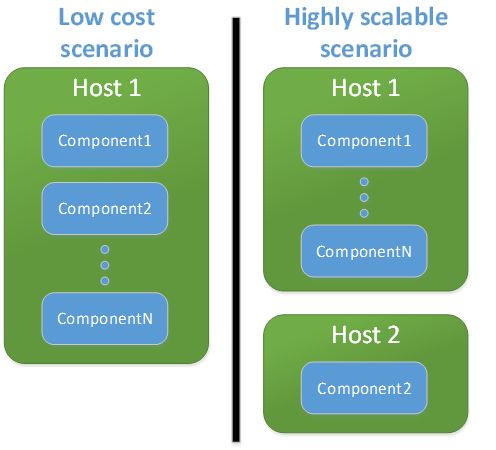
**his is the first one of two posts concerning approach to build cost effective, but prepared for scaling…**

In the era of building systems that aim to provide services at global scale, requirements for scalability and high availability are becoming our bread and butter. What is more, it is absolutely normal that stakeholders want first shippable version of software as soon as possible.

Due to recent hype, microservices architecture comes to our mid as the first answer to mentioned challenges. However, as it is commonly known: “(…) microservices introduce complexity on their own account. This adds a premium to a project’s cost and risk – one that often gets projects into serious trouble” ([Martin Fowler](http://martinfowler.com/bliki/MicroservicePremium.html)). We can see this requirement does not help to deliver system fast. So where is the golden mean? How to build fast, but be prepared to scale and provide high availability without dramatic, expensive changes to system? Let me propose architecture that answers those questions. At least some of them.

**Overview**

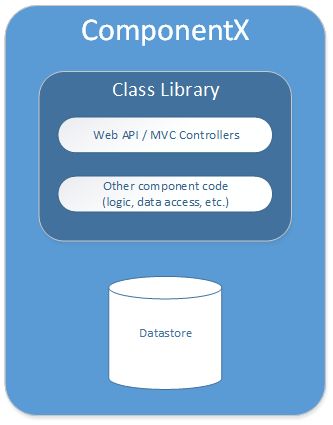
The main idea of this article is to organize system into multiple lightweight, logical, decoupled components, rather than multiple independently hosted services. With this approach we can start with hosting all components in single service and dividing them into multiple ones overtime.

[](https://www.future-processing.com/blog/wp-content/uploads/2015/08/Hosts.jpg)

It looks nice as an idea, however what actually hosts and those “mysterious” components here are.

Hosts are, in short, ASP.NET Web API (or MVC) applications that can be hosted as Azure Web Apps or started on developers’ machine. Their only responsibility is to host and initially configure application, but there should be no application logic, or simply: no controllers.

What is this component then? Let’s zoom in!

[](https://www.future-processing.com/blog/wp-content/uploads/2015/08/ComponentX.jpg)

In the suggested architecture, component is a bunch of Web API (MVC) controllers with all related business logic. All those classes are placed outside of ASP.NET Web API host application, usually in simple class library. Single component should consist of controllers and logic related to single business functionality. Usually, each component has its own place to store data, it can be common or dedicated database.

To put all this together, host project just need reference class libraries containing components code. Since now, controllers are available to process HTTP requests sent to host application. Moving component to separate host is as simple as changing reference between projects.

With this approach we benefit from most of microservice architecture advantages, and initially limit pain that comes with it by keeping development environment easy to run on one machine, limiting DevOps work, and cutting off a lot of distributed system problems. Additionally, we achieve flexibility of production environment costs. Initially limited number of machines do not dry out our accounts, however, when necessary, we are able to distribute components to multiple independent hosts, achieving better scalability.

**Building actual system**

Let’s assume we are creating social system, where users can find and rate nearby shops and markets. For scope of this article we want to:

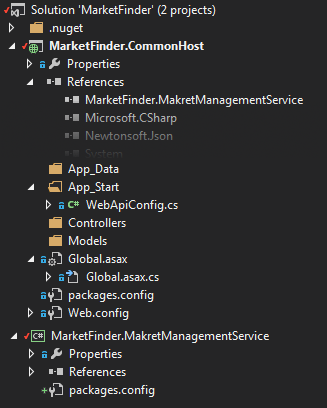
* Create, read, update and delete data about markets and user’s ratings.
* Generate suggestions what another markets user may like too.

And of course our stakeholders want the first version of project quickly, however, at some point after release, they expect system to handle millions of users from entire word.

**Let’s build it**

Source code of a sample project described in this article can be found here:  
[https://github.com/FutureProcessing/Microservices—ASP.NET-Web-API—Azure](https://github.com/FutureProcessing/Microservices---ASP.NET-Web-API---Azure)

1. Project structure  
   First of all, we need basic project structure.
   1. Create empty ASP.NET Web Application with Web API.
   2. Create empty Class Library and install Microsoft.AspNet.WebApi NuGet Package in it.
   3. Add reference from Web API project to Class Library.

After those steps your solution should look like this:  
[](https://www.future-processing.com/blog/wp-content/uploads/2015/08/Project-structure.png)

1. Market management service implementation

As the next step, we want to implement CRUD operations within MakretManagementService. For simplicity of this example I’ve just scaffolded MarkedController and RatingsController using Visual Studio tools.

Source code of this state of system can be found in repository at following revision  
[https://github.com/FutureProcessing/Microservices—ASP.NET-Web-API—Azure/tree/market\_management\_service\_impl](https://github.com/FutureProcessing/Microservices---ASP.NET-Web-API---Azure/tree/market_management_service_impl)

Now, we can test if this solution actually works. To do so we need appropriate tool e.g. Fiddler or PostMan to send HTTP requests to our services. Works on my machine!

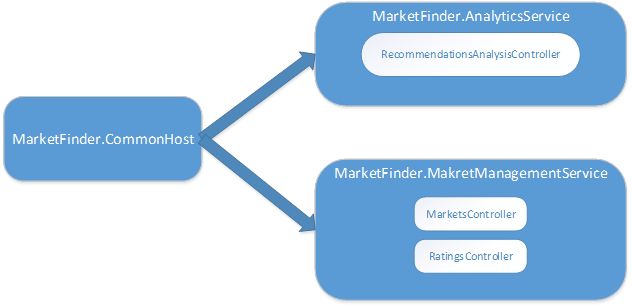
1. Create Analytics service  
   The last feature we need before release is suggestions generation. Such feature is expected to be time consuming operation that we want to run periodically, and store results in database so they can be accessed by client applications later on. What is more, in future we may not want this logic to run at the same machine that MakretManagementService, because this heavy operation might slow system down. For this reason we are going to create it as a separate component.To achieve that we just need to:
   1. Create empty Class Library and install Microsoft.AspNet.WebApi NuGet Package in it.
   2. Add reference from Web API project to Class Library.

Code itself is not important here, so just skip discussion on this.

Source code of this state of system can be found in repository at following revision  
[https://github.com/FutureProcessing/Microservices—ASP.NET-Web-API—Azure/tree/analytics\_service\_impl](https://github.com/FutureProcessing/Microservices---ASP.NET-Web-API---Azure/tree/analytics_service_impl)

Now we can send POST request to http://<<base\_address>/RecommendationsAnalysis and have our suggestions created.

At this stage system looks as follows:

[](https://www.future-processing.com/blog/wp-content/uploads/2015/08/MarketFinder.jpg)

It is important to notice at this point that there is absolutely no coupling between two services on the code level, however as you might noticed it doesn’t come for free. There is some duplication in data access code and data model classes. Although, at this moment, it can be generalized and placed in common class library, it may not be the best solution when we think about long term perspective and further independent development of those services. There are no good and bad solutions, there are only pros and cons in particular context.

Sounds like we are ready to ship our Minimum Viable Product to customers!

**Time for scaling**

Until now we have created application optimized for low development and hosting costs. It runs as a single application and store data in single database. Let’s now react to scenario where we already have a lot of data about markets, single database is getting large, calculating user recommendation takes long enough to significantly slow website down for long periods.

Some improvements that should help in a given problem are:

1. Move recommendations data to different database.
2. Run Analysis service in separate host application, so we can run it in independent container.

The first change is nearly trivial. We need to create another database with Recommendations table. Next, because our system is already running, we need to migrate data from first database. After that there is a single connection string to change in web.config.

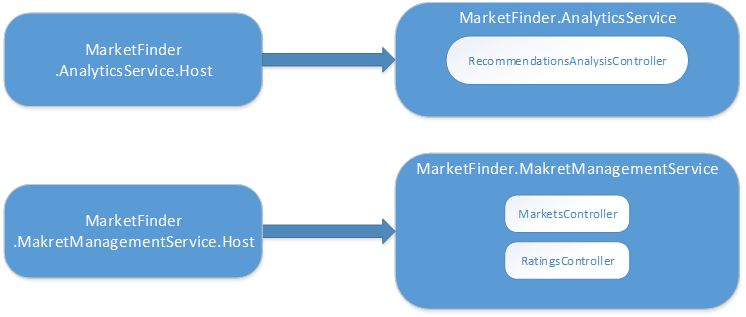
Such change will be that easy only if we took care not to introduce coupling between data behind different components. It is not simple and, again, usually requires some data duplication and synchronization, however that is the cost for possibility of keeping parts of system independent. If you are familiar with Domain Driven Design concept of bounded contexts, you have probably noticed that what I’m suggesting here is similar to concept of aggregates and persisting them as an internally cohesive, decoupled from each other set of entities.

Changes to database creation scripts and project configuration can be analyzed here:  
[https://github.com/FutureProcessing/Microservices—ASP.NET-Web-API—Azure/commit/scale\_db](https://github.com/FutureProcessing/Microservices---ASP.NET-Web-API---Azure/commit/scale_db)

The second change isn’t complicated too, all you need to do is to create new host projects (same way as previously) and add references to appropriate class libraries.

Changes to C# code can be analyzed here:  
[https://github.com/FutureProcessing/Microservices—ASP.NET-Web-API—Azure/commit/scale\_csharp](https://github.com/FutureProcessing/Microservices---ASP.NET-Web-API---Azure/commit/scale_csharp)

Now you can enjoy system running its components in two microservices. From broader perspective it works this way:

[](https://www.future-processing.com/blog/wp-content/uploads/2015/08/MarketFinder.CommonHost.jpg)

You might have noticed that I’ve left in solution MarketFinder.CommonHost project that is hosting both applications. This project is useful during development, as it is usually faster to start single web site than multiple ones, especially when we have a lot of them.

**Summary**

Most of community advice against starting system development with microservice architecture, because it comes with huge costs and complexity overhead. In return, it is recommended to build monolithic system and extract microservices overtime when required. Advised approach sounds great, but to divide system into services in future, we need to make sure we do not create highly coupled ball of mud. Architectural approach presented in this article may help us by creating some logical boundaries of system components. Having them prevent us, and our less experiences teammates, from introducing hidden coupling.

However, nothing in software development is the silver bullet. Suggested approach introduces less complexity and cost overhead than fully blown microservices architecture, however it still does. The biggest problem when cutting system on independent components, is that it can never be done without actually cutting of some dependencies that exist. If we find boundaries where those dependencies are minimal, we will benefit from low coupling in future, however when we set those boundaries wrongly, we will heavily suffer from development complexity and probably poor system performance.

## Key takeaways

* Repositories make sense when you are doing more than basic CRUD.
* To facilitate testing and improve reliability, repositories should be treated as reusable libraries.
* Moving security and auditing into the repository can reduce bugs and simplify your application.
* Your ORM choice doesn’t limit what your repository can do, just how much work it will take to do it.

In our previous article, [Implementation Strategies for the Repository Pattern with Entity Framework, Dapper, and Chain](https://www.infoq.com/articles/repository-implementation-strategies), we looked at the basic patterns needed to implement a repository. In many cases these patterns were such a thin layer around the underlying data access technology they were essentially unnecessary. However, once you have a repository in place, many new opportunities become available.

When designing a repository, you should be thinking in terms of “what must happen”. For example, let us say you have a rule that whenever a record is updated, its “LastModifiedBy” column must be set to the current user. Rather than trying to remember to update the LastModifiedBy in application code before every save, you can bake that functionality right into the repository.

Related Vendor Content

### [MongoDB: Production Ready, Cloud Hosted - Try FREE for 30 Days](https://www.infoq.com/infoq/url.action?i=16189&t=f)

### [How to Overcome the Microservices Sprawl](https://www.infoq.com/vendorcontent/show.action?vcr=4439&primaryTopicId=4&vcrPlace=EMBEDDED&pageType=ARTICLE_PAGE&vcrReferrer=https%3A%2F%2Fwww.infoq.com%2Farticles%2Frepository-advanced&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click&utm_content=embedded)

### [Microservices War Stories](https://www.infoq.com/vendorcontent/show.action?vcr=4438&primaryTopicId=4&vcrPlace=EMBEDDED&pageType=ARTICLE_PAGE&vcrReferrer=https%3A%2F%2Fwww.infoq.com%2Farticles%2Frepository-advanced&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click&utm_content=embedded)

### [Microservice Databases: Migrating from Relational Monolith to Distributed Data (By O'Reilly)](https://www.infoq.com/vendorcontent/show.action?vcr=4348&primaryTopicId=4&vcrPlace=EMBEDDED&pageType=ARTICLE_PAGE&vcrReferrer=https%3A%2F%2Fwww.infoq.com%2Farticles%2Frepository-advanced&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click&utm_content=embedded)

### [Introduction to Graph Databases](https://www.infoq.com/vendorcontent/show.action?vcr=4464&primaryTopicId=4&vcrPlace=EMBEDDED&pageType=ARTICLE_PAGE&vcrReferrer=https%3A%2F%2Fwww.infoq.com%2Farticles%2Frepository-advanced&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click&utm_content=embedded)

Related Sponsor

[https://cdn.infoq.com/statics_s2_20170613-0338/resource/sponsorship/featuredcategory/16690/compose-logo-rsb-052017.png](https://www.infoq.com/infoq/url.action?i=16191&t=f)

Deploy and manage industry leading databases in minutes. Our platform comes with high availability, auto-scaling, auto-backup and more. [**Free to try.**](https://www.infoq.com/infoq/url.action?i=16190&t=f)

By treating your data access layer as a standalone library that manages all of the “must happen” details, you can dramatically reduce implementation errors. At the same time, you can simplify the code that is built on top of the repository, as it no longer needs to be concerned about bookkeeping tasks.

Note: where appropriate, this article will include code samples for [Entity Framework](https://msdn.microsoft.com/en-us/data/ef.aspx), [Dapper](https://github.com/StackExchange/dapper-dot-net), and/or [Tortuga Chain](https://github.com/docevaad/Chain). However, you will find most repository features can be implemented in an ORM-agnostic fashion.

## Audit Columns

Most applications eventually find the need to track who made changes to the database and when. For simple databases this takes the form of audit columns. The names vary, but they usually fall into these four roles:

* Created by User Key
* Created Date/Time
* Last Modified by User Key
* Last Modified Date/Time

Depending on the security requirements of the application, additional audit columns may be considered such as:

* Deleted by User Key
* Deleted Date/Time
* [Created | Last Modified | Deleted] by Application Key
* [Created | Last Modified | Deleted] by IP Address

The date columns are easy enough to handle behind the scenes, but for the user keys you need to do something a bit more interesting. What you need is a “context aware repository”.

Normally repositories are context free, meaning they have no information other than what’s absolutely necessary to connect to the database. When correctly designed, the repository can be entirely stateless, allowing you to share one instance across the whole application.

Context aware repositories are a bit more complex. They cannot be constructed until you know the context, which at the very least includes the currently active user’s id or key. For some applications, this is enough. For others, you may need to pass in an entire user object and/or an object representing the running application.

### Chain

Chain has built in support through a concept known as audit rules. Audit rules allow you to specify overrides based on a column name. Out of the box, it includes date-based rules and rules that copy a property from a user object into a column. Here is an example,

dataSource = dataSource.WithRules(

new UserDataRule("CreatedByKey", "UserKey", OperationType.Insert),

new UserDataRule("UpdatedByKey", "UserKey", OperationType.InsertOrUpdate),

new DateTimeRule("CreatedDate", DateTimeKind.Local, OperationType.Insert),

new DateTimeRule("UpdatedDate", DateTimeKind.Local, OperationType.InsertOrUpdate)

);

As mentioned above, you are going to need a context aware repository for this to work. In the constructor below you can see how the context is passed to an immutable data source, creating a new data source with the necessary information.

public EmployeeRepository(DataSource dataSource, User user)

{

m\_DataSource = dataSource.WithUser(user);

}

Thus setup, you can leverage your DI framework of choice to automatically create and populate the repository on a per-request basis.

### Entity Framework

To globally apply audit columns in Entity Framework, you need to leverage the ObjectStateManager and create a specialized interface. The interface, or base class if you prefer, will look something like this:

public interface IAuditableEntity

{

DateTime CreatedDate {get; set;}

DateTime UpdatedDate {get; set;}

DateTime CreatedDate {get; set;}

DateTime CreatedDate {get; set;}

}

The interface (or base class) is then applied to every entity for which the database has matching audit columns.

Next you need to override the Save method of your DataContext class as follows.

public override int SaveChanges()

{

// Get added entries

IEnumerable<ObjectStateEntry> addedEntryCollection = Context

.ObjectContext

.ObjectStateManager

.GetObjectStateEntries(EntityState.Added)

.Where(m => m != null && m.Entity != null);

// Get modified entries

IEnumerable<ObjectStateEntry> modifiedEntryCollection = Context

.ObjectContext

.ObjectStateManager

.GetObjectStateEntries(EntityState.Modified)

.Where(m => m != null && m.Entity != null);

// Set audit fields of added entries

foreach (ObjectStateEntry entry in addedEntryCollection)

{

var addedEntity = entry.Entity as IAuditableEntity;

if (addedEntity != null)

{

addedEntity.CreatedDate = DateTime.Now;

addedEntity.CreatedByKey = m\_User.UserKey;

addedEntity.UpdatedDate = DateTime.Now;

addedEntity.UpdatedByKey = m\_User.UserKey;

}

}

// Set audit fields of modified entries

foreach (ObjectStateEntry entry in modifiedEntryCollection)

{

var modifiedEntity = entry.Entity as IAuditableEntity;

if (modifiedEntity != null)

{

modifiedEntity.UpdatedDate = DateTime.Now;

modifiedEntity.UpdatedByKey = m\_User.UserKey;

}

}

return SaveChanges();

}

If you are going to be working a lot with EF, it really pays to become intimately familiar with the ObjectStateManager and its capabilities. This is where most of the useful metadata about transactions in progress can be found.

Finally, you need to modify the constructor of your data context (and possibly repository) to accept a user object.

While this looks like a lot of code, it only needs to be done once per EF data context. And as with the previous example, the actual creation of the data context and repository can be performed by your DI framework.

## History Table

Tracking changes to records is often required due to local laws and regulations. Other times it is desirable simply to make diagnostics easier.

Our general recommendation is to simply allow the database to do this. Some databases have this capability built-in, which is often referred to as a [temporal table](https://en.m.wikipedia.org/wiki/Temporal_database). Others can emulate it using triggers. In either case, the application is unaware of the additional logging, which makes the technique far less error prone.

If for some reason you cannot use a temporal table or trigger, then the repository needs to explicitly write to the history table.

Regardless of where you put the code that maintains the history table, there are two basic conventions that you can follow. Consistency is really important here, as it would be quite confusing to have one convention for some tables and the other convention for the rest.

**Copy before Write**: In this convention you copy the old record from live table to the history table prior to performing the update or delete operation. This means the history table never contains the current record. As a result, you’ll need to join the live and history tables together to see a complete history of changes.

**Write before Copy**: Alternately you can update the live table first, then copy that row to the history table. This has the advantage of having a complete picture in the history table, eliminating the aforementioned join. The downside is it takes more space due to the duplicated data.

With either convention, you’ll want to use soft deletes in order to track who is actually deleting the row. If hard deletes are desired, then they should only be performed following a soft delete.

## Soft Deletes

One of the advantages of using a repository is you can switch from hard deletes to soft deletes without the rest of the application realizing it. A soft delete removes the record as far as the application is concerned, but allows it to remain in the database for auditing purposes. Optionally, an application can even undelete records.

To avoid data loss, applications should not be granted DELETE privileges on tables that support soft deletes. If the application accidentally tries to perform a hard delete, the permission check will raise an error instead of silently deleting the row.

### Chain

Chain offers implicit soft delete support via the audit rules infrastructure. When configuring a soft delete rule, it is customary to also configure the matching audit columns:

var dataSource = dataSource.WithRules(

new SoftDeleteRule("DeletedFlag", true, OperationTypes.SelectOrDelete),

new UserDataRule("DeletedByKey", "EmployeeKey", OperationTypes.Delete),

new DateTimeRule("DeletedDate", DateTimeKind.Local, OperationTypes.Delete)

);

Whenever a table is found with a soft delete column (DeletedFlag in this example), two things happen automatically:

* All queries implicitly add “AND DeletedFlag = 0” to the WHERE clause.
* All calls to DataSource.Delete become update statements to set the deleted flag.

### Entity Framework

In Entity Framework, you can include an additional where clause on every query that reads from a table supporting soft deletes. You’ll also need to manually convert any delete operation into an update, which can be tricky when working with object graphs.

Another option takes quite a bit of work, but may be less error prone. It starts by explicitly listing every table that supports soft deletes in the DataContext.OnModelCreating override.

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<Employee>().Map(m => m.Requires("IsDeleted").HasValue(false));

}

You then need to override the Save method to ensure deletes become updates. [Colin on Stackoverflow](http://stackoverflow.com/a/18985828/5274) offers this pattern.

public override int SaveChanges()

{

foreach (var entry in ChangeTracker.Entries()

.Where(p => p.State == EntityState.Deleted

&& p.Entity is ModelBase))

SoftDelete(entry);

return base.SaveChanges();

}

private void SoftDelete(DbEntityEntry entry)

{

var e = (ModelBase)entry.Entity;

string tableName = GetTableName(e.GetType());

Database.ExecuteSqlCommand(

String.Format("UPDATE {0} SET IsDeleted = 1 WHERE ID = @id", tableName)

, new SqlParameter("id", e.ID));

//Marking it Detached prevents the hard delete

entry.State = EntityState.Detached;

}

You’ll want to read the rest of Colin’s answer, as there are a lot of edge cases to be addressed.

## Access Logging

While audit columns, history tables, and soft deletes cover all of the write scenarios, there are times when you also need to log reads. An example of this is the US health care industry. Any doctor or nurse needs the ability to access any patient’s medical records in the event of an emergency. But in the normal course of business, they are only legally allowed to do so when they are actively treating that patient.

Since the records cannot be fully locked down, the next best thing is to track who is reading each record. This can be easily accomplished at the repository level by logging each query involving sensitive data. This is most easily done manually at the top of the relevant repository methods.

## Performance Logging

When user experience is a feature, it is important to know how much time is being spent on a per-query basis. Merely tracking per-page performance isn’t enough, as one page may involve multiple queries. This is especially true of Entity Framework, as lazy-loading can hide database calls.

### Explicit Logging in the Repository

Though it is tedious and easy to miss a query, one can simply wrap every query in a disposable timer. The pattern is as follows:

public class OperationTimer : IDisposable

{

readonly object m\_Context;

readonly Stopwatch m\_Timer;

public OperationTimer(object context)

{

m\_Context = context;

m\_Timer = Stopwatch.StartNew();

}

public void Dispose()

{

//Write to log here using timer and context

}

}

And the usage:

using(new OperationTimer("Load employees"))

{

//execute query here

}

### Chain

Chain exposes a set of events at the data source level. The one needed in this case is DataSource.ExecutionFinished. Here is an example:

static void DefaultDispatcher\_ExecutionFinished(object sender, ExecutionEventArgs e)

{

Debug.WriteLine($"Execution finished: {e.ExecutionDetails.OperationName}. Duration: {e.Duration.Value.TotalSeconds.ToString("N3")} sec. Rows affected: {(e.RowsAffected != null ? e.RowsAffected.Value.ToString("N0") : "<NULL>")}.");

}

You can also attach a handler to DataSource.GlobalExecutionFinished, which listens to events from all data sources.

### Entity Framework

The built-in logging capabilities of Entity Framework don’t make it possible to time individual queries. You can work around this limitation using a custom [IDbCommandInterceptor](https://msdn.microsoft.com/en-us/data/dn469464.aspx).

public class EFLoggerForTesting : IDbCommandInterceptor

{

static readonly ConcurrentDictionary<DbCommand, DateTime> m\_StartTime = new ConcurrentDictionary<DbCommand, DateTime>();

public void ReaderExecuted(DbCommand command, DbCommandInterceptionContext<DbDataReader> interceptionContext)

{

Log(command, interceptionContext);

}

public void NonQueryExecuted(DbCommand command, DbCommandInterceptionContext<int> interceptionContext)

{

Log(command, interceptionContext);

}

public void ScalarExecuted(DbCommand command, DbCommandInterceptionContext<object> interceptionContext)

{

Log(command, interceptionContext);

}

private static void Log<T>(DbCommand command, DbCommandInterceptionContext<T> interceptionContext)

{

DateTime startTime;

TimeSpan duration;

m\_StartTime.TryRemove(command, out startTime);

if (startTime != default(DateTime))

{

duration = DateTime.Now - startTime;

}

else

duration = TimeSpan.Zero;

string message;

var parameters = new StringBuilder();

foreach (DbParameter param in command.Parameters)

{

parameters.AppendLine(param.ParameterName + " " + param.DbType + " = " + param.Value);

}

if (interceptionContext.Exception == null)

{

message = string.Format("Database call took {0} sec. RequestId {1} \r\nCommand:\r\n{2}", duration.TotalSeconds.ToString("N3"), requestId, parameters.ToString() + command.CommandText);

}

else

{

message = string.Format("EF Database call failed after {0} sec. RequestId {1} \r\nCommand:\r\n{2}\r\nError:{3} ", duration.TotalSeconds.ToString("N3"), requestId, parameters.ToString() + command.CommandText, interceptionContext.Exception);

}

Debug.WriteLine(message);

}

public void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<int> interceptionContext)

{

OnStart(command);

}

public void ReaderExecuting(DbCommand command, DbCommandInterceptionContext<DbDataReader> interceptionContext)

{

OnStart(command);

}

public void ScalarExecuting(DbCommand command, DbCommandInterceptionContext<object> interceptionContext)

{

OnStart(command);

}

private static void OnStart(DbCommand command)

{

m\_StartTime.TryAdd(command, DateTime.Now);

}

}

This doesn’t offer a way to capture contextual data, but you can work around that limitation by shoving the context in [ThreadLocal](https://msdn.microsoft.com/en-us/library/dd642243%28v=vs.110%29.aspx) or [AsyncLocal](https://msdn.microsoft.com/en-us/library/dn906268%28v=vs.110%29.aspx) as appropriate.

## Permission Checks – Table Level

While permission checks can be done at the application level, it is often advantageous to also enforce them in the repository. This eliminates the possibility the permission check is forgotten on newly created screens/pages.

### Repository Enforcement

The simplest way to implement this is a role check at the beginning of each relevant function. For example,

public int Insert(Employee employee)

{

if (!m\_User.IsAdmin)

throw new SecurityException("Only admins may add employees");

### Database Enforced

A more sophisticated option would involve creating multiple connection strings. When the repository is created, a connection string is selected based on the user’s role. In this case, the connection string for non-admin users wouldn’t have INSERT privileges on the employee table.

Due to the complexity and maintenance headaches involved, this approach is not recommended except under very high security environments where multiple layers of defense are desired. Even then, it requires extensive automated testing to ensure every connection string has all the permissions it needs.

## Permission Checks – Column Level

Permission checks may also be needed at the column level. For example, you may want to prevent users from giving themselves admin privileges. Or you may want to block non-managers from seeing employee salaries.

### Chain

Chain leverages its audit rules capabilities to implement column level permission checks. An anonymous function is passed to the RestrictColumn constructor along with the column name and list of restricted operations. (A table name can be optionally specified as well.)

var IsAdminCheck = user => ((User)user).IsAdmin;

dataSource = dataSource.WithRules(

new RestrictColumn("Users", "IsAdmin", OperationTypes.Insert|OperationTypes.Update, IsAdminCheck));

To prevent reading of a restricted column, pass in the OperationTypes.Select flag.

### Dapper

The easiest way to do this in Dapper is to simply have multiple SQL statements. If the user lacks a specific privilege, you simply select the SQL statement that omits those columns.

### Entity Framework

For queries, there are a couple of options available.

1. Manually create difference projections (i.e. Select clauses) depending on the user’s role
2. Perform the query normally. Then if the permission check fails, loop through the result set, setting the restricted properties to null/0.

For inserts, simply blank out the restricted properties as above.

Updates are trickier. When restricting writes to individual columns, you cannot attach entities. Rather, you need to re-fetch the original record, copy across the permitted values, and then save that object instead of the one passed in by the application code. (Essentially our “novice” pattern from the previous article.)

## Mapping One Model to Multiple Tables

An important concept in data architecture is the idea that you don’t need a one-to-one mapping between tables and classes. In order to make the database work more efficiently or to address a particular business rule, you will often find it advantageous to map one class to multiple tables.

Say, for example, you were tracking baseball teams. You may have these tables:

|  |  |
| --- | --- |
| **Table** | **Primary Key** |
| **Team** | **TeamKey** |
| **TeamSeasonMap** | **TeamKey+SeasonKey** |

If your application only understands the concept of a team in the context of a season, you may have one Team object that covers both tables.

### Chain

In Chain, there isn’t a strong relationship between classes and tables. This means for updates you would write code such as this:

dataSource.Update("Team", myTeam).Execute();

dataSource.Update("TeamSeasonMap", myTeam).Execute();

At runtime it will determine which properties are applicable to which tables and generate the SQL accordingly.

Under this model, you would fetch the Team object from a view that joined both tables. (Chain doesn’t support joins directly and assumes they will always occur via views.)

### Entity Framework

Entity Framework expects that multiple tables mapping to a single entity all share exactly the same primary key. This means that it will not support that scenario.

* For reads, you can perform the join and projection using EF’s normal LINQ syntax.
* For updates, you will need to copy the model into a separate entity for each table.

## Caching

Generally speaking, caching is a repository concern. Since the repository knows when data is being altered, it is the best equipped to handle cache invalidation.

### Chain

Caching is supported by Chain, but it needs to be applies on a query by query basis using appenders. Appenders are attached to operations before they are executed. In this case there are four appenders we care about:

* .Cache(...)
* .CacheAllItems(...)
* .InvalidateCache(...)
* .ReadOrCache(...)

They are best explained by means of an example repository. Here you can see the interplay between caching individual records and caching collections using `CacheAllItems`.

public class EmployeeCachingRepository

{

private const string TableName = "HR.Employee";

private const string AllCacheKey = "HR.Employee ALL";

public IClass1DataSource Source { get; private set; }

public CachePolicy Policy { get; private set; }

public EmployeeCachingRepository(IClass1DataSource source, CachePolicy policy = null)

{

Source = source;

Policy = policy;

}

protected string CacheKey(int id)

{

return $"HR.Employee EmployeeKey={id}";

}

protected string CacheKey(Employee entity)

{

return CacheKey(entity.EmployeeKey.Value);

}

public Employee Get(int id)

{

return Source.GetByKey(TableName, id).ToObject<Employee>().ReadOrCache(CacheKey(id), policy: Policy).Execute();

}

public IList<Employee> GetAll()

{

return Source.From(TableName).ToCollection<Employee>().CacheAllItems((Employee x) => CacheKey(x), policy: Policy).ReadOrCache(AllCacheKey, policy: Policy).Execute();

}

public Employee Insert(Employee entity)

{

return Source.Insert(TableName, entity).ToObject<Employee>().InvalidateCache(AllCacheKey).Cache((Employee x) => CacheKey(x), policy: Policy).Execute();

}

public Employee Update(Employee entity)

{

return Source.Update(TableName, entity).ToObject<Employee>().Cache(CacheKey(entity)).InvalidateCache(AllCacheKey).Execute();

}

public void Delete(int id)

{

Source.DeleteByKey(TableName, id).InvalidateCache(CacheKey(id)).InvalidateCache(AllCacheKey).Execute();

}

}

As you can see, Chain gives you a lot of control over your invalidation logic at the cost of having to carefully specify everything.

### Entity Framework

Entity Framework has two levels of caching. The first level is limited to the data context and is primarily concerned with ensuring the object graph doesn’t have duplication entities that represent the same physical database record. Since this cache is destroyed along with the data context, it is not relevant to most caching scenarios.

In EF terminology, what we’re looking for is known as a “second level cache”. While available in EF 5, version 6 of Entity Framework didn’t ship with any sort of caching out of the box. For this, you’ll need to turn to the third party libraries such as [EntityFramework.Cache](https://efcache.codeplex.com/) or [EFSecondLevelCache](https://github.com/VahidN/EFSecondLevelCache). As you can see from these example libraries, there is no standard pattern for adding second level caching to EF.

**Create database, table & stored procedure for CRUD operation**   
  
SQL Script  
  
**Script 1**

**Script to create database**

1. createdatabase DataManagement

**Script 2**

**Script to create table**

1. CREATETABLE[dbo].[Users](
2. [UserId][int] IDENTITY(1, 1) NOT NULL, [UserName][varchar](50) NULL, [UserMobile][varchar](50) NULL, [UserEmail][varchar](50) NULL, [FaceBookUrl][varchar](50) NULL, [LinkedInUrl][varchar](50) NULL, [TwitterUrl][varchar](50) NULL, [PersonalWebUrl][varchar](50) NULL, [IsDeleted][bit] NULL) ON[PRIMARY]
3. GO
4. ALTERTABLE[dbo].[Users] ADDCONSTRAINT[DF\_User\_IsDeleted] DEFAULT((0)) FOR[IsDeleted]
5. GO

**Script 3**

**Script to add user**

1. createPROCEDURE[dbo].[AddUser]
2. @UserName varchar(50),
3. @UserMobile varchar(50),
4. @UserEmail varchar(50),
5. @FaceBookUrl varchar(50),
6. @LinkedInUrl varchar(50),
7. @TwitterUrl varchar(50),
8. @PersonalWebUrl varchar(50)
9. AS
10. BEGIN
11. SETNOCOUNTON;
12. insertinto Users(UserName, UserMobile, UserEmail, FaceBookUrl, LinkedInUrl, TwitterUrl, PersonalWebUrl)
13. values(@UserName, @UserMobile, @UserEmail, @FaceBookUrl, @LinkedInUrl, @TwitterUrl, @PersonalWebUrl)
14. END
15. GO

**Script 4**

**Script to delete user**

1. CREATEPROCEDURE[dbo].[DeleteUser]
2. @UserId int
3. AS
4. BEGIN
5. SETNOCOUNTON;
6. update Users set IsDeleted = 1 where UserId = @UserId
7. END
8. Script 5 Script to get all users
9. CREATEPROCEDURE[dbo].[GetAllUsers]
10. AS
11. BEGIN
12. SETNOCOUNTON;
13. select \* from Users
14. END

**Script 6**

**Script to get user by Id**

1. CREATEPROCEDURE[dbo].[GetUserById]
2. @UserId int
3. AS
4. BEGIN
5. SETNOCOUNTON;
6. select \* from Users where UserId = @UserId;
7. END
8. GO

**Script 7**

**Script to update user**

1. CREATEPROCEDURE[dbo].[UpdateUser]
2. @UserId int,
3. @UserName varchar(50),
4. @UserMobile varchar(50),
5. @UserEmail varchar(50),
6. @FaceBookUrl varchar(50),
7. @LinkedInUrl varchar(50),
8. @TwitterUrl varchar(50),
9. @PersonalWebUrl varchar(50)
10. AS
11. BEGIN
12. SETNOCOUNTON;
13. update Users set
14. UserName = @UserName,
15. UserMobile = @UserMobile,
16. UserEmail = @UserEmail,
17. FaceBookUrl = @FaceBookUrl,
18. LinkedInUrl = @LinkedInUrl,
19. TwitterUrl = @TwitterUrl,
20. PersonalWebUrl = @PersonalWebUrl
21. where UserId = @UserId
22. END
23. GO

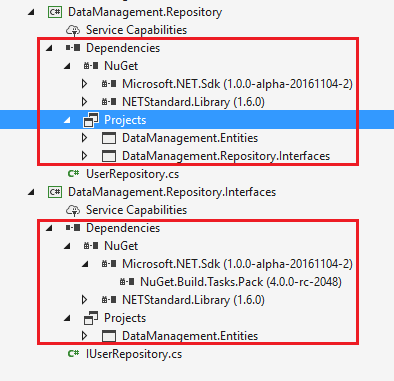
**Step 9**

**Add an Entity for User**

1. Create a class“ User.cs” inside“ DataManagement.Entities” project.
2. Complete Code
3. namespace DataManagement.Entities {
4. publicclassUser {
5. publicint UserId {
6. get;
7. set;
8. }
9. publicstring UserName {
10. get;
11. set;
12. }
13. publicstring UserMobile {
14. get;
15. set;
16. }
17. publicstring UserEmail {
18. get;
19. set;
20. }
21. publicstring FaceBookUrl {
22. get;
23. set;
24. }
25. publicstring LinkedInUrl {
26. get;
27. set;
28. }
29. publicstring TwitterUrl {
30. get;
31. set;
32. }
33. publicstring PersonalWebUrl {
34. get;
35. set;
36. }
37. publicbool IsDeleted {
38. get;
39. set;
40. }
41. }
42. }

**Step 10**

**Write Repository Layer Code for User**

1. Add reference of project “DataManagement.Entities”inside “DataManagement.Repository.Interfaces”.  
     
   For ASP.NET Core you will find it a little bit different as compared to earlier versions while checking for added references. Following is a snapshot for the same.  
     
   
2. Create an Interface “IUserRepository.cs” inside the solution “DataManagement.Repository.Interfaces”.
   1. IUserRepository.cs Complete Code
   2. using DataManagement.Entities;
   3. using System.Collections.Generic;
   4. namespace DataManagement.Repository.Interfaces {
   5. publicinterfaceIUserRepository {
   6. bool AddUser(User user);
   7. bool UpdateUser(User user);
   8. bool DeleteUser(int userId);
   9. IList < User > GetAllUser();
   10. User GetUserById(int userId);
   11. }
   12. }
3. Add reference of projects  
     
   “DataManagement.Entities”&“DataManagement.Repository.Interfaces” inside “DataManagement.Repository”.