**EF Core Relationships in ASP .NET Core**

By Shahed C on February 4, 2019

[1 Reply](https://wakeupandcode.com/ef-core-relationships-in-asp-net-core/#comments)

This is the **fifth**of a new [series of posts](https://wakeupandcode.com/aspnetcore/#aspnetcore2019) on ASP .NET Core for 2019. In this series, we’ll cover 26 topics over a span of 26 weeks from January through June 2019, titled **A-Z of ASP .NET Core!**

** A – Z of ASP .NET Core!**

**In this Article:**

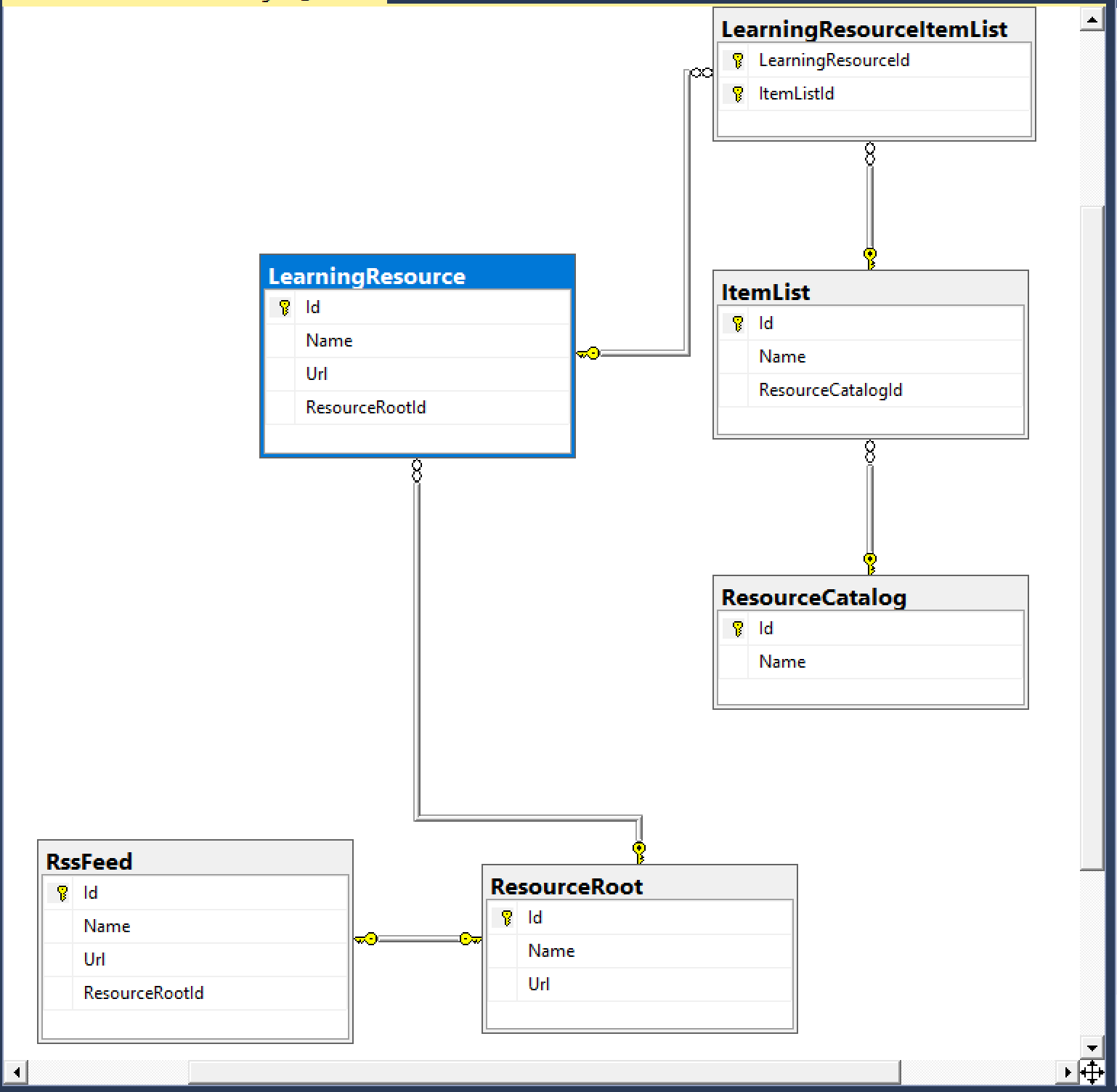
* E is for EF Core Relationships
* Classes and Relationships
* One to One
* One to Many (Example 1)
* One to Many (Example 2)
* Many to Many
* References

**E is for EF Core Relationships**

In my [2018 series](https://wakeupandcode.com/aspnetcore/#aspnetcore2018), we covered [EF Core Migrations](https://wakeupandcode.com/ef-core-migrations-in-asp-net-core/) to explain how to add, remove and apply Entity Framework Core Migrations in an ASP .NET Core web application project. In this article, we’ll continue to look at the [NetLearner](https://wakeupandcode.com/netlearner-asp-net-core-internet-learning-helper/) project, to identify entities represented by C# model classes and the relationships between them.

* NetLearner on GitHub: <https://github.com/shahedc/NetLearner>

**NOTE:** Please note that NetLearner is a work in progress as of this writing, so its code is subject to change. The UI still needs work (and will be updated at a later date) but the current version has the following models with the relationships shown below:

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-Diagram.png)

NetLearner database diagram

**Classes and Relationships**

The heart of the application is the [LearningResource](https://github.com/shahedc/NetLearner/blob/master/web/Models/LearningResource.cs) class. This represents any online learning resource, such as a blog post, single video, podcast episode, ebook, etc that can be accessed with a unique URL.

public class LearningResource : InternetResource  
{  
 public List<LearningResourceItemList> LearningResourceItemLists  
 {  
 get; set;  
 }  
}

The abstract class [InternetResource](https://github.com/shahedc/NetLearner/blob/master/web/Models/InternetResource.cs) defines the common properties (e.g. Id, Name and Url) found in any Internet resource, and is also used by other classes [ResourceRoot](https://github.com/shahedc/NetLearner/blob/master/web/Models/ResourceRoot.cs) and [RssFeed](https://github.com/shahedc/NetLearner/blob/master/web/Models/RssFeed.cs).

public abstract class InternetResource  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public string Url { get; set; }  
}

The **ResourceRoot** class represents a root-level resource (e.g. a blog home or a podcast website) while the **RssFeed** class represents the RSS Feed for an online resource.

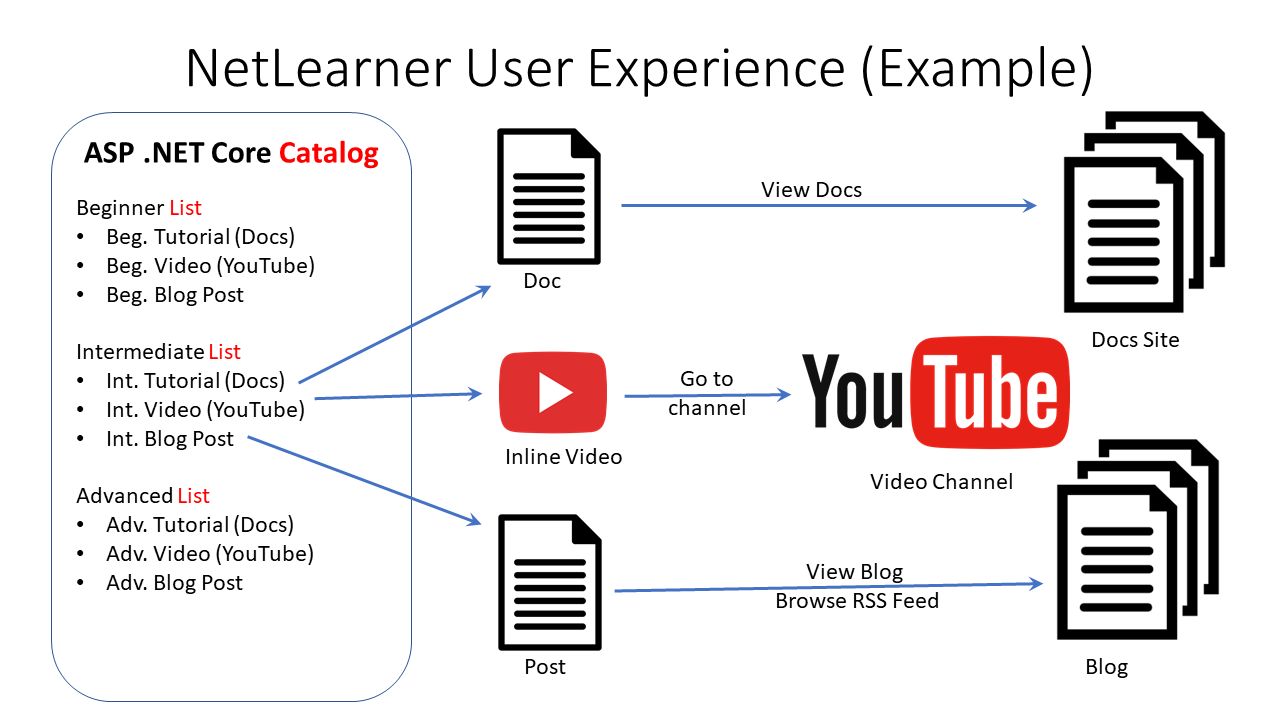
public class ResourceRoot: InternetResource  
{  
 public RssFeed RssFeed { get; set; }  
 public List<LearningResource> LearningResources { get; set; }  
} public class RssFeed: InternetResource  
{  
 public int ResourceRootId { get; set; }  
}

The [ItemList](https://github.com/shahedc/NetLearner/blob/master/web/Models/ItemList.cs) class represents a logical container for learning resources in the system. It is literally a list of items, where the items are your learning resources.

public class ItemList  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public List<LearningResourceItemList> LearningResourceItemLists  
 {  
 get; set;  
 }  
}

At this point, you may have noticed both the **LearningResource** and **ItemList** classes contain a List<T> property of **LearningResourceItemList**. If you browse the database diagram, you will notice that this table appears as a connection between the two aforementioned tables, to establish a many-to-many relationship. (more on this later)

The following diagram shows an example of how the a **LearningResource** is a part of a list (which is a part of a **ResourceCatalog**), while each **LearningResource** also has a **ResourceRoot**.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Example.png)

NetLearner example

**One to One**

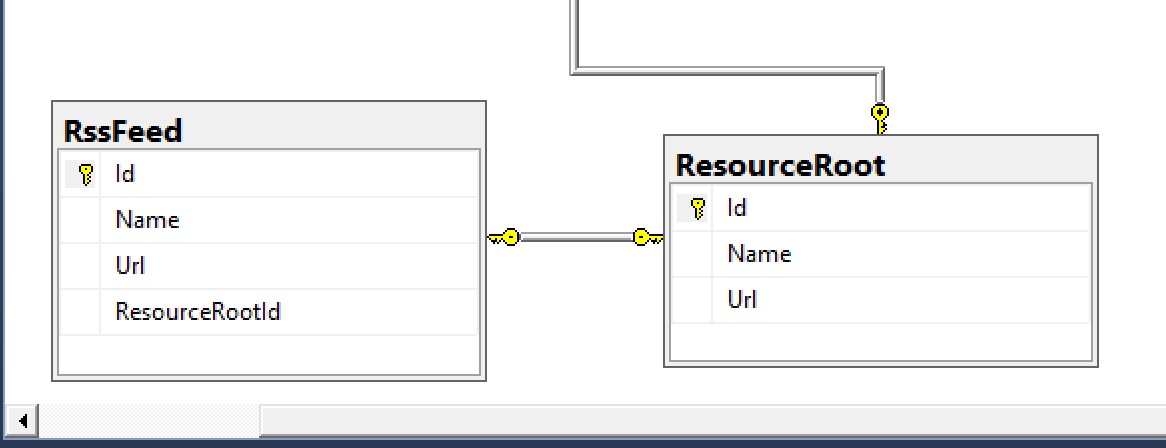
Having looked through the above entities and relationships, we can see that each ResourceRoot has an RssFeed. This is an example of a 1-to-1 relationship. For example:

* Resource Root = Wake Up and Code! blog site
* Rss Feed = RSS Feed for blog site

In the two classes, we see the following code:

public class ResourceRoot: InternetResource  
{  
 public RssFeed RssFeed { get; set; }  
 public List<LearningResource> LearningResources { get; set; }  
} public class RssFeed: InternetResource  
{  
 public int ResourceRootId { get; set; }  
}

Each Resource Root has a corresponding Rss Feed, so the **ResourceRoot** class has a property for **RssFeed**. That’s pretty simple. But in the **RssFeed** class, you don’t need a property pointing back to the **ResourceRoot**. In fact, all you need is a **ResourceRootId** property. EF Core will ensure that **ResourceRoot**.**Id** points to **RssFeed**.**ResourceRootId** in the database.

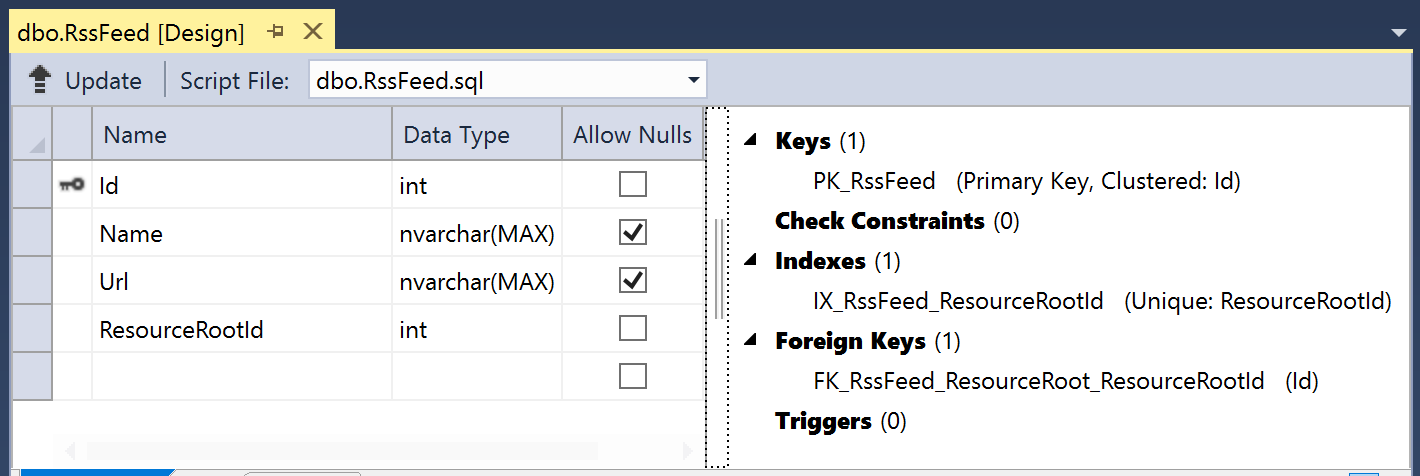
[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-One-to-One.png)

One to One Relationships

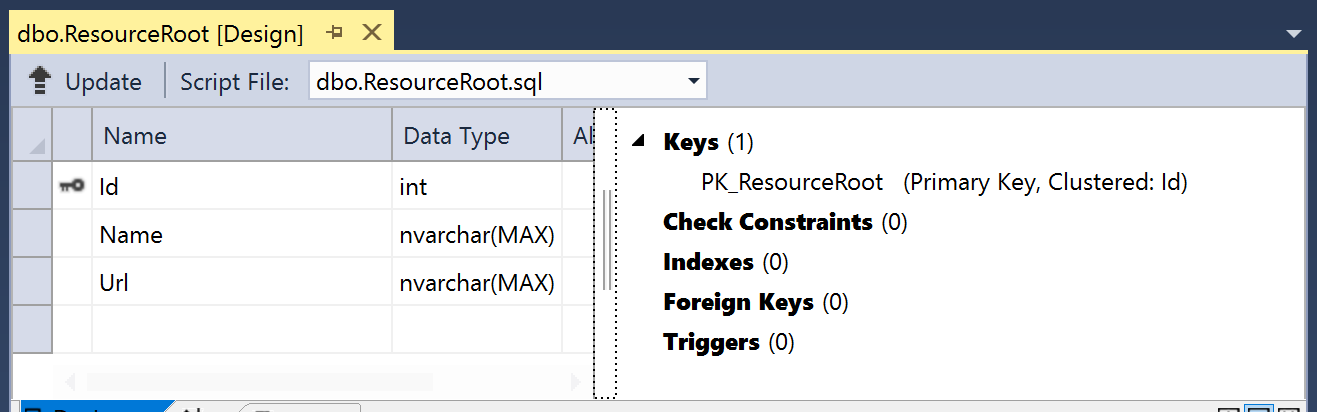
If you’re wondering how these two classes got their Id, Name and Url fields, you may recall that they are both derived from a common abstract parent class (InternetResource) that define all this fields for reuse. ***But wait a second***… why doesn’t this parent class appear in the database? That’s because we don’t need it in the database and have intentionally ommitted it from the list of DBSet<> definitions in the DB Context for the application, found in [ApplicationDbContext.cs](https://github.com/shahedc/NetLearner/blob/master/web/Data/ApplicationDbContext.cs):

public class ApplicationDbContext : IdentityDbContext  
{  
...  
 public DbSet<ItemList> ItemList { get; set; }  
 public DbSet<LearningResource> LearningResource { get; set; }  
 public DbSet<ResourceCatalog> ResourceCatalog { get; set; }  
 public DbSet<ResourceRoot> ResourceRoot { get; set; }  
 public DbSet<RssFeed> RssFeed { get; set; }  
...  
}

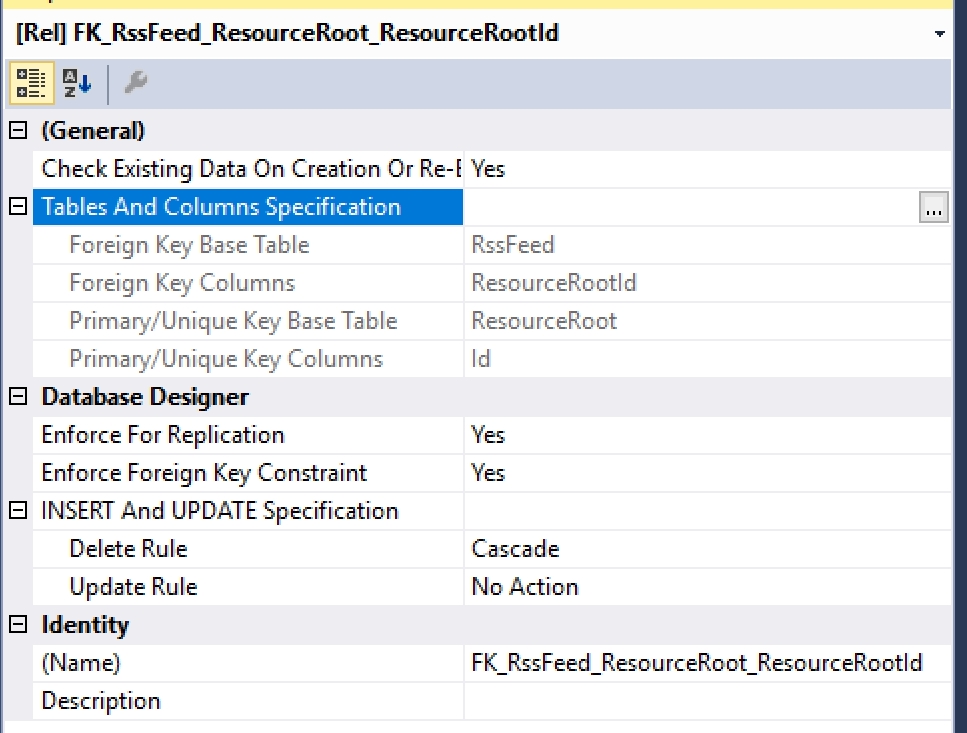
Another way of looking at the One-to-One relationship is to view the constraints of each database entity in the visuals below. Note that both tables have an Id field that is a Primary Key (inferred by EF Core) while the **RssFeed** table also has a Foreign Key for the **ResourceRootId** field used for the constraint in the relationship.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-RssFeed.png)

RssFeed table

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-ResourceRoot.png)

ResourceRoot Table

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-ResourceRoot-RssFeed.png)

1-to-1 Relationship: ResourceRoot.Id points to RssFeed.ResourceRootId

**One to Many (Example 1)**

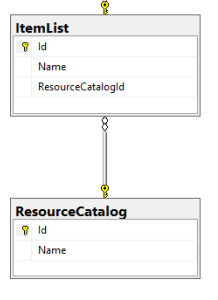
Next, let’s take a look at the One-to-Many relationship for each **ResourceCatalog** that has zero or more **ItemList**s. For example:

* Resource Catalog = ASP .NET Core Blogs
* Item List = ASP .NET Core A-Z Blog Series

In the two classes, we see the following code:

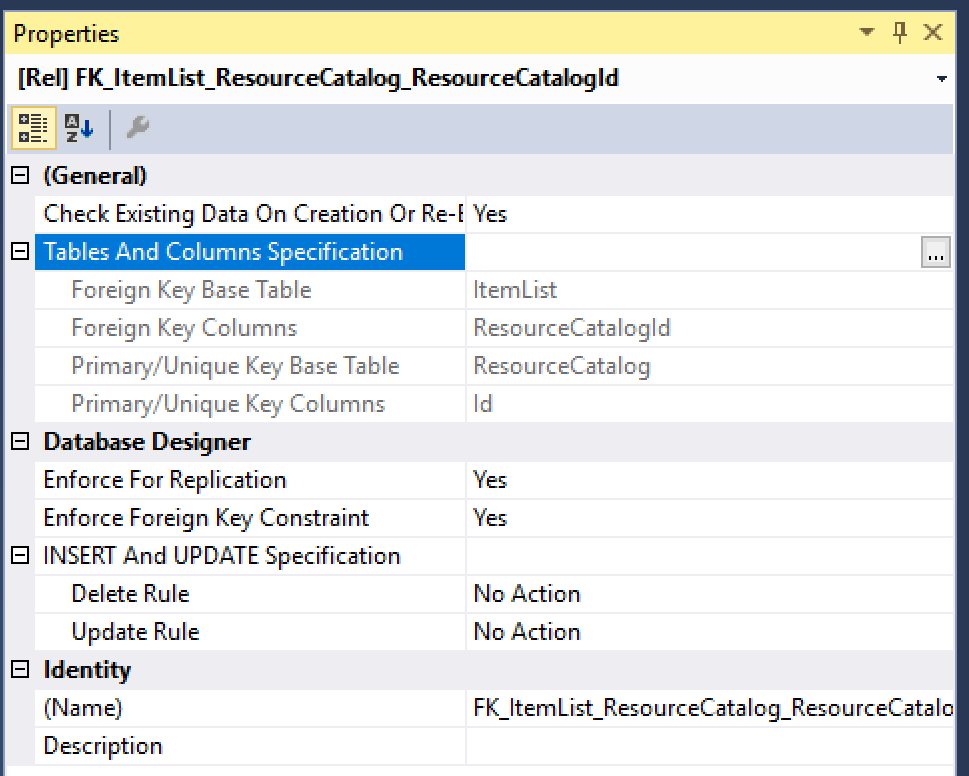
public class ResourceCatalog  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public List<ItemList> ItemLists { get; set; }  
} public class ItemList  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public List<LearningResourceItemList> LearningResourceItemLists  
 {  
 get; set;  
 }  
}

Each Resource Catalog has zero or more Item Lists, so the **ResourceCatalog** class has a List<T> property for **ItemLists**. This is even simpler than the previously described 1-to-1 relationship. In the **ItemList** class, you don’t need a property pointing back to the **ResourceCatalog**.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-ResourceCatalog-ItemList.png)

One-to-Many Relationship

Another way of looking at the *One-to-Many* relationship is to view the constraints of each database entity in the visuals below. Note that both tables have an Id field that is a Primary Key (once again, inferred by EF Core) while the **ItemList** table also has a Foreign Key for the **ResourceCatalogId** field used for the constraint in the relationship.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-ResourceCatalog-ItemList-Constraint.png)

1-to-Many Relationship: ResourceCatalog.Id points to ItemList.ResourceCatalogId

**One to Many (Example 2)**

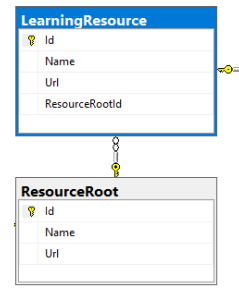
Let’s also take a look at another One-to-Many relationship, for each **ResourceRoot**that has zero or more **LearningResource**s. For example:

* Resource Root = Wake Up and Code! blog site
* Learning Resource = Specific blog post on site

In the two classes, we see the following code:

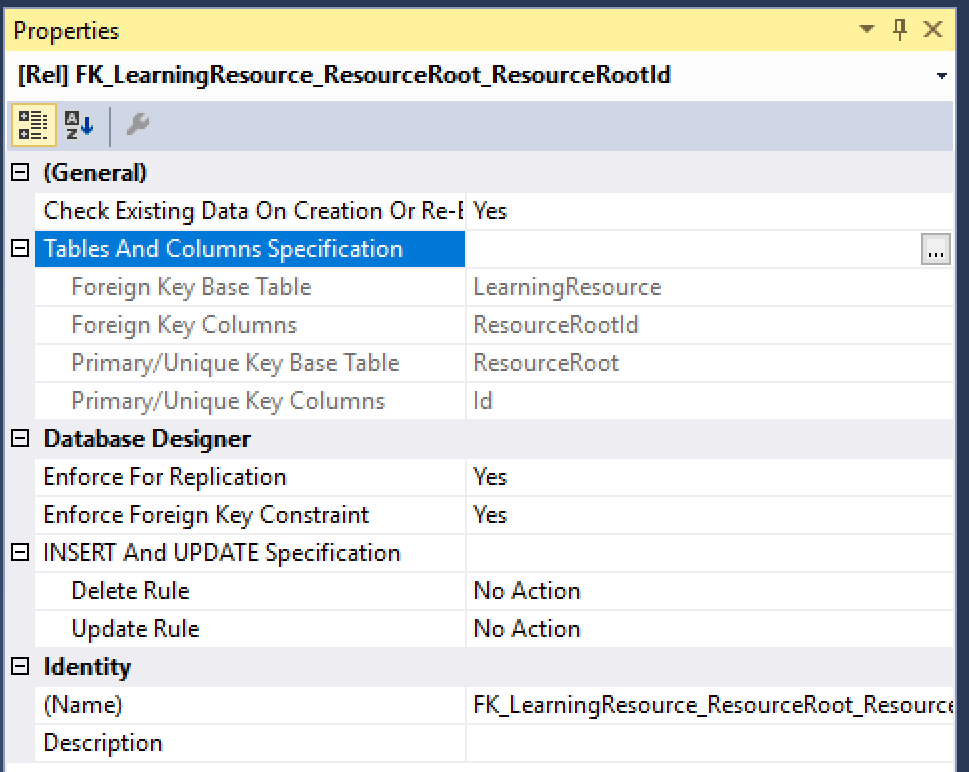
public class ResourceRoot: InternetResource  
{  
 public RssFeed RssFeed { get; set; }  
 public List<LearningResource> LearningResources { get; set; }  
} public class LearningResource : InternetResource  
{  
 public List<LearningResourceItemList> LearningResourceItemLists  
 {  
 get; set;  
 }  
}

Each Resource Root has zero or more Learning Resources, so the **ResourceRoot**class has a List<T> property for **LearningResources**. This is just as simple as the aforementioned 1-to-many relationship. In the **LearningResource**class, you don’t need a property pointing back to the **ResourceRoot**.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-ResourceRoot-LearningResource.png)

ResourceRoot and LearningResource

Another way of looking at the *One-to-Many* relationship is to view the constraints of each database entity in the visuals below. Note that both tables have an Id field that is a Primary Key (inferred by EF Core, as you should know by now) while the **LearningResource**table also has a Foreign Key for the **ResourceRootId** field used for the constraint in the relationship.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-ResourceRoot-LearningResource-Constraint.png)

1-to-Many Constraint for ResourceRoot and LearningResource

**Many to Many**

Finally, let’s also take a look at a Many-to-Many relationship, for each **ItemList** and **LearningResource,**either of which can have many of the other. For example:

* Item List = ASP .NET Core A-Z Blog Series
* Learning Resource = Specific blog post on site

This relationship is a little more complicated than all of the above, as we will need a “join table” to connect the two tables in question. Not only that, we will have to describe the entity in the C# code with connections to both tables we would like to connect with this relationship.

If you’re wondering when EF Core will support this type of relationship *without* a join table, check out the following GitHub issue discussions:

* Many-to-many relationships without an entity class to represent the join table: <https://github.com/aspnet/EntityFrameworkCore/issues/13009>
* Implement many-to-many relationships without mapping join table: <https://github.com/aspnet/EntityFrameworkCore/issues/10508>

Specifically, take a look at [this comment](https://github.com/aspnet/EntityFrameworkCore/issues/10508#issuecomment-389922109): *“Current plan for 3.0 is to implement skip-level navigation properties as a stretch goal. If property bags (*[*#9914*](https://github.com/aspnet/EntityFrameworkCore/issues/9914)*) also make it into 3.0, enabling a seamless experience for many-to-many could become easier.*”

In the two classes we would like to connect, we see the following code:

public class ItemList  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public List<LearningResourceItemList> LearningResourceItemLists  
 {  
 get; set;  
 }  
} public class LearningResource : InternetResource  
{  
 public List<LearningResourceItemList> LearningResourceItemLists  
 {  
 get; set;  
 }  
}

Next, we have the [LearningResourceItemList class](https://github.com/shahedc/NetLearner/blob/master/web/Models/LearningResourceItemList.cs) as a “join entity” to connect the above:

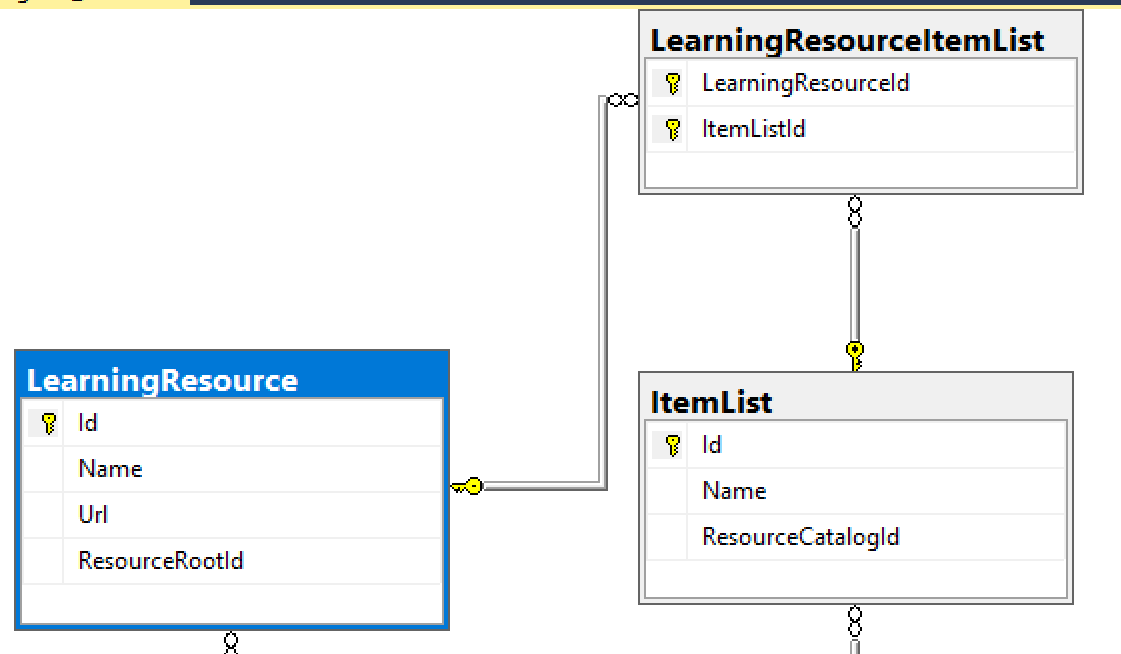
public class LearningResourceItemList  
{  
 public int LearningResourceId { get; set; }  
 public LearningResource LearningResource { get; set; }  
 public int ItemListId { get; set; }  
 public ItemList ItemList { get; set; }  
}

This special class has the following properties:

* **LearningResourceId**: integer value, pointing back to LearningResource.Id
* **LearningResource**: optional “navigation” property, reference back to connected LearningResource entity
* **ItemListId**: integer value, pointing back to ItemList.Id
* **ItemList**:  optional “navigation” property, reference back to connected **ItemList**entity

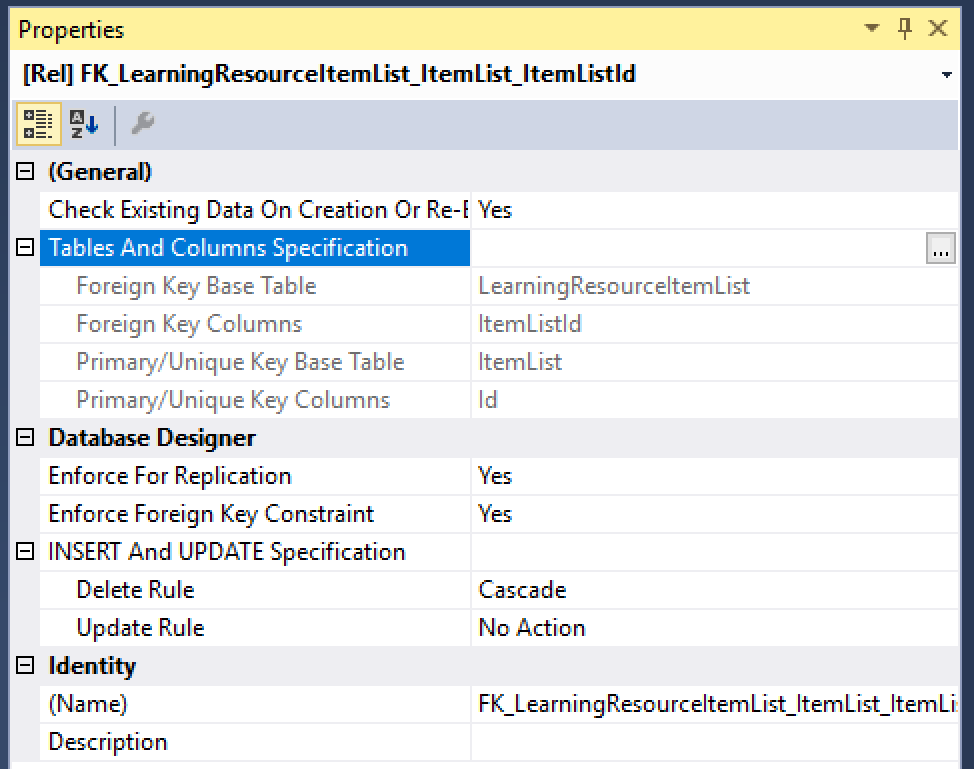
To learn more about navigation properties, check out the official docs at:

* Relationships – EF Core: <https://docs.microsoft.com/en-us/ef/core/modeling/relationships>

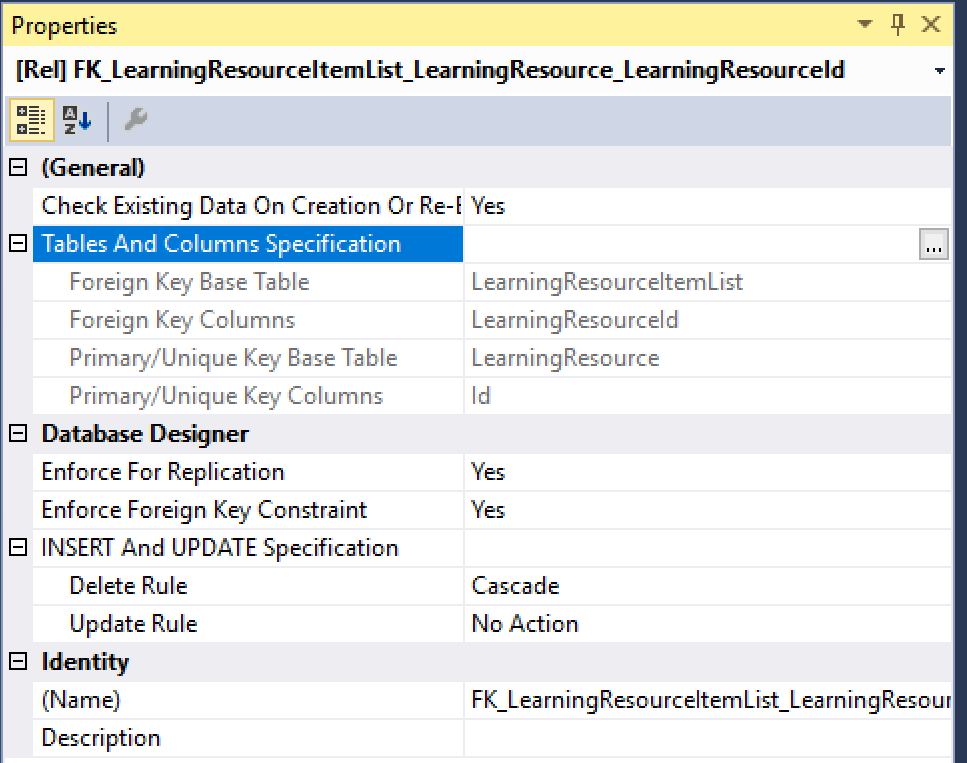
[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-Many-to-Many.png)

Many-to-Many Relationship

Another way of looking at the Many*-to-Many* relationship is to view the constraints of each database entity in the visuals below. Note that the two connected tables both have an Id field that is a Primary Key (yes, inferred by EF Core!) while the **LearningResourceItemList**table has a *Composite Key* for the **ItemListId and LearningResourceId**fields used for the constraints in the relationship.

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-ItemList-Many.png)

ItemList constraints

[](https://wakeupandcode.com/wp-content/uploads/2019/02/NetLearner-Db-LearningResource-Many.png)

LearningResource constraints

The composite key is described in the **ApplicationDbContext** class inside the **OnModelCreating**() method:

public class ApplicationDbContext : IdentityDbContext  
{  
 ...  
 protected override void OnModelCreating(ModelBuilder modelBuilder)  
 {  
 modelBuilder.Entity<LearningResourceItemList>()  
 .HasKey(r => new { r.LearningResourceId, r.ItemListId });  
 base.OnModelCreating(modelBuilder);  
 }  
}

Here, the [HasKey() method](https://docs.microsoft.com/en-us/ef/core/modeling/keys#fluent-api) informs EF Core that the entity **LearningResourceItemList** has a composite key defined by both **LearningResourceId** and **ItemListId**.

**References**

For more information, check out the list of references below.

* Relationships – EF Core: <https://docs.microsoft.com/en-us/ef/core/modeling/relationships>
* Keys – EF Core: <https://docs.microsoft.com/en-us/ef/core/modeling/keys>
* Introduction to Relationships: <https://www.learnentityframeworkcore.com/relationships>
* Julie Lerman on Pluralsight: <https://app.pluralsight.com/profile/author/julie-lerman>
* 2.0 Getting Started: <https://app.pluralsight.com/library/courses/entity-framework-core-2-getting-started>
* 2.0 Mappings: <https://app.pluralsight.com/library/courses/e-f-core-2-beyond-the-basics-mappings>
* 2.1 What’s New: <https://app.pluralsight.com/library/courses/playbook-ef-core-2-1-whats-new>

For detailed tutorials that include both Razor Pages and MVC, check out the official tutorials below:

* New database – EF Core: <https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/new-db?tabs=visual-studio>
* Existing Database – EF Core: <https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/existing-db>
* ASP.NET Core MVC with EF Core: <https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc>
* ASP.NET Core Razor Pages with EF Core: <https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp>