WEB524

WEB PROGRAMMING ON WINDOWS

WEEK 4 – LECTURE 1

ASSOCIATED DATA

Topics

- Previously, we worked with single standalone entities.

- Today, we learn the design and coding patterns for associated entities.

Associated Data Introduction

- We are working with in-memory data (objects and collections of objects).
- We do not care about foreign keys, joins, and so on.
- Real-life and software objects can have associations (or relationships) between them. There are several kinds of associations that we will work with:
  - One-to-many
  - Many-to-many
  - One-to-one *(covered in future class)*
  - To-one, self-associated *(covered in future class)*
  - To-many, self-associated *(covered in future class)*

One-to-Many

- You will often model real-life objects that have associations with other objects, in a one-to-many relationship.
- For example:
  - “Program” has a collection of “Course” objects,
  - “Supplier” has a collection of “Product” objects, and
  - “Manufacturer” has a collection of “Vehicle” objects.
- Looking at the other side of the association:
  - “Course” object belongs to a “Program”, and similarly,
  - “Product” is made by a “Supplier”, and
  - “Vehicle” is made by a “Manufacturer”.

Many-to-Many

- Consider your role as a student here at Seneca:
  - A “Course” (like WEB524) has a collection of “Student” objects, and
  - A “Student” has a collection of “Course” objects.
- Now, think about the faculty employees here in the School of SDDS:
  - An “Employee” (e.g. Peter McIntyre, or Ian Tipson) can have many *(a collection of)* “Job Duty” objects, and
  - A “Job Duty” (e.g. teacher, coordinator, or researcher) can be done by many *(a collection of)* “Employee” objects.

Are associations *always* required?

- When you are writing entity classes, associations are not always required between all classes.
- We use associations for data that is used for lookup or validation purposes.

Declaring a Navigation Property

- When writing design model classes (entities) with associations, the associations are coded as properties.
- We call the associations “navigation properties“.

*A **navigation property** has a data type of another class in the model.*

- ALWAYS add a navigation property to both associated classes.

Declaring a “to-one” Navigation Property

- Declare a “to-one” navigation property with the data type of the associated class.
- For example, if a Product is sold by a single Supplier:

```
[Required]
public Supplier Supplier { get; set; }
```

Setting the value of a “to-one” nav property

- Before setting the value of a navigation property, you must have a reference to an object (of that data type).
- Even though you can create a new object in the statement, you will often have an existing variable for the referenced object.

- Assume that “walmart” is a Supplier object, and “shirt” is a Product object.
- Set a “to-one” property as follows.

```
// shirt is a Product object
// walmart is a Supplier object
shirt.Supplier = walmart;
```

Declaring a “to-many” Navigation Property

- Declare a “to-many” navigation property with a collection type of the associated class.
- For example, a Supplier sells a number of Product items:

```
// In a design model class...
public ICollection<Product> Products { get; set; }

// In a view model class...
public IEnumerable<Product> Products { get; set; }
```

- Remember: whenever you declare a collection property, you MUST initialize it in a default constructor, most often as a List<TEntity> or HashSet<TEntity>.

Setting the value of a “to-many” nav property

- Setting a “to-many” property can happen in a few different ways.
  - You can add a new object to the collection.
  - You can replace the existing collection with a new collection

- Here are some examples, using the same “walmart” and “shirt” objects:

```
// add a new object to the collection
walmart.Products.Add(shirt);

// replace the existing collection with a new collection
walmart.Products = shirts;
```

Question …

- When you write code to set one end of the association, do you need to write another statement to set the other end?

- For example, if you set the supplier of the “shirt” product to “walmart“, must you also add the “shirt” to the collection of “products” for the object “walmart”?

Answer

- When using a persistent store such as SQL Server (via the Entity Framework)… then you don’t have to.
- In this scenario, when you set one end of the association, the persistence framework sets both ends of the association upon saving changes.

Reading the value of a navigation property

- Getting the value of a navigation property is simple but pay attention to the data type of the property. It may be an object or a collection of objects.

- When you have a “to-one” association, it’s easy to “walk” the object graph to get access to properties in the associated object. For example, using the “shirt” object, you could get the “Name” or “Address” of the supplier:

```
string supplierName = shirt.Supplier.Name;
```

Question

- At this point, you have learned how classes can be associated with one another by using navigation properties.

- You can use navigation properties in any class that describes a real-world object – design model class or view model class.

- Do we add navigation properties to ALL view model classes?

Including an associated object or collection in View Models

- No, add them only when it makes sense.
- Often, the “Base” view model class is intended to hold all, or almost all, of the properties defined in the design model class. You may be tempted to add a navigation property to the “Base” class but don’t!
- Instead, create another view model class with the desired navigation property. It’s okay to use inheritance to simplify coding.

- Let’s look at three scenarios from the code example.
  - The problem domain is a one-to-many association of recent National Football League playoff *teams* and *players*.
  - A **Team** has a collection of **Players**.
  - A **Player** belongs to a **Team**.

TeamBaseViewModel

```
public class TeamBaseViewModel
{
  public int Id { get; set; }
  public string CodeName { get; set; }
  public string Name { get; set; }
  public string City { get; set; }
  public int YearFounded { get; set; }
  public string Conference { get; set; }
  public string Division { get; set; }
  public string Stadium { get; set; }
}
```

PlayerBaseViewModel

```
public class PlayerBase
{
  public PlayerBase() {
    BirthDate = DateTime.Now.AddYears(-25);
  }

  public int Id { get; set; }
  public int UniformNumber { get; set; }
  public string PlayerName { get; set; }
  public string Position { get; set; }
  public string Height { get; set; }
  public int Weight { get; set; }
  public DateTime BirthDate { get; set; }
  public int YearsExperience { get; set; }
  public string College { get; set; }
}
```

Add the Navigation Property for the Player Collection

- Notice:
  - The name of the view model class. It inherits from the “Base” view model.
  - The collection type, in a view model class, is IEnumerable<T>.
  - The data type (of the navigation property) is a collection of a view model class type.
  - The name (of the navigation property) exactly matches the name in the design model class.

```
public class TeamWithPlayersViewModel : TeamBaseViewModel
{
  public TeamWithPlayersViewModel()
  {
    Players = new List<PlayerBaseViewModel>();
  }

  public IEnumerable<PlayerBaseViewModel> Players { get; set; }
}
```

Add the Navigation property for the Team Object

- Notice:
  - The name of the view model class. It inherits from the “Base” view model.
  - The data type (of the navigation property) is a view model class type.
  - The name (of the navigation property) exactly matches the name in the design model class.
  - The to-one navigation property almost always needs a “Required” data annotation.

```
public class PlayerWithTeamInfoViewModel : PlayerBaseViewModel
{
  [Required]
  public TeamBaseViewModel Team { get; set; }
}
```

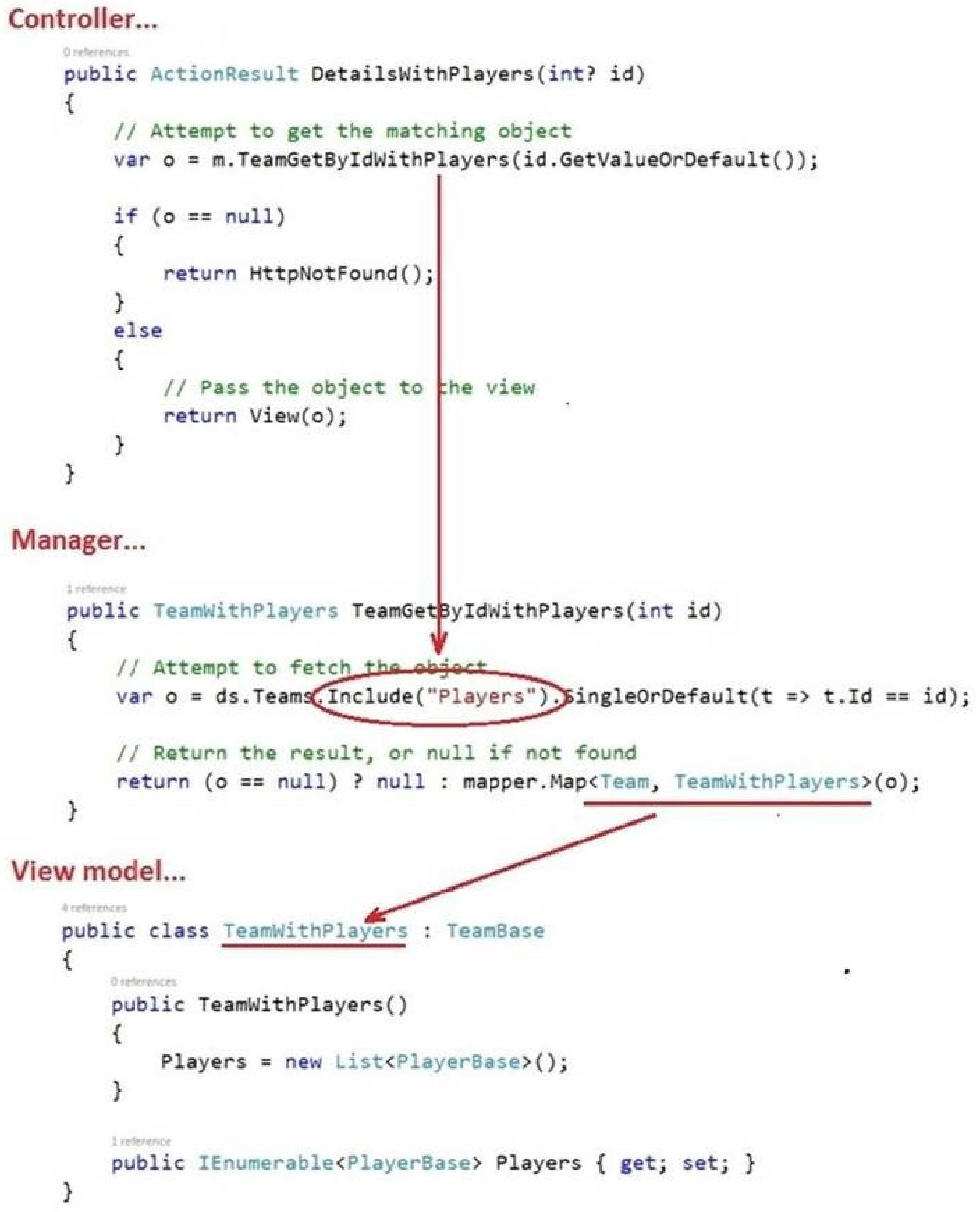
AutoMapper and Associated Data

- If you define maps from design model classes to view model classes that have navigation properties, AutoMapper will work the same way that it does for other data types.
- In other words, AutoMapper works nicely with associated entities in all of these scenarios:
    - For an object, mapping an associated <u>collection</u>
    - For an object, mapping an associated <u>object</u>
    - Mapping the individual properties of a related object (in a "to-one" association)
- Let's look at each one more closely.

Scenario 1: Including an associated **collection**

- If you have an entity that has an associated collection, you may want to return an object with its associated collection already populated.
- Some examples: a Program has a collection of Subjects; a Supplier has a collection of Products; or a Team has a collection of Players.

- To do this, you need to perform four tasks:
  1. In a new view model class, add a navigation property for the collection. The type is a view model class type and the property name must match the property name in the design model class.
  2. Define a map from the design model class to the new view model class that you created in step 1 above.
  3. In the manager, when the code fetches the object, add the **Include()** extension method to fetch its associated collection.
  4. In the controller, add a method that will call the manager method.



Controller...

```
0 references
public ActionResult DetailsWithPlayers(int? id)
{
    // Attempt to get the matching object
    var o = m.TeamGetByIdWithPlayers(id.GetValueOrDefault());

    if (o == null)
    {
        return HttpNotFound();
    }
    else
    {
        // Pass the object to the view
        return View(o);
    }
}
```

Manager...

```
1 reference
public TeamWithPlayers TeamGetByIdWithPlayers(int id)
{
    // Attempt to fetch the object
    var o = ds.Teams.Include("Players").SingleOrDefault(t => t.Id == id);

    // Return the result, or null if not found
    return (o == null) ? null : mapper.Map<Team, TeamWithPlayers>(o);
}
```

View model...

```
4 references
public class TeamWithPlayers : TeamBase
{
    0 references
    public TeamWithPlayers()
    {
        Players = new List<PlayerBase>();
    }

    1 reference
    public IEnumerable<PlayerBase> Players { get; set; }
}
```

- Note: View model should have "ViewModel" appended to the name.

Scenario 2: Mapping an associated **object**

- If you have an entity that has an associated entity you may want to return an object with its associated object already populated.
- For example, a Subject is associated with a single Program.

- To do this, you need to perform four tasks:
    1. In a new view model class, add a navigation property for the object. The type is a view model class type and the property name must match the property name in the design model class.
    2. Define a map from the design model class to the new view model class that you created in step 1 above.
    3. In the manager, when the code fetches the object, add the **Include()** extension method to fetch its associated object.
    4. In the controller, add a method that will call the manager method.

Mapping an associated **object**

```
// View Model
public class PlayerWithTeamInfoViewModel : PlayerBaseViewModel
  {
      public TeamBaseViewModel Team { get; set; }
  }

// Manager
public IEnumerable<PlayerWithTeamInfoViewModel> PlayerGetAllWithTeamInfo()
      {
          var c = ds.Players
              .Include("Team")
              .OrderBy(p => p.PlayerName);

          return mapper.Map<IEnumerable<Player>, IEnumerable<PlayerWithTeamInfoViewModel>>(c);
      }
```

Scenario 3: Mapping some individual properties of a related object

- If you have an entity that has an associated object you may want to return individual properties from its associated object.
- An example: Product belongs to a Supplier, we may want the Supplier Name.

- To do this, you need to do four tasks:
    1. In a new view model class, add a property with a type that matches the property in the associated type. The property name must be a composite of the design model class name and the property name.
    2. Define a map, from the design model class to the new view model class that you created in step 1 above.
    3. In the manager, when the code fetches the object, add the **Include()** extension method to fetch its associated object.
    4. In the controller, add a method that will call the manager method.

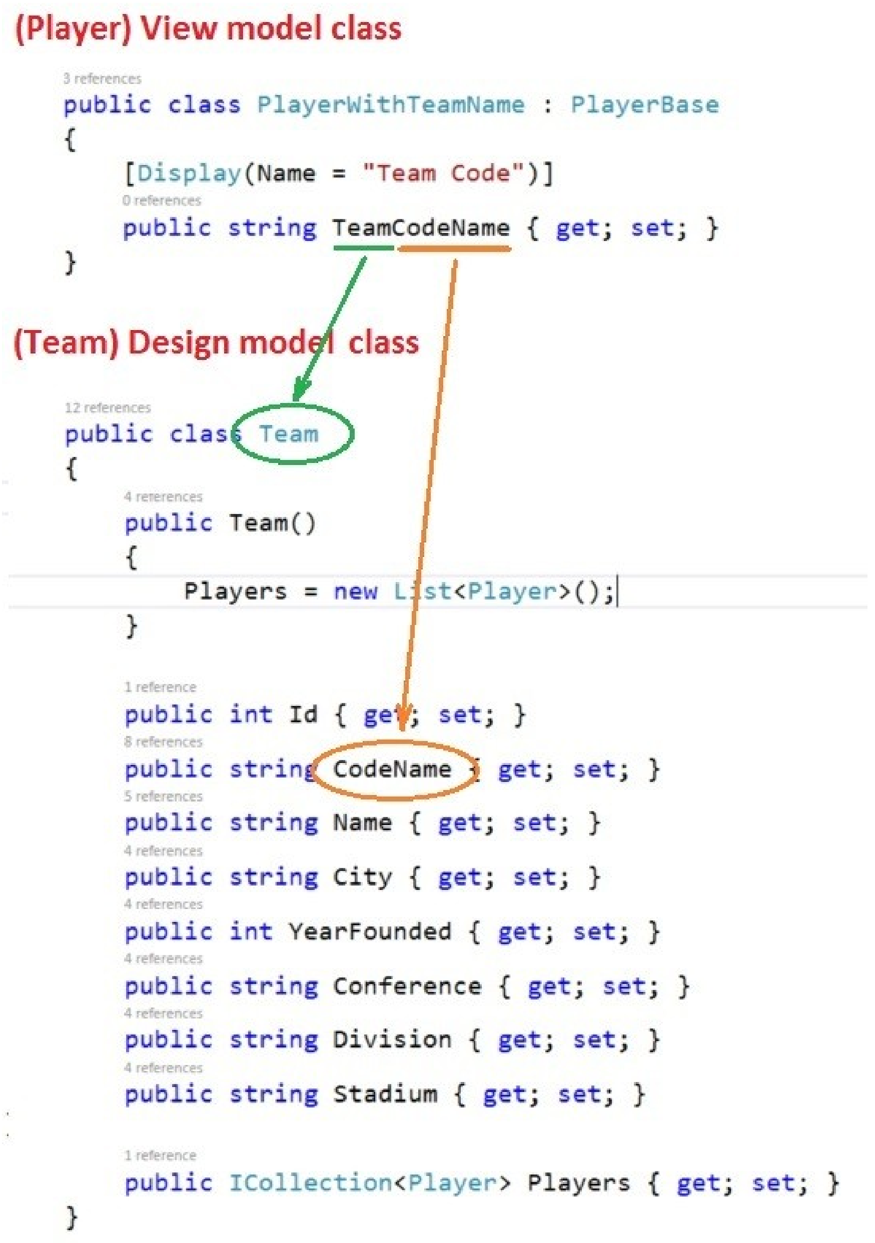
AutoMapper “Flattening” Feature

- It is a common scenario to add a navigation property for a single object, for example, you may add the **Team** object to the **PlayerWithTeamInfo** view model class.
- Sometimes you do not want, or need, all the properties from the associated object. Instead, you may only need one or two properties from the associated object.
- AutoMapper has a feature called “flattening” which helps with this scenario.

AutoMapper “Flattening” Feature Example

- Assume that we only want the Team’s “Name” and “CodeName” properties.
- Create new properties, with a composite name, Class + Property Name.
- The data type must match the data type of the property in the design model class.
- Here’s what it looks like:

```
public class PlayerWithTeamNamesViewModel : PlayerBaseViewModel
{
  // Design model class name is "Team"
  // Property names are "CodeName" and "Name"
  public string TeamCodeName { get; set; }
  public string TeamName { get; set; }
}
```



```
3 references
public class PlayerWithTeamName : PlayerBase
{
    [Display(Name = "Team Code")]
    0 references
    public string TeamCodeName { get; set; }
}
```

```
12 references
public class Team
{
    4 references
    public Team()
    {
        Players = new List<Player>();
    }

    1 reference
    public int Id { get; set; }
    8 references
    public string CodeName { get; set; }
    5 references
    public string Name { get; set; }
    4 references
    public string City { get; set; }
    4 references
    public int YearFounded { get; set; }
    4 references
    public string Conference { get; set; }
    4 references
    public string Division { get; set; }
    4 references
    public string Stadium { get; set; }

    1 reference
    public ICollection<Player> Players { get; set; }
}
```

- Again, view model name should include “ViewModel”

Displaying associated data in a view

- Please review the code example posted on Blackboard: **AssocOneToMany**

Displaying the objects in a collection property

- In a controller method (for example: get all, or get one), the code will call a manager method that will return a package of data that's described by a view model.
- For example, the "get one" action will return a Team object that includes a (nested) collection of Player objects.
- Review the **DetailsWithPlayers()** method.
  - It passes the fetched object to the view.
  - The view was scaffolded by using the standard Details template.
  - The view was then edited.
- The Visual Studio scaffolder will handle simple properties correctly.
- The scaffolder will not render HTML for other data types. We must usually add code to the views to render the other data types.

DetailsWithPlayers view

- Review the DetailsWithPlayers view. You will notice additional code added to the bottom. *For each object in the Players collection render some HTML.*

```
<dt>
  @Html.DisplayNameFor(model => model.Players)
</dt>
 <dd>
  @foreach (var p in Model.Players)
  {
    var player = string.Format("{0} ({1}), # {2}", p.PlayerName, p.Position, p.UniformNumber);
    <span>@player</span><br />
  }
</dd>
```

Displaying an associated individual/single object

- Consider the other end of the “to-one” association.
- A Player object has a “to-one” association with a Team object.
- In other words, a player is on only one team at a time, and we want to display the team name.
- In the players controller, there are two “get all” players methods that fetch team data.
  - One fetches the Team object itself (…WithTeamInfo).
  - The other fetches some data, using AutoMapper flattening (…WithTeamName).
  - Each method has its own view.

WithTeamInfo View

- Review the WithTeamInfo View.
- There’s an HTML table header element, but look in the “foreach” code block. There’s a new element, which dereferences the player’s Team object, to get its CodeName (i.e. the abbreviated team name/characters):

```
<td>
  @Html.DisplayFor(m => m.Team.CodeName)
</td>
```

WithTeamName View

- Review the WithTeamName view. The view model class is different from the previous example. It has a flattened string property for the team name instead of the object itself.

```
<td>
  @Html.DisplayFor(m => m.TeamCodeName)
</td>
```

- This HTML code is rendered by the Visual Studio scaffolder but the code in the WithTeamInfo view is not supported by the scaffolder.

Query techniques

- This section will introduce query techniques when working with associated data.
- For now, let’s assume that we are working with a one-to-many association.
- If you need an object and its associated object or collection, this work must be done in the Manager class.
- Use the **Include()** method on the DbSet<TEntity> collection.
- As you will read in the MSDN Reference:
  - The Include() method accepts a string which is the name of the navigation property for the associated object/collection.
  - Include() is defined in the DbQuery<TEntity> class.
  - DbSet<TEntity> inherits from DbQuery<TEntity>.

Query Example

```
// The following code is in a method in the Manager class
// Get all, each with its associated object
var c = ds.Products.Include("Supplier").OrderBy(p =>p.ProductCode);

// Get one, with its associated collection
var o = ds.Suppliers.Include("Products").SingleOrDefault(s => s.Id ==
id);
```

Include Multiple Associations (Branch Pattern)

- If you need to include data from more than one associated entity, simply add another **Include()** method to the query expression.
- This is sometimes referred to as “branching out” to include the associated entities.

```
var c = ds.Products.Include("Supplier").Include("Category");
```

Include Multiple Associations (Chain Pattern)

- Assume that you want to include two (or more) associated entities.
- One is a navigation property (e.g. Foo) but the other (e.g. Bar) is not directly associated with the entity class you're starting with.
- In other words, you're following a linked path (or a chain) of associated entities.
- In the above situation, in the Include() method parameter, specify a dot-separated path to the final destination using the design model class property names.

```
var c = ds.Tracks.Include("Album.Artist");
```

Filtering and Associated Objects

- Let’s assume that we wish to select only those **Program** objects that include a **Subject** object with a specific identifier. We can build the statement in two steps.
- By default, when we fetch an object that is related/associated with another object and/or collection, the object that is fetched does not include the related/associated objects. This is the default behaviour of the way we have configured the **DbContext** object.
- The program must explicitly “include” the associated data, this is referred to as “eager loading”.

Filtering and Associated Objects Example

- To build the statement, we first must use the **Include()** method. The return type is **DbQuery<TEntity>** (it will be a collection).

```
var progs = ds.Programs.Include("Subjects");
```

- The second task is to add filtering to the previous statement:

```
var progs = ds.Programs.Include("Subjects").Where(p => p.Credential ==
"Degree");
```

- The range variable *p* represents an object in the **Programs** collection. Its type is **Program** (not **Subject**).
- The return type of the entire statement is IQueryable<TEntity> (remember "Deferred Execution"?).

Filtering and sorting, with associated objects

- You can combine tasks by chaining the filtering and sorting tasks, from the previous slide, using the “fluent” syntax.

```
var progs = ds.Programs
    .Include("Subjects")
    .Where(p => p.Credential == "Degree")
    .OrderBy(p => p.Code);
```

- Here’s how to read, in English, the above statement:
  - In the Program collection…
  - for each Program object…
  - include the related Subject object(s)…
  - but only where a Program object’s Credential property value matches the string “Degree” …
  - then sort the result by the Program object’s Code property value.

Suggested query strategies for web apps

- *The following are suggested query strategies for web apps. These are not rules, they're just suggestions. Although these are not "best practices" you must balance the needs of the use case with the cost of fulfilling the query.*
- Get all
  - We do not typically use the Include() method in a "get all" method.
  - The main reason is data volume – it's possible that the included data will be sizeable in relation to the size of the fetched collection.
  - If you really need some or all of an included object, for example, then that may be acceptable.
  - You can limit the size of the fetched collection by using the Take() method.
  - There are advanced LINQ query expression components that will result in the generation of an efficient query at the database.
- Get one
  - We often use the Include() method in a "get one" method.
  - The included object/collection typically adds good value to the fetched object.
  - Richer strings can be composed and collection items can be rendered.

Lazy Loading vs Eager Loading

- When working with associated data in a persistent store (like a database or web service), it is possible to configure the querying settings to use lazy or eager loading.
- Lazy Loading:
  - A query that includes associated data will fetch the desired object or collection.
  - The syntax does NOT require you to use the Include() method in the query expression.
  - In the fetch action, the query will not immediately fetch the associated data. Instead, it will wait for an attempt to access (or dereference) the associated object/collection. When the programmer dereferences a property, a second fetch will execute and return the associated data automatically.
- Eager Loading:
  - A query that includes associated data will fetch the desired object or collection and the associated data.
  - The syntax of the query expression MUST use the Include() method.

Lazy Loading

- Sounds like lazy loading offers a pretty good deal, right? Simpler syntax etc.
- Nope – at least not for web apps.
- Do NOT be tempted to use it. In fact, when I create a new project one of the first things I do is ensure Lazy Loading has been turned off.
- Why? Interactions with web apps (from a client) follow the HTTP protocol. A request-response cycle is an atomic task, which completes when the response is delivered. The server does not persist request state, in memory, or anywhere else. Therefore, there's no opportunity for lazy loading to work.
- Unfortunately, many books and web documents that include coverage of Entity Framework tend to use lazy loading code. For example, the design model classes use the virtual keyword on navigation properties, which is supposed to trigger the lazy loading functionality. However, it's ineffective and useless for web apps.