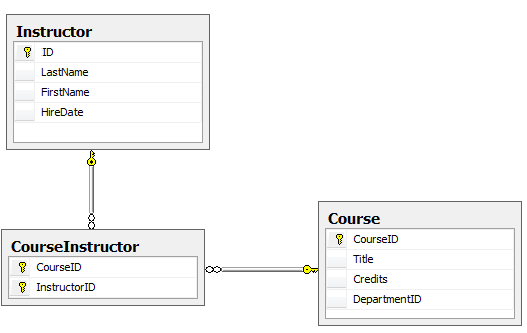
Eager loading with Include is useful for scenarios where you know in advance that you want the related data for all of the core data being queried. But remember the two potential downsides. If you have too many Includes or navigation paths, the Entity Framework may generate a poorly performing query. And you should be careful about returning more related data than necessary thanks to the ease of coding with Include.

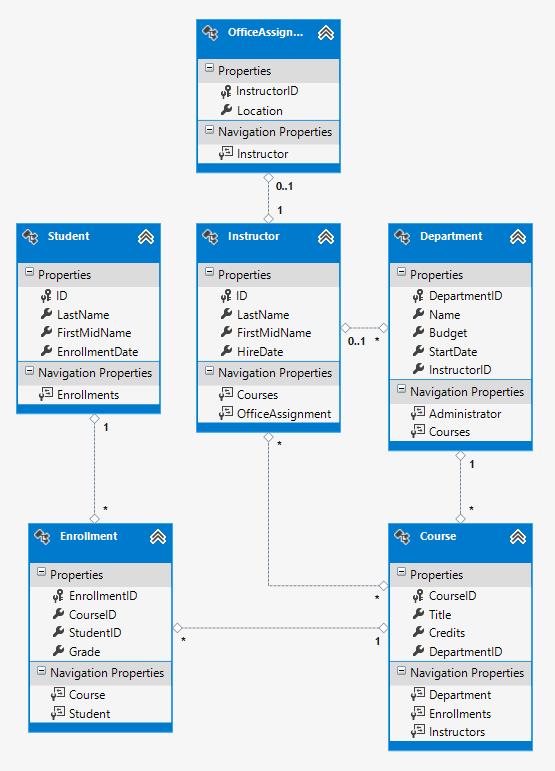
Lazy loading very conveniently retrieves related data behind the scenes for you in response to code that simply makes mention of that related data. It, too, makes coding simpler, but you should be conscientious about how much interaction it’s causing with the database. You may cause 40 trips to the database when only one or two were necessary.

If you are developing a Web Application where every communication with the server is a new context anyway, Lazy Loading will just create unnecessary overhead to maintain the dynamic class for related objects that will never be loaded. Many people will disable lazy loading in these scenarios. Ultimately, it's still best to evaluate your SQL queries which EF has built and determine which options will perform best for the scenario you are developing under.

Join table = Instructor has many courses and course has many instructors

The Entity Framework automatically creates the CourseInstructor table

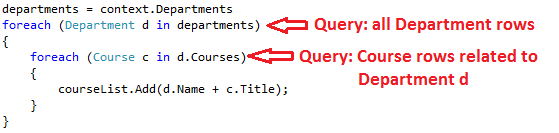




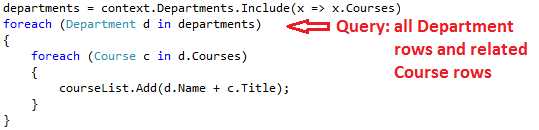
Lazy, Eager, and Explicit Loading of Related Data

There are several ways that the Entity Framework can load related data into the navigation properties of an entity:

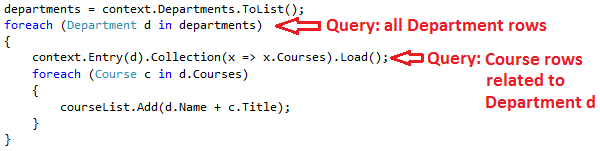
* Lazy loading. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. This results in multiple queries sent to the database — one for the entity itself and one each time that related data for the entity must be retrieved. The DbContext class enables lazy loading by default.



* Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading by using the Include method.



* Explicit loading. This is similar to lazy loading, except that you explicitly retrieve the related data in code; it doesn't happen automatically when you access a navigation property. You load related data manually by getting the object state manager entry for an entity and calling the [Collection.Load](http://msdn.microsoft.com/en-us/library/gg696220(v=vs.103).aspx) method for collections or the [Reference.Load](http://msdn.microsoft.com/en-us/library/gg679166(v=vs.103).aspx) method for properties that hold a single entity. (In the following example, if you wanted to load the Administrator navigation property, you'd replace Collection(x => x.Courses) with Reference(x => x.Administrator).)  Typically **you'd use explicit loading only when you've turned lazy loading off.**



**Because they don't immediately retrieve the property values, lazy loading and explicit loading are also both known as deferred loading.**

### Performance considerations

### The automatic scaffolding has specified eager loading for the Department navigation property by using the Include method.

If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, in the above examples, suppose that each department has ten related courses. The eager loading example would result in just a single (join) query and a single round trip to the database. The lazy loading and explicit loading examples would both result in eleven queries and eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios lazy loading is more efficient. Eager loading might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, lazy loading might perform better because eager loading would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Lazy loading can mask code that causes performance problems. For example, code that doesn't specify eager or explicit loading but processes a high volume of entities and uses several navigation properties in each iteration might be very inefficient (because of many round trips to the database). An application that performs well in development using an on premise SQL server might have performance problems when moved to Azure SQL Database due to the increased latency and lazy loading. Profiling the database queries with a realistic test load will help you determine if lazy loading is appropriate. For more information see [Demystifying Entity Framework Strategies: Loading Related Data](http://msdn.microsoft.com/en-us/magazine/hh205756.aspx) and [Using the Entity Framework to Reduce Network Latency to SQL Azure](http://msdn.microsoft.com/en-us/magazine/gg309181.aspx).

### Disable lazy loading before serialization

If you leave lazy loading enabled during serialization, you can end up querying significantly more data than you intended. Serialization generally works by accessing each property on an instance of a type. Property access triggers lazy loading, and those lazy loaded entities are serialized. The serialization process then accesses each property of the lazy-loaded entities, potentially causing even more lazy loading and serialization. To prevent this run-away chain reaction, turn lazy loading off before you serialize an entity.

Serialization can also be complicated by the proxy classes that the Entity Framework uses, as explained in the [Advanced Scenarios tutorial](https://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/advanced-entity-framework-scenarios-for-an-mvc-web-application#proxies).

**One way to avoid serialization problems is to serialize data transfer objects (DTOs) instead of entity objects, as shown in the** [**Using Web API with Entity Framework**](https://www.asp.net/web-api/overview/creating-web-apis/using-web-api-with-entity-framework/part-5) **tutorial.**

**If you don't use DTOs, you can disable lazy loading and avoid proxy issues by** [**disabling proxy creation**](http://msdn.microsoft.com/en-US/data/jj592886.aspx)**.**

Here are some other [ways to disable lazy loading](http://msdn.microsoft.com/en-US/data/jj574232):

* For specific navigation properties, omit the virtual keyword when you declare the property.
* For all navigation properties, set LazyLoadingEnabled to false, put the following code in the constructor of your context class:

this.Configuration.LazyLoadingEnabled = false;

Create a Courses Page That Displays Department Name

The Course entity includes a navigation property that contains the Department entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the Name property from the Department entity that is in the Course.Department navigation property.

**Lazy loading** is a **design pattern** that is commonly used to defer initialization of an object up until it is needed by the program. The gains of using the pattern include efficiency (if it’s used right) and sometime performance. **Eager loading** is the opposite **pattern** of **lazy loading**. In this pattern we initialize the object before hand and don’t wait to the second we really need it. In **Entity Framework** we can use the **Include** method in order to **eager load** an entity graph.

**Your service layer exposes DTO's. This means that in the service layer you define data contracts as you would like them to be exposed to the outside world. In most cases they are flattened entities that not necessarily have the same structure as your database entities.**

**It is the responsibility of your service layer to use business/data layer and construct the DTO's that you expose to the outside world.**

**What you use in your business and data layer depends on the architecture. You could have a domain model that is mapped with code first. In that case, the service layer maps domain entities to data contracts (DTO's). If you don't have a domain model (anemic model), then you could as well just map the database directly to your DTO's.**

**The ASP.NET MVC site consumes the service, and maps the DTO's it receives to dedicates view models that are then passed to the specific view.**

**In addition, you may decide to also split queries from commands. This is a good approach because the DTO's that you get back as the reqult of a query are totally different than commands that you send to the service. A command only contains what's needed to execute the command and contain the business intend what you want to achieve, while a query returns a flattened model of what you need in the UI.**

**Other remarks:**

* **Don't expose your database entities.**

---Don't convert in business layer, as it's not business logic.

Here are some other [ways to disable lazy loading](http://msdn.microsoft.com/en-US/data/jj574232):

* For specific navigation properties, omit the virtual keyword when you declare the property.
* For all navigation properties, set LazyLoadingEnabled to false, put the following code in the constructor of your context class:

this.Configuration.LazyLoadingEnabled = false;

**Why to use the Repository Pattern?**

**There are a lot of reasons for using the Repository Pattern.**

**For example:**

**Testability. Using the pattern we can create stubs that**

**can replace the real data access objects. This can**

**help us to test our business logic without concerning what the**

**data access is doing.**

**Abstraction. Using the pattern we create an abstraction above**

**our data access functionality. This abstraction can help us when we**

**want to change the implementation of the data access without**

**affecting our business logic code. For example, I had to change**

**implementation of data access with a call to a web service. Using**

**the pattern I only needed to change the object that I used and that is**

**it.**

**Dependency Injection. Using the pattern we can use DI containers to**

**inject the relevant object that we want to use in the code.**