Virtual Navigation Properties on Entities

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.

Demystifying Entity Framework Strategies: Loading Related Data

By [Julie Lerman](https://msdn.microsoft.com/en-us/magazine/mt149362?author=julie+lerman) | June 2011

In [last month’s Data Points column](https://msdn.microsoft.com/en-us/magazine/hh148150.aspx), I provided some high-level guidance for choosing a modeling workflow strategy from the Database First, Model First and Code First options. This month, I’ll cover another important choice you’ll need to make: how to retrieve related data from your database. You can use eager loading, explicit loading, lazy loading or even query projections.

This won’t be a one-time decision, however, because different scenarios in your application may require different data-loading strategies. Therefore, it’s good to be aware of each strategy so that you can choose the right one for the job.

As an example, let’s say you have an application that keeps track of family pets and your model has a Family class and a Pet class with a one-to-many relationship between Family and Pet. Say you want to retrieve information about a family and their pets.

In the next column, I’ll continue this series by addressing the various choices you have for querying the Entity Framework using LINQ to Entities, Entity SQL and variations on each of those options. But in this column, I’ll use only LINQ to Entities for each of the examples.

**Eager Loading in a Single Database Trip**

Eager loading lets you bring all of the data back from the database in one trip. The Entity Framework provides the Include method to enable this. Include takes a string representing a navigation path to related data. Here’s an example of the Include method that will return graphs, each containing a Family and a collection of their Pets:

from f in context.Families.Include("Pets") select f

If your model has another entity called VetVisit and that has a one-to-many relationship with Pet, you can bring families, their pets and their pet’s vet visits back all at once:

from f in context.Families.Include("Pets.VetVisits") select f

Results of eager loading are returned as object graphs, as shown in **Figure 1**.

Figure 1 **Object Graph Returned by Eager Loading Query**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Id | Name | Pets |  |  |  |  |  |
|  |  | Id | Name | Type | VetVisits |  |  |
| 2 | LermanJ | 2 | Sampson | Dog | 1 | 2/1/2011 | Excellent |
| 5 | 4/1/2011 | Nail Clipping |
| 4 | Sissy | Cat | 1 | 3/2/2011 | Excellent |
| 3 | GeigerA | 3 | Pokey | Turtle | 3 | 2/5/2011 | Excellent |
| 4 | Riki | Cat | 6 | 4/8/2011 | Excellent |

Include is pretty flexible. You can use multiple navigation paths at once and you can also navigate to parent entities or through many-to-many relationships.

Eager loading with Include is very convenient, indeed, but if overused—if you put many Includes in a single query or many navigation paths in a single Include—it can detract from query performance pretty rapidly. The native query that the Entity Framework builds will have many joins, and to accommodate returning your requested graphs, the shape of the database results might be much more complex than necessary or may return more results than necessary. This doesn’t mean you should avoid Include, but that you should profile your Entity Framework queries to be sure that you aren’t generating poorly performing queries. In cases where the native queries are particularly gruesome, you should rethink your query strategy for the area of the application that’s generating that query.

**Lazy Loading in Additional Database Trips**

Often when retrieving data, you don’t want or need the related data right away, or you may not want related data for all of the results. For example, you might need to pull all of the families into your app, but then only retrieve pets for some of those people. Does it make sense in this case to eager load all of the pets for every person entity with Include? Probably not.

The Entity Framework offers two ways to load related data after the fact. The first is called lazy loading and, with the appropriate settings, it happens automatically.

With lazy loading, you simply need to make some reference to the related data and the Entity Framework will check to see whether it’s been loaded into memory. If not, the Entity Framework will create and execute a query behind the scenes, populating the related data.

For example, if you execute a query to get some Family objects, then trigger the Entity Framework to get the Pets for one of those Families simply by making mention, so to speak, of the Pets property, the Entity Framework will retrieve the Pets for you:

var theFamilies= context.Families.ToList();

var petsForOneFamily = theFamilies[0].Pets;

In the Entity Framework, lazy loading is enabled or disabled using the ObjectContext ContextOptions.LazyLoadingEnabled property. By default, Visual Studio will define newly created models to set LazyLoadingEnabled to true, the result being that it’s enabled by default with new models.

Having lazy loading enabled by default when you instantiate a context may be a great choice for your application, but can also be a problem for developers who aren’t aware of this behavior. You may trigger extra trips to the database without realizing you’re doing so. It’s up to you to be aware of whether you’re using lazy loading, and you can explicitly choose to use it or not—enabling or disabling it in your code by setting LazyLoadingEnabled to true or false—as needed.

Lazy loading is driven by the EntityCollection and EntityReference classes and therefore won’t be available by default when you’re using Plain Old CLR Object (POCO) classes—even if LazyLoadingEnabled is true. However, the Entity Framework dynamic proxy behavior, triggered by marking navigation properties as virtual or Overridable, will create a runtime proxy that will allow your POCOs to lazy load as well.

Lazy loading is a great feature to have available in the Entity Framework, but only if you’re aware of when it’s active and consider making choices of when it’s appropriate and when it’s not. For example, the *MSDN Magazine* article, “Using the Entity Framework to Reduce Network Latency to SQL Azure” ([msdn.microsoft.com/magazine/gg309181](https://msdn.microsoft.com/magazine/gg309181)), highlights performance implications of using lazy loading against a cloud database from an on-premises server. Profiling the database queries executed by the Entity Framework is an important part of your strategy to help you choose between loading strategies.

It’s also important to be aware of when lazy loading is disabled. If LazyLoadingEnabled is set to false, the reference to theFamilies[0].Pets in the previous example would not trigger a database query and would report that the family has no pets, even though there may be some in the database. So if you’re relying on lazy loading, be sure that you’ve got it enabled.

**Explicitly Loading in Additional Database Trips**

You may want to leave lazy loading disabled and have more explicit control over when related data is loaded. In addition to explicitly loading with Include, the Entity Framework allows you to selectively and explicitly retrieve related data using one of its Load methods.

If you generate entity classes using the default code-generation template, the classes will inherit from EntityObject, with related data exposed in an EntityCollection or an EntityReference. Both of these types have a Load method that you can call to force the Entity Framework to retrieve the related data. Here’s an example of loading an EntityCollection of Pets objects. Notice that Load doesn’t have a return value:

var theFamilies = context.Families.ToList();

theFamilies[0].Pets.Load();

var petsForOneFamily = theFamilies[0].Pets;

The Entity Framework will create and execute a query that populates the related property—the Pets collection for the Family—and then you can work with the Pets.

The second way to explicitly Load is from the ObjectContext, rather than from EntityCollection or EntityReference. If you’re relying on POCO support in the Entity Framework, your navigation properties won’t be EntityCollections or EntityReferences, and therefore won’t have the Load method. Instead, you can use the ObjectContext.LoadProperty method. LoadProperty uses generics to identify the type that you’re loading from and then a lambda expression to specify which navigation property to load. Here’s an example of using LoadProperty to retrieve the Pets for a particular person instance:

context.LoadProperty<Family>(familyInstance, f => f.Pets)

**Query Projections as an Alternative to Loading**

Don’t forget that you also have the option to use projections in your queries. For example, you can write a query to retrieve entities, but filter which related data is retrieved:

var famsandpets = from f in context.Families

let returnAllPets = f.Pets.Any(p => p.Type == "Reptile")

select new { Family = f, Pets = f.Pets .Where(p => returnAllPets ? true : false) };

This will return all of the families and the pets for any of those families that have any reptiles—all in a single trip to the database. But rather than a graph of family with their pets, the famsAndPets query will return a set of anonymous types with one property for Family and another for Pets (see **Figure 2**).

Figure 2 **Projected Anonymous Types with Family and Pets Properties**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Family | Pets |  |  |  |
| Id | Name | Id | Name | Type |
| 2 | LermanJ |  |  |  |
| 3 | GeigerA | 3 | Pokey | Turtle |
| 4 | Riki | Cat |

**Evaluate the Pros and Cons**

You now have four strategies available for retrieving related data. They need not be mutually exclusive in your application. You may very well find cause for each of these different features in various scenarios throughout your applications. The pros and cons of each strategy should be considered before choosing the right one for each case.

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.

Explicit loading gives you more control over when related data is retrieved (and which data), but if you don’t know about this option, your application could be misinformed about the presence of related data in the database. Many developers find it cumbersome to explicitly load, while others are happy to have the granular control it provides.

Using projection in your queries potentially gives you the best of both worlds, selectively retrieving related data in a single query. However, if you’re returning anonymous types from these projections, you may find those more cumbersome to work with, as the objects aren’t tracked by the Entity Framework state manager, so they aren’t updateable.

**Figure 3** shows a decision-making flowchart that you can use for your first pass at choosing a strategy. But you should also be considerate of performance. Take advantage of query profiling and performance-testing tools to ensure that you’ve made the right data-loading choices.

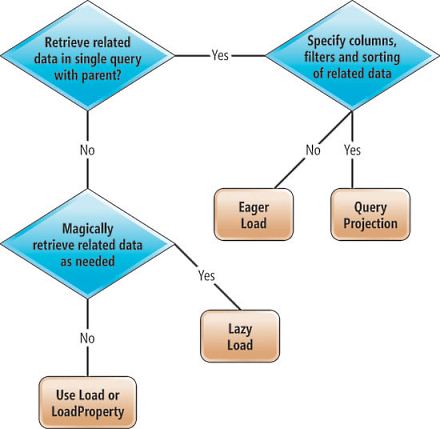


Figure 3 **Your First Pass at Loading Strategy Decisions**