



Case Study:

Optimizing Data Access

with Microsoft Entity Framework

Tech Challenger series

by Kwork Innovations



Antti Törrönen is a Microsoft Certified Professional Developer with a long experience in web development starting mid-90's with Pascal and moving on to Microsoft ASP in late-90's. After a career in e-commerce and distribution he is currently team lead in a technology company building web service ecosystems based on patented technology.

Kwork Innovations

Kwork Innovations is a challenger in the web technology space. We aim to stop boring websites. Say yes to cloud services that serve!



Secure Software Development

Kwork Innovation was the first company in Finland to achieve Development Process Security Certificate at Level 2. The audit was done by 2NS.



Code Quality from Finland

Kwork is a member of Code From Finland community, National Software Association, Chamber of Commerce, as well as a registered Microsoft Partner. Our team has been working with web for a long time – all the way from 1996. In current form we've operated since 2014



Strong Research & Development focus

Our digitalization platform has been granted the first patent, with others pending. Our research and development efforts have gained multiple grants and participation in various support programs. You may have seen our products at TechCrunch Disrupt, Aalto Make It Digital, NFOpen, Start TLV Helsinki and Slush among others.



Strong hosting

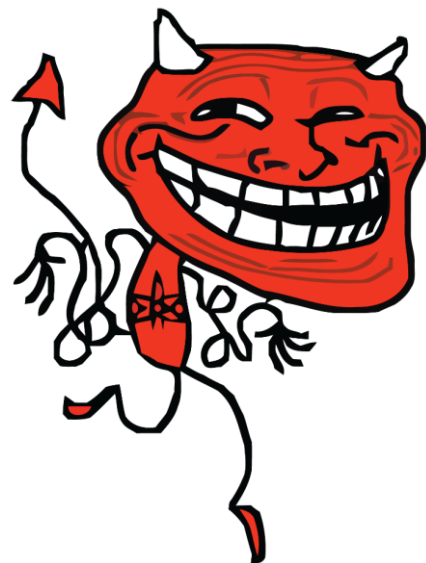
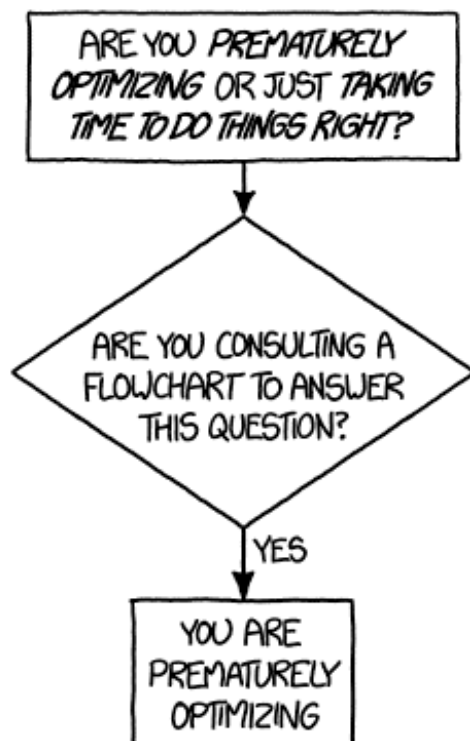
We use Microsoft Azure for cloud hosting as well as dedicated Windows Servers for fast, secure and scalable hosting.



Certified Expertise

Web communications and web technology are a wide area of topics. We believe it is important to have a professional understanding of the topics. To demonstrated professionalism we are certified in multiple Microsoft technologies, Google Analytics and Adwords and Inbound marketing among others.

*"We should forget about small efficiencies, say about 97% of the time: **premature** optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"*



*Come on! **Premature** optimization feels so good! Just do it!*

Case Study

Improving data access

This is a case study based on code review of a ASP.NET MVC 5 website experiencing slow performance. The purpose is to introduce various techniques for improving data access through Entity Framework. Each method is demonstrated within the same project to exemplify that numerous options exists. The correct course of action depends on a variety of situational elements, most notably the complexity of data structures and the amount of data fetched.

7 methods and 3 bonus methods for improving performance are introduced. The case was solved by combination of these methods, except 1-3. Methods 1-3 are introduced as they may be suitable in other cases.

Case description

The website is a specialty blog with 50-250 articles. Each article includes also images, list of references and links. In addition, each article is categorized to 1 or more categories.

The problem is identified in main page of the site. This page includes buttons for showing the different articles dynamically after clicking. The current load time for the page is 10-30 seconds.

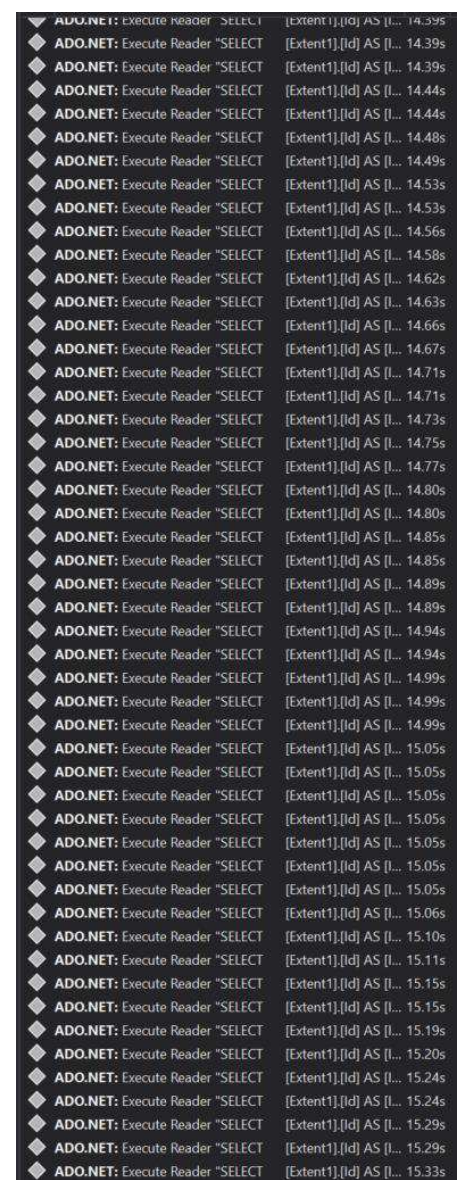
Current flow of the page is:

- Fetch all articles from database
- Fetch all links, images and categories for each article
- Add the article, links, images and categories to a list in a View Model
- All articles are set on the page as hidden div's which are dynamically shown based on user actions

The purpose of this approach has been to make the website a single page application without need for page reloads.

However, this approach had made the site very slow to download, mostly due to

- The page is heavy to download
- Over 200 sequential SQL queries are observed in Visual Studio



ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.59s	[Extent1].[Id] AS [I... 14.59s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.39s	[Extent1].[Id] AS [I... 14.39s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.39s	[Extent1].[Id] AS [I... 14.39s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.44s	[Extent1].[Id] AS [I... 14.44s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.44s	[Extent1].[Id] AS [I... 14.44s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.48s	[Extent1].[Id] AS [I... 14.48s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.49s	[Extent1].[Id] AS [I... 14.49s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.53s	[Extent1].[Id] AS [I... 14.53s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.53s	[Extent1].[Id] AS [I... 14.53s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.56s	[Extent1].[Id] AS [I... 14.56s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.58s	[Extent1].[Id] AS [I... 14.58s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.62s	[Extent1].[Id] AS [I... 14.62s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.63s	[Extent1].[Id] AS [I... 14.63s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.66s	[Extent1].[Id] AS [I... 14.66s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.67s	[Extent1].[Id] AS [I... 14.67s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.71s	[Extent1].[Id] AS [I... 14.71s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.71s	[Extent1].[Id] AS [I... 14.71s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.73s	[Extent1].[Id] AS [I... 14.73s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.75s	[Extent1].[Id] AS [I... 14.75s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.77s	[Extent1].[Id] AS [I... 14.77s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.80s	[Extent1].[Id] AS [I... 14.80s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.80s	[Extent1].[Id] AS [I... 14.80s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.85s	[Extent1].[Id] AS [I... 14.85s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.85s	[Extent1].[Id] AS [I... 14.85s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.89s	[Extent1].[Id] AS [I... 14.89s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.89s	[Extent1].[Id] AS [I... 14.89s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.94s	[Extent1].[Id] AS [I... 14.94s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.94s	[Extent1].[Id] AS [I... 14.94s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.99s	[Extent1].[Id] AS [I... 14.99s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 14.99s	[Extent1].[Id] AS [I... 14.99s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.05s	[Extent1].[Id] AS [I... 15.05s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.05s	[Extent1].[Id] AS [I... 15.05s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.05s	[Extent1].[Id] AS [I... 15.05s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.05s	[Extent1].[Id] AS [I... 15.05s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.05s	[Extent1].[Id] AS [I... 15.05s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.05s	[Extent1].[Id] AS [I... 15.05s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.06s	[Extent1].[Id] AS [I... 15.06s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.10s	[Extent1].[Id] AS [I... 15.10s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.11s	[Extent1].[Id] AS [I... 15.11s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.15s	[Extent1].[Id] AS [I... 15.15s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.15s	[Extent1].[Id] AS [I... 15.15s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.19s	[Extent1].[Id] AS [I... 15.19s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.20s	[Extent1].[Id] AS [I... 15.20s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.24s	[Extent1].[Id] AS [I... 15.24s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.24s	[Extent1].[Id] AS [I... 15.24s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.29s	[Extent1].[Id] AS [I... 15.29s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.29s	[Extent1].[Id] AS [I... 15.29s
ADO.NET: Execute Reader "SELECT [Extent1].[Id] AS [I... 15.33s	[Extent1].[Id] AS [I... 15.33s

Screenshot from Events window in Visual Studio debug mode shows hundreds of database calls

Starting point

Code to be refactored

```
/// <summary>
/// Get the main view with published articles
/// </summary>
/// <returns>the articles in single page app format</returns>
public ActionResult Index()
{
    List<Data.Article> articlesByAdmin = this.db.Articles
        .Where(a => a.PublishStatus == Status.AdminPublished).ToList();
    List<Data.Attachment> images = new List<Attachment>();
    List<Data.Link> links = new List<Link>();
    List<MainVM> articles = new List<MainVM>();
    List<Data.Category> searchTerms = new List<Category>();

    foreach (Data.Article a in articlesByAdmin)
    {
        article = new MainVM
        {
            Description = a.Description;
            ...
        };

        images = this.db.Attachments
            .Where(t => t.Article.Id == a.Id).ToList();
        article.ImageURLs = images;

        links = this.db.Links
            .Where(l => l.Article.Id == a.Id).ToList();
        article.Links = links;

        searchTerms = this.db.Categories
            .Where(t => t.Article.Id == a.Id).ToList();
        List<string> terms = new List<string>();

        foreach (var t in searchTerms)
        {
            terms.Add(t.CategoryName);
        }
        article.SearchTerms = terms;

        articles.Add(article);
    }

    art.Articles = articles;
    return View(art);
}
```

Method 1: .Select()

Limit number of fields fetched

In some cases the database contains a lot of data that is not needed all the time. In case of articles or blogs a common task is to only list the titles of the blog posts. There is no need to fetch large amount of data from the text section!

Use .Select() method. Select() will allow you to create a new object, and select which properties you will need to fetch. The result of .Select() is a new query tree, IQueryable. This will be compiled to SQL and executed when you call one of the non-deferred methods, e.g. .ToList(), .Single(), .First()

When using .select() you can either use the EF entity class, or you can create your own class. If you need just a single element, use .select(o => o.Title).ToList() to get a List<T> where T is the type for o.Name. This example will result in a List<string> containing titles.

In the example, we are fetching the data directly to the view model for Article.

```
articlesByAdmin = this.db.Articles
    .where(a => a.PublishStatus == Status.AdminPublished)
    .select(a => new ArticleVM
    {
        ArticleId = a.Id,
        ArticleTitle = a.Title
    });
```

Do this only when you know which fields are needed.
Avoid doing this when you are still unsure about the needed fields



Method 2: Parallel.For & Parallel.ForEach

Parallel looping

The code contains a loop which is run sequentially – a single SQL command must be executed before beginning the next one.

.NET 4.5. provides Task Parallel Library TPL, which has some easy and cool functions for parallel programming. The main concern is that all shared data structures must be **thread-safe**, which is indicated in MSDN documentation for the .NET library. DataContext is not thread-safe. Thus, we must create a new DataContext for each iteration.

The below code replaces the for loop with a parallel loop. The number of simultaneous threads depends on the number of cores available. You may also set the number manually in a overload of the ForEach function. The optimal number of threads depends on how much CPU intensive the function is (or how long each iteration is waiting for other I/O operations)

As these are easy to use function it is also very easy run parallel loop within a parallel loop – thus slowing the performance with high number of threads.

```
Parallel.ForEach(articlesByAdmin, (Data.Article a) =>
{
    article = new MainVM
    {
        Description = a.Description;
        ...
    };
    ArticleDb db = new ArticleDb();

    images = db.Attachments
        .where(t => t.Article.Id == a.Id).ToList();
    article.ImageURLs = images;

    ...

    articles.Add(article);
}
```

Method 3: ToListAsync()

Asynchronous SQL calls

Entity Framework 6 allows for asynchronous calls. The asynchronous calls are best for starting a data call, and then doing other CPU intensive operations while waiting for the return. Data Context support only one operation at a time, thus we need to create separate data contexts for each asynchronous call.

.ToListAsync() returns a Task object. A Task object has a property called .Result. Accessing this property will wait for the result to complete, and then returns the call.

```
ArticleDb db1 = new ArticleDb();
ArticleDb db2 = new ArticleDb();
ArticleDb db3 = new ArticleDb();

Task<List<Attachment>> images = db1.Attachments
    .where(t => t.Article.Id == a.Id).ToListAsync();
Task<List<Link>> links = db2.Links
    .where(l => l.Article.Id == a.Id).ToListAsync();
Task<List<Category>> searchTerms = db3.Categories
    .where(t => t.Article.Id == a.Id).ToListAsync();

article.ImageURLs = images.Result;
article.Links = links.Result;
List<string> terms = searchTerms.Result
    .Select(o => o.CategoryName).ToList();
article.SearchTerms = terms;
```


Method 4: Include()

Fetch all data at once

Entity Framework allows to fetch related data from different entity sets using the `.Include()` method. `Include()` will modify the query tree so that when the call is executed at `.ToList()` the related data is also fetched. This way, you can access `article.LinkSet` without any further database calls.

`List<T>` is thread-safe you might further optimize the below by using `Parallel.For` if you have unused CPU cores (observe CPU usage)

```
articlesByAdmin = this.db.Articles
    .where(a => a.PublishStatus == Status.AdminPublished)
    .Include("AttachmentSet")
    .Include("CategorySet")
    .Include("LinkSet")
    .ToList();

foreach(Article article in articlesByAdmin)
{
    article = new MainVM()
    {
        Links = a.LinkSet.Select( o => o.Url).ToList(),
        Description = a.Description;
        ...
    }; ...
}
```

For the `.Include()` method to work the EF model must know about the connections between tables. In this case, we had to add the following snippet to Code-first model, because the current code used a manual way to fetch related items using the foreign key (`ArticleId`). A model-first approach would need drawing the connection.

The below will also allow lazy-loading. If you fetch `AttachmentSet` without the above `.Include()` method run earlier a new data access call is made to fetch the data. This is an important concepts to understand. Lazy-loading without `.Include()` might cause a large number of unnecessary database calls because the data is fetched only at the time when needed.

```
public virtual ICollection<Attachment> AttachmentSet { get; set; }

public virtual ICollection<Link> LinkSet { get; set; }

public virtual ICollection<Category> CategorySet { get; set; }
```

We prefer using Set suffix instead of plural forms for collections. A collection of categories is called `CategorySet`.

Method 4: [OutputCache]

Use output cache

The most powerful cache on the server is the output cache. This cache stores the result of the Action method, and thus it does not need to be executed on each call.

Output cache is enabled in web.config, IIS settings or with [OutputCache] annotation. The annotation allows for setting the duration of cache in settings and having different outputs based on one of the Action method parameters.

There is a caveat, though. There is not easy way to invalidate the output cache when the contents of the database change. It is best to set the duration to fairly low, and only enable output caching on high traffic sites with no or few dynamic elements. Also, avoid caching pages with personal data – or at least, disable caching headers so browser or proxies won't store the page.

"There are only two hard things in Computer Science: cache invalidation and naming things."
– a popular saying, possibly by Phil Karlton

```
/// <summary>
/// Get the main view with published articles
/// </summary>
/// <returns>the articles in single page app format</returns>
[OutputCache(Duration=60, VaryByParam="none")]
public ActionResult Index()
{
```

Output cache may introduce problems with the site not reflecting latest updates. Avoid using output cache until you know which should be the parameters to vary based on.



Method 5: MemoryCache

Use object cache

All object can be stored on a cache on the web server. The cache must fit in the RAM so you may want to observe RAM usage on the server. My experience has been though that many web servers have plenty unused RAM, and use of object cache could improve performance and take some load of the SQL server.

.NET has a abstract class `ObjectCache` which you may inherit to create your own variations. .NET in-memory cache implementation is called `MemoryCache`. It's use is simple: try to fetch from `MemoryCache`. If the return is null, then make the database call and add to cache for the next call.

`MemoryCache` supports multiple invalidation methods: absolute expiration, sliding expiration, and change track elements. In this simple example, we will simply use a sliding expiration.

```
ObjectCache cache = MemoryCache.Default;
List<Data.Article> articlesByAdmin =
    cache["articles"] as List<Data.Article>;

if (articlesByAdmin == null)
{
    articlesByAdmin = this.db.Articles
        .Where(a => a.PublishStatus == Status.AdminPublished)
        .Include("Attachments")
        .Include("Categories")
        .Include("Links")
        .ToList();

    CacheItemPolicy policy = new CacheItemPolicy()
    { SlidingExpiration = new TimeSpan(0, 0, 60) };

    cache.Set("articles", articlesByAdmin, policy);

    // Use MemoryCache.Default.Remove("articles");
    // to clear cache when database is updated
}
```

Use of output cache requires that whenever database is updated the cache is cleared. If you the database is updated from multiple sources you may introduce new problems



Method 6: Select() with sub-queries

Select all & only needed fields

LINQ is powerful! A Internet joke is that the worst part of .NET is that you will want to spend the whole day rewriting the whole program in a single LINQ statement just because you can. In fact, most of the LINQ commands return a query tree (IQueryable) which is then executed and compiled. This allows to write subqueries. In this examples we replace the for loop by including subqueries in the main call.

We are using LINQ method syntax. For more complex queries, see LINQ statement syntax which resembles SQL.

```
articlesByAdmin = this.db.Articles
    .Where(a => a.PublishStatus == Status.AdminPublished)
    .Select(a => new MainVM
    {
        ArticleId = a.Id,
        ArticleTitle = a.Title,
        ArticleUrl = a.FriendlyURL,
        ArticleDescription = a.Description,
        PublishStatus = a.PublishStatus,
        Language = a.Language,
        Links = a.Links.ToList(),
        ImageURLs = a.Attachments.ToList(),
        SearchTerms = a.Categories
            .Select(o => o.CategoryName).ToList()
    }).ToList();
```

Method 7:

AJAX calls

In this case, the most obvious way to improve performance is to delay the database calls for articles to the point when client requests them. This is called AJAX: Asynchronous JavaScript and XML. When the user clicks on a "Show article" button a JavaScript call is made to request the article data from the server. For brevity, we use JQuery on client-side. Remember to install JQuery library, or see youmightnotneedjquery.com to adapt the code.

AJAX calls are a good option when only part of the data is visible until user requests it, there is a stable Internet connection and there is large number of data (e.g. long articles). The disadvantage of AJAX call is that the web server must respond to a new HTTP request every time the users wants to see another element.

The below sample show the JavaScript syntax on the page and the ASP.NET method used for returning the AJAX data.

```
public ActionResult GetArticle(int id)
{
    // Not cached currently, but object cache should
    // be considered due to low number
    // of articles (<500) which will fit in RAM

    ArticleVM article = this.db.Articles
        .Where(a => a.Id == id)
        .Select(a => new ArticleVM
        {
            ArticleId = a.Id,
            Title = a.Title,
            Url = a.FriendlyURL,
            Description = a.Description,
            PublishStatus = a.PublishStatus,
            Language = a.Language,
            Links = a.Links.ToList(),
            ImageURLs = a.Attachments.ToList(),
            SearchTerms = a.Categories
                .Select(o => o.CategoryName)
                .ToList()
        })
        .Single();
    return Json(article, JsonRequestBehavior.AllowGet);
}
```


Below is the JavaScript call used for making the AJAX call and updating the retrieved data in correct locations.

```
@* Get elements used for the article *@
var articleTitle = document.getElementsByClassName("title");
var articleContent = document.getElementsByClassName("content");
var articleLinks = document.getElementsByClassName("links");
var articleImages = document.getElementById("images");

$.ajax({
    url: "/Content/GetArticle",
    type: "POST",
    data: JSON.stringify({ 'id': id }),
    dataType: "json",
    traditional: true,
    contentType: "application/json; charset=utf-8",
    success: function (data) {

        @*Looping through in case there are multiple
        elements needs to be updated*@
        for(var i = 0, len = articleTitle.length; i < len; i++) {
            articleTitle[i].innerText = data.Title;
        }
        for(var i = 0, len = articleContent.length; i < len; i++) {
            articleContent[i].innerHTML = data.Description;
        }
        for(var i = 0, len = articleLinks.length; i < len; i++) {
            articleLinks[i].innerHTML = getLinks(data.Links);
        }

        articleImages.innerHTML = getImages(data.ImageURLs);
        articleShare.innerHTML = getUrl(data.ArticleUrl);
    },
    error: function () {
        alert("An error has occurred");
    }
});
```

Finally, we removed the data distributed through AJAX from the main data query.

```
articlesByAdmin = this.db.Articles.Where(a => a.PublishStatus == Status.AdminPublished)
.Select(a => new ArticleVM
{
    ArticleId = a.Id,
    ArticleTitle = a.Title,
    ArticleUrl = a.FriendlyURL,
    ArticleDescription = a.Description.Substring(0, 134),
    PublishStatus = a.PublishStatus,
    Language = a.Language,
    Links = a.Links.ToList(),
    ImageURLs = a.Attachments.ToList(),
    SearchTerms = a.Categories
        .Select(o => o.CategoryName).ToList()
}).ToList();
```

Further methods:

LINQ commands

Call `.SaveChanges()` or `.SaveChangesAsync()` once

Let Entity Framework Data Context track the changes for you – because that is what it was meant for! Everytime you call `SaveChanges()` a SQL INSERT command is sent to database, and the non-async version will return for the response. To save time save all changes once. Note: you must call `SaveChanges` for every data context you may use separately.

Use `Find()` instead of `Where().Single()`

When you are using primary key to search for items, prefer `Find()` instead of `Where().First()` or `Where.Single()`. Not only does it show the intent of the call more clearly it only is (very) slightly faster. `First()` and `Single()` both use an iterator to access the collection (`IEnumerable` interface) which will cause slight overhead.

Understand `Single()` vs. `First()`

`Single()` will verify that only one items exists in the collection. `First()` will return first element in the collection. `Single()` is useful for catching potential problems in the database when you expect only one element to be present. Both will throw an exception if no items exist. When you are unsure if elements exist, use `SingleOrDefault()` or `FirstOrDefault()` which will return null if no case no elements are found. Checking for null is faster than catching exceptions.

Use `.Skip()` and `.Take()` for paging

When paging items remember to use `Skip()` and `Take()` instead of first getting the whole collection and using c# logic to page...

`AddRange()` and `DeleteRange()` for collections

`.Add()` command will call `DetectChanges()` in the context every time. However, when you use `AddRange()` or `DeleteRange()` this command will be only ran once.

```
IList<Product> products = new List<Product>();
newStudents.Add(new Product() { Title="Coffee Maker" });
newStudents.Add(new Product() { Title="Teapot" });

using (var db = new ProductDb())
{
    db.Students.AddRange(newStudents);
    db.SaveChanges();
}
```

EF Optimizations

using or remember to Dispose the connection

Remember to close connections you use. Easiest way is the using construct. Alternatively, call BlogDb.Dispose().

```
using (BlogDb context
    = new BlogDb())
{
    // run db commands
}
```



Ensure your codebase is stable and mature enough to employ the below optimization methods. These may cause more problems than they are worth during the development phase...

.AsNoTracking()

This LINQ command will make Entity Framework tracking any changes on the result set. This is useful when you only want to read data and you don't expect to make any updates. While you may achieve small performance improvements this may also lead to hard to debug problems if requirements change in the future, and .AsNoTracking() is not seen.

Disable AutoDetectChanges for bulk updates

You can disable automatic detection of changes to save time. You may also disable AutoDetectChanges for bulk updates, and then re-enable it after the updates. You can also call dbContext.DetectChanges() manually.

```
dbContext.Configuration.AutoDetectChangesEnabled = false;
BulkUpdatesAndInserts(dbContext);
dbContext.Configuration.AutoDetectChangesEnabled = true;
```

Disable ValidateOnSaveEnabled

For minor performance improvement you may disable validation during SaveChanges() if you know the data is correct. You may also disable it to circumvent validation in Entity Framework models. The problem is that SQL command will save in the database server if the data does not fulfill database validation requirements. Getting the error back from SQL Server is way slower.

```
dbContext.Configuration.ValidateOnSaveEnabled = false;
```

Split database to multiple EF models

Entity Framework does not need to know about every table that is in your database. You may be able to improve maintainability, and slightly performance, by splitting your data model to multiple small entity models (data context's).

SQL Server optimization

Write direct SQL commands

Entity Framework is a good option for writing a large amount of varying data calls. However, it will also add considerable overhead. Use direct SQL for reading large amounts of data or bulk updates. Just understand the limitations, e.g. higher vulnerability to SQL injection attacks. Entity Framework has two commands .SqlQuery and .ExecuteSqlCommand for sending direct SQL Command. When SqlQuery is run from the correct contity class, EF will map the data to the correct classes.



```
using (var context = new BloggingContext())
{
    List<Blog> blogs = context.Blogs
        .SqlQuery("SELECT * FROM dbo.BlogSet").ToList();
    context
        .Database.ExecuteSqlCommand(
            "UPDATE dbo.BlogSet SET IsDisabled=1");
}
```

Use stored procedures

SQL Server will need to parse SQL commands – as they are only text - and SQL optimizer will then estimate what is the fastest execution strategy. This creates overhead which can be overcome by using pre-built stored procedures. Stored procedures can be precompiled, and they may also provide security benefits.



You can map stored procedures in Entity Framework Model Editor (EDMX Model) or in code-first classes (EF6+ only). Code-first has two options:

- 1) Use conventions and name procedures in format
<type_name>_Insert, <type_name>_Delete,
<type_name>_Update and parameters with same name as properties.
- 2) Manually configure stored procedures as below

```
// Use naming conventions for stored procedures
modelBuilder.Entity<Blog>().MapToStoredProcedures();

// Manually, set stored procedure names and methods
modelBuilder.Entity<Blog>().MapToStoredProcedures(s => {
    s.Update(u => u.HasName("modify_blog_url")
        .Parameter(b => b.BlogId, "blog_id")
        .Parameter(b => b.Url, "blog_url"));
    s.Delete(d => d.HasName("delete_blog")
        .Parameter(b => b.BlogId, "blog_id"));
    s.Insert(i => i.HasName("insert_blog")
        .Parameter(b => b.Name, "blog_name")
        .Parameter(b => b.Url, "blog_url"));
});
```

Optimization outside EF & LINQ

Use SqlBulkCopy or direct SQL for bulk insert/update / fetch

Entity Framework has overhead for every call as LINQ is ran against the EF datamodel which is translated to SQL. Thus, this will be slower. Luckily, System.Data.SqlClient has function SqlBulkCopy for bulk SQL commands.

Know when to use nvarchar and varchar

nvarchar stores Unicode data, and you almost always want to use it to enable internationalization in the future. However, nvarchar uses 2 bytes per data which means less can be indexed. Maximum index size in SQL Server is 900 bytes. You may use varchar for ASCII-only characters to get multiple columns included for faster exact match queries.



Verify you use nvarchar or varchar in both EF and SQL Server

Entity Framework uses by default nvarchar. If you, however, decide to use varchar also setup Entity Framework to use the same. You can do it by one these ways in EF code-first:

1. `[Column(TypeName = "VARCHAR")]`
2. `modelBuilder.Properties<string>().Configure(c => c.HasColumnType("varchar"));`
3. `modelBuilder.Properties<string>().Configure(c => c.IsUnicode(false));`

Minor startup performance updates by pre-compilation

You may get minor startup time decreases by using ngen, and pre-compiling EF views. This may be useful if you need to decrease program startup time by up to few seconds. Likely, runtime performance will not be affected. You will need to ensure the Entity Framework class library and the pre-compiled views stay up-to-date, though.



Multiple result sets

Multiple Active Result Sets (MARS) is a feature that works with SQL Server to allow the execution of multiple batches on a single connection. When MARS is enabled for use with SQL Server, each command object used adds a session to the connection. Normally it is enables if had EF tooling create the connection string. In case, you've manually created the connection string you can enable it easily by inserting to connection string:

```
MultipleActiveResultSets=True;
```


Optimization options outside MVC, EF and LINQ

File system, indices, CDN

Do you need a SQL database?

If we are retrieving articles based on a single key we could also have a well-tested system with good performance - the file system. The file system is also a sophisticated database for reading files. If we are mostly reading long texts and don't need complex queries we could consider simple text files. This will simplify our setup and free up RAM and costs as we don't need a database software or server. Drawbacks are, for example, all writes and updates lock the file and we can not efficiently find files based on the content. If we always read all files anyway or simply read files based on the name the file system could be an option. We may additionally create our own in-memory data structure for better performance.

Use a Content Delivery Network proxy

A good proxy server will cache our files. Combine it with a globally distributed Content Delivery Network and you will have fast delivery for static content. Often, just a DNS update is required to take benefit of the proxies. The biggest provider is CloudFlare.

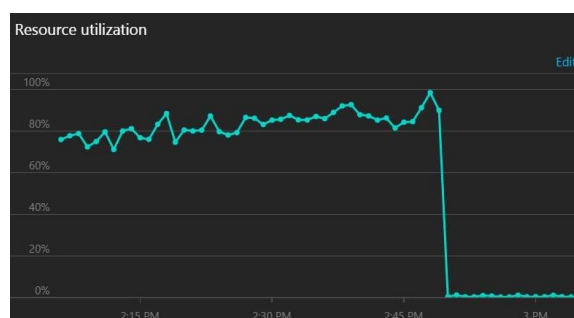
Verify indices are used for all queries

A database has two options for fetching data: use an index, or go through all database rows (perform a full-table scan). The challenges with indices is that only the first data column in an index can be used for queries. This means you will need to verify that there is at least one index that begins with one of the column set as the criteria for a LINQ call (the criteria in the .Where() method). Additionally, you should verify all foreign keys (e.g. ArticleID column) has an index.

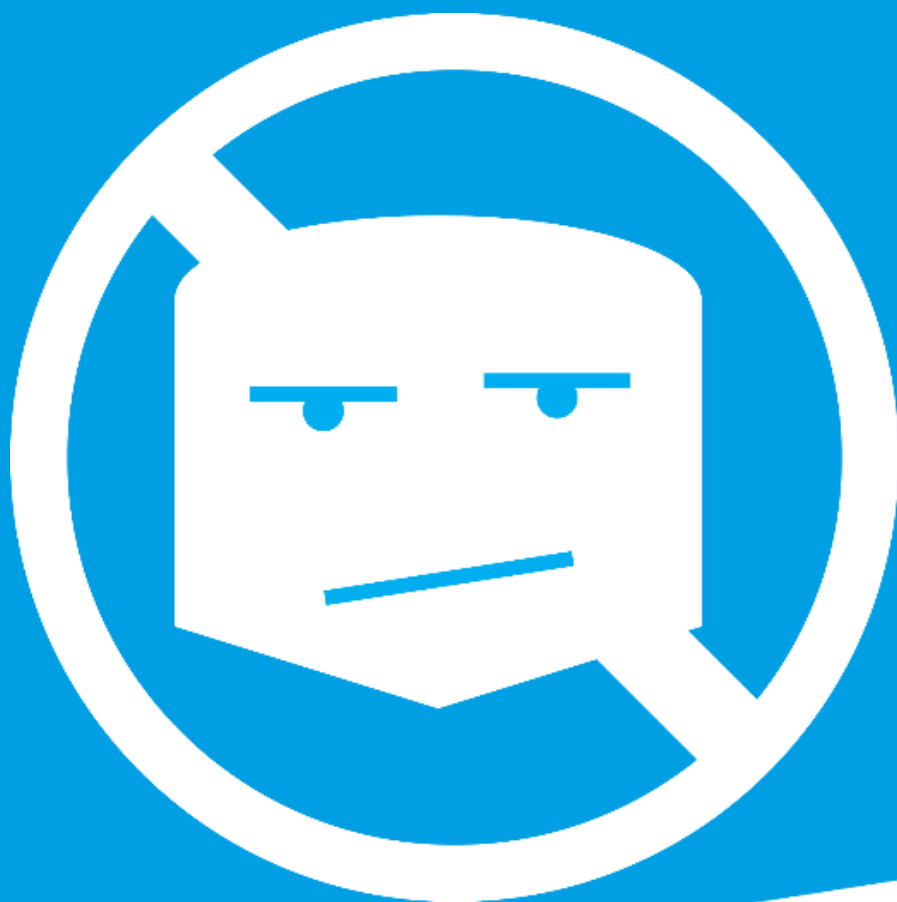
In our example, we should have at least one index beginning with:

<i>ArticleSet.ArticleId</i>	<i>Primary Key</i>
<i>LinkSet.ArticleId</i>	<i>Foreign Key</i>
<i>AttachmentSet.ArticleId</i>	<i>Foreign Key</i>
<i>CatgorySet.ArticleId</i>	<i>Foreign key</i>
<i>ArticleSet.PublishStatus</i>	<i>Often used query criteria</i>

Indices are a powerful way to improve database performance. How to create powerful indices and avoid full-table scans is an important topic which probably has the best return on time spent.



Example of how adding a single index changed resource consumption in a large Azure SQL Database.



STOP

BORING

web sites