<https://www.tektutorialshub.com/lazy-loading-entity-framework/>

static void LazyLoading()

{

using (AdventureWorks db = new AdventureWorks())

{

db.Database.Log = Console.Write;

var product = (from p in db.Products

where p.ProductID == 814

select p).ToList();

foreach (var p in product)

{

Console.WriteLine("{0} {1} {2}", p.ProductID, p.Name, p.ProductModel.Name);

}

Console.ReadLine();

}

}

The Code above retrieves the Product from the database when the ToList() method is called. The Entity framework does not retrieve the Product model at that time. The Product model name is accessed from inside the for the loop. At that point, Entity Framework sends the query to the database to retrieve the ProductModel. This behaviour is known as Lazy Loading.

<https://www.tektutorialshub.com/eager-loading-entity-framework/>

using (AdventureWorks db = new AdventureWorks())

{

//Disable Lazy Loading

db.Configuration.LazyLoadingEnabled = false;

//Log SQL Command to Console

db.Database.Log = Console.Write;

var product = (from p in db.Products

.Include("ProductModel") //ProductModel table to be included in the result

where p.ProductID == 814

select p).ToList();

foreach (var p in product)

{

Console.WriteLine("{0} {1} {2}", p.ProductID, p.Name, p.ProductModel.Name);

}

}

Eager loading solves this problem by loading all the product models along with products using a single query. Eager loading does this by using the Include method. Entity Framework creates a join query, when it sees the Include method, thus bringing all the records in one single query.

<https://www.tektutorialshub.com/explicit-loading-entity-framework/>

using (AdventureWorks db = new AdventureWorks())

{

//Disable Lazy Loading

db.Configuration.LazyLoadingEnabled = false;

//Log Database

db.Database.Log = Console.Write;

//List of Products queried here.

var product = (from p in db.Products

orderby p.ProductID descending

select p).Take(5).ToList();

foreach (var p in product)

{

//Product model is retrieved here

db.Entry(p).Reference(m => m.ProductModel).Load();

Console.WriteLine("{0} {1} Product Model => {2}", p.ProductID, p.Name, ( p.ProductModel==null) ? "" : p.ProductModel.Name );

Console.ReadKey();

}

}

In the above query, we retrieve five products from the database. Inside the for loop of each product, we use load method to retrieve the product model data

The Microsoft.Web.WebPages.OAuth namespace contains core classes that are used to work with OAuth and OpenID authentication.

The Microsoft.Owin.Security.OAuth namespace contains types related to OAuth providers.

**Bundling** is a new feature in ASP.NET 4.5 that makes it easy to combine or bundle multiple files into a single file. **Minification** performs a variety of different code optimizations to scripts or css, such as removing unnecessary white space and comments and shortening variable names to one character.

bundles.Add(new ScriptBundle("~/bundles/jquery").Include("~/Scripts/jquery-{version}.js"));  
<%: Scripts.Render("~/bundles/jquery") %>

ScriptBundle thirdPartyScripts = new ScriptBundle("~/Scripts/ThirdParty");  
thirdPartyScripts.Include("~/Scripts/jquery-{version}.js","~/Scripts/bootstrap.min.js");  
bundles.Add(thirdPartyScripts);

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

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

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

Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks.

Defining Attributes in Metadata Classes

System.ComponentModel.DataAnnotations

@\*To call same same controller view\*@

@Html.ActionLink("Index", "ActionName")

@\*To call another controller view\*@

@Html.ActionLink("Index", "ActionName", "ControllerName", null, new { id = "OT", style = "color: white”})

routes.MapRoute(

name: "DefaultRoute",

url: "Home/{action}",

defaults: new { controller = "Home", action = "Index" },

constraints: new { action = "[A-Za-z]\*" }

);