Couchbase with Windows and .NET

In this blog post, I'm going to show the very basics of interacting with Couchbase for .NET developers on Windows. I'll start with the basics, and build towards a "vertical" slice of a complete ASP.NET MVC app. If you are interested in diving deeper, please check out my blog posts on Couchbase[1] and the Couchbase Developer Portal[2].

1. Installing Couchbase Server

Let's start with the easiest part: downloading and installing Couchbase Server. I'm going to install it on Windows 10, but you can install it pretty much everywhere, including Mac OSX and Linux. Go to the Downloads page on the Couchbase website[3]. You can now choose from the Enterprise Edition or the Community Edition. I'm going to proceed with the Enterprise Edition for this blog post.

Run the exe file that you just downloaded. You'll be taken through a simple 4-step wizard.

At this point, Couchbase should be running as a Windows Service (you can verify by opening services.msc[4]).

You should also be automatically taken to the Couchbase Console, which you access via web browser. Here you can customize your Couchbase setup. You can always return to this console via **http://localhost:8091**

Click "Setup" to start the configuration wizard. I'm going to stick with the default settings (mostly) for now.

1. In step 1, you need to specify how much RAM to give to Couchbase. You may want to dial down some of the default RAM Quotas (you can always change them later). Since I'm only using the one node for development, I've enabled all services on this node (Index, Query, and Full Text). I would recommend checking out some of the free Couchbase training[5] available to dive deeper into tweaking these settings.

[1] http://blog.couchbase.com/
[2] http://developer.couchbase.com
[3] http://www.couchbase.com/nosql-databases/downloads
[4] https://technet.microsoft.com/en-us/library/cc755249.aspx
[5] http://learn.couchbase.com/

1. In step 2, you can choose to install some sample data. For this blog post, you don't need to do this. However, the "travel-sample" is useful for trying out some of the N1QL functionality (more on that later).
2. In step 3, you create a "default" bucket, which you'll need to follow along with this blog post.
3. In step 4, you can elect to receive notifications from Couchbase and register the product.
4. Finally, in step 5, you need to enter a username and password to create an administrator account.

Now you are ready to start using Couchbase. On the "overview" page, you can see how much RAM is available to Couchbase, and how much is actually being used. If we wanted to scale out Couchbase by adding additional servers, then we would see them listed in the Servers section. If you click on the "Data Buckets" tab, you'll see that there's at least the default bucket you created.

Feel free to play around with the Couchbase Console and check out the Couchbase Console documentation[6].

2. Some NoSQL Lingo

If you're like me, you're itching to get into some code and see what you can do. But before we go there, I want to go over some of the lingo with Couchbase. It's not a difficult tool to use, but it is different from the RDBMS systems like SQL Server that you're probably used to.

Node

A node is an invidual machine that is running Couchbase Server.

Cluster

Nodes can be joined together into a cluster[7] that act in concert to provide scaling and improved availability. Scaling: more nodes means more RAM and disk space. Availability: if one node goes down, the cluster will continue to function. As a developer, you don't generally have to worry about node management and clustering in your code: it's all handled by Couchbase and the Couchbase .NET SDK.

Bucket

[6] http://developer.couchbase.com/documentation/server/4.5/admin/ui-intro.html
[7] http://developer.couchbase.com/documentation/server/current/clustersetup/manage-cluster-intro.html

A bucket[8] is a place to store documents. Each document has a key. Within a bucket, each key must be unique. Typically, a single bucket corresponds to a single application. The documents within a bucket do not have to be similar at all. You could store a document that contains information about a user, and a document with information about a building.

Document

In a very basic sense, a Couchbase bucket is just a giant `Dictionary<string,string>`. Each entry in a bucket consists of a key and a document. Documents consist of JSON. Because Couchbase uses JSON documents, it's called a "document database".

N1QL

Couchbase recognizes that relational databases have been a huge part of many developer's careers. Many developers feel comfortable writing SQL. With Couchbase Server, you can write queries in a language called N1QL[9] (N1QL stands for "Non-first Normal Form Query Language, and is pronounced "nickel"). N1QL is a superset of SQL. This means that, basically, if you know SQL, then you know N1QL. An example:

```
SELECT name AS bookName, author AS bookAuthor
FROM `books-bucket`
WHERE YEAR(published) >= 1998
```

That will return something like:

```
{
 "results": [
  { "bookName" : "The Little Book of Calm", "bookAuthor" : "Manny Bianco" },
  { "bookName" : "AOP in .NET", "bookAuthor" : "Matthew D. Groves" }
 ]
}
```

As I'll show you in later blog posts, the Linq2Couchbase library leverages N1QL to give you a Linq provider that will feel very similar to Entity Framework, NHibernate.Linq, or other Linq providers that you're used to.

[8] http://developer.couchbase.com/documentation/server/4.5/clustersetup/bucket-setup.html
[9] http://developer.couchbase.com/documentation/server/4.5/developer-guide/querying.html

Indexes

Indexes[10] in Couchbase are just as important as in relational databases. Probably more so, because buckets contain a variety of documents.

To enable N1QL queries on a bucket, at the very least you need to create a primary index. This is an index on the bucket itself. Here's how to create one with N1QL: `CREATE PRIMARY INDEX ON \`my-bucket\``

You can create indexes based on the fields in the documents. For instance, if you have a lot of documents that have "name" or "author", and you will often be querying based on this fields, you can create indexes for them. These are called "secondary indexes."

3. ASP.NET MVC

Now that we've got some lingo out the way, let's start a new ASP.NET MVC project, add the Couchbase SDK to it with NuGet, and get the infrastructure in place to start using Couchbase.

I just started in Visual Studio with a File→New, and selected ASP.NET Web Application, then selected "MVC". I'm going to assume you have some familiarity with ASP.NET MVC (I'm going to use traditional .NET, but an .NET Core Couchbase SDK is coming soon).

3.1. Installing the Couchbase client library

The first thing we'll need to do is add the Couchbase .NET client. You can do this with the NuGet UI by right-clicking on "References", clicking "Manage NuGet Packages", clicking "Browse", and then searching for "CouchbaseNetClient". (If you want to, you can search for "Linq2Couchbase" instead. Installing that will also cause CouchbaseNetClient to be installed, but I won't actually be using any Linq2Couchbase until later blog posts).

If you prefer the NuGet command line, then open up the Package Manager Console, and type `Install-Package CouchbaseNetClient`.

3.2. Getting ASP.NET to talk to a Couchbase cluster

Now let's setup the ASP.NET app to be able to connect to Couchbase. The first thing we need to do is locate the Couchbase Cluster. The best place to do this is in the `Global.asax.cs`

[10] http://developer.couchbase.com/documentation/server/4.5/indexes/n1ql-in-couchbase.html

when the application starts. At a minimum, we need to specify one node in the cluster, and give that to the `ClusterHelper`. This only needs to be done once in `Application_Start`. When the application ends, it's a good idea to close the `ClusterHelper`.

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        var config = new ClientConfiguration();
        config.Servers = new List<Uri>
        {
            new Uri("http://localhost:8091")
        };
        config.UseSsl = false;
        ClusterHelper.Initialize(config);
    }

    protected void Application_End()
    {
        ClusterHelper.Close();
    }
}
```

3.3. Using the IBucket in a controller

Just to show that this works, go ahead and add IBucket to a constructor of a controller, say `TestController`.

```
public class TestController : Controller
{
    IBucket _bucket;

    public TestController()
    {
        _bucket = ClusterHelper.GetBucket("default");
    }
}
```

(In the long run, we probably don't want an IBucket being used directly in MVC controllers, more on that later).

Next, add a document to your bucket, directly in Couchbase Console. Make note of the key you give it.

Now, add an action to `TestController`. This action is the simplest thing that can be done: it will get the document based on the key, and write the document values in the response.

```
public ActionResult Index()
{
    var doc = _bucket.Get<dynamic>("foo::123");
    return Content("Name: " + doc.Value.name + ", Address: " + doc.Value.address);
}
```

`doc.Value` is of type `dynamic`, so make sure that the fields you use (in my case, name and address) match up to the JSON document you put into the bucket. Run your MVC site in a browser, and you should see something like this:

Congratulations, you've successfully written an ASP.NET site that uses Couchbase!

4. Introducing Linq2Couchbase

I'm going to build on what we've already done by introducing Linq2Couchbase[11]. I'll also move Couchbase out of the Controller and put it into a very basic repository[12] class. My goal is to have you feeling comfortable with the basics of Couchbase and Linq2Couchbase, and be able to start applying it in your web application.

4.1. Moving Couchbase out of the Controller

The Controller's job is to direct traffic: take incoming requests, hand them to a model, and then give the results to the view. To follow the SOLID principles[13] (specifically the Single Responsibility Principle), data access should be somewhere in a "model" and not the controller.

[11] https://github.com/couchbaselabs/Linq2Couchbase
[12] http://www.martinfowler.com/eaaCatalog/repository.html
[13] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

The first step is to refactor the existing code. We can keep the 'really simple example', but it should be moved to a method in another class. Here is the refactored HomeController and the new PersonRepository:

```
public class HomeController : Controller
{
    private readonly PersonRepository _personRepo;

    public HomeController(PersonRepository personRepo)
    {
        _personRepo = personRepo;
    }

    public ActionResult Index()
    {
        var person = _personRepo.GetPersonByKey("foo::123");
        return Content("Name: " + person.name + ", Address: " + person.address);
    }
}

public class PersonRepository
{
    private readonly IBucket _bucket;

    public PersonRepository(IBucket bucket)
    {
        _bucket = bucket;
    }

    public dynamic GetPersonByKey(string key)
    {
        return _bucket.Get<dynamic>(key).Value;
    }
}
```

Now, `HomeController` no longer depends directly on Couchbase.

4.2. Refactoring to use a Person Class

In the above example, I'm using a `dynamic` object. `dynamic` is great for some situations, but in this case, it would be a good idea to come up with a more concrete definition of what a "Person" is. I can do this with a C# class.

```
public class Person
{
```

```
    public string Name { get; set; }
    public string Address { get; set; }
}
```

I'll also update the PersonRepository to use this class.

```
public Person GetPersonByKey(string key)
{
    return _bucket.Get<Person>(key).Value;
}
```

While we're at it, I'm going to take some steps to make this more of a proper MVC app. Instead of returning `Content()`, I'm going to make the Index action return a View, and I'm going to pass it a **list** of Person objects. I'll create an `Index.cshtml` file, which will delegate to a partial of _person.cshtml. I'm also going to drop in a layout that uses Bootstrap. This last part is completely gratuitous, but it will make the app look nicer.

New Index action:

```
public ActionResult Index()
{
    var person = _personRepo.GetPersonByKey("foo::123");
    var list = new List<Person> {person};
    return View(list);
}
```

Index.cshtml:

```
@model List<CouchbaseAspNetExample.Models.Person>

@{
    ViewBag.Title = "Home : Couchbase & ASP.NET Example";
}

@if (!Model.Any())
{
    <p>There are no people yet.</p>
}

@foreach (var item in Model)
{
    @Html.Partial("_person", item)
}
```

_person.cshtml:

```
@model CouchbaseAspNetExample.Models.Person

<div class="panel panel-default">
    <div class="panel-heading">
        <h2 class="panel-title">@Model.Name</h2>
    </div>
    <div class="panel-body">
        @Html.Raw(Model.Address)
    </div>
</div>
```

Now it looks a little nicer. Additionally, we'll be able to show a whole list of Person documents later in the demo.

4.3. Linq2Couchbase

Earlier I mentioned that Couchbase Server supports a query language known as N1QL[14]. It's a superset of SQL, and allows you to leverage your existing knowledge of SQL to construct very powerful queries over JSON documents in Couchbase. Linq2Couchbase takes this a step further and converts Linq queries into N1QL queries (much like Entity Framework converts Linq queries into SQL queries).

Linq2Couchbase is part of Couchbase Labs[15], and is not yet part of the core, supported Couchbase .NET SDK library. However, if you're used to Entity Framework, NHibernate.Linq, or any other Linq provider, it's a great way to introduce yourself to Couchbase. For some operations, you will still need to use the core Couchbase .NET SDK, but there is a lot we can do with Linq2Couchbase.

Start by adding Linq2Couchbase with NuGet (if you haven't already).

N1QL (and therefore Linq2Couchbase) depends on the bucket being indexed[16]. Go into Couchbase Console, click the 'Query' tab, and create a primary index on the `default` bucket.

[14] http://www.couchbase.com/n1ql
[15] https://github.com/couchbaselabs
[16] http://developer.couchbase.com/documentation/server/4.5/n1ql/n1ql-language-reference/createprimaryindex.html

```
CREATE PRIMARY INDEX ON `default`;
```

If you don't have an index, Linq2Couchbase will give you a helpful error message like "No primary index on keyspace default. Use CREATE PRIMARY INDEX to create one."

In order to use Linq2Couchbase most effectively, we have to start giving Couchbase documents a "type" field. This way, we can differentiate between a "person" document and a "location" document. In this example, I'm only going to have "person" documents, but it's a good idea to do this from the start. I'll create a `Type` field, and set it to "Person". I'll also put an attribute on the C# class so that Linq2Couchbase understands that this class is meant for a certain type of document.

```
using Couchbase.Linq.Filters;

[DocumentTypeFilter("Person")]
public class Person
{
    public Person()
    {
        Type = "Person";
    }
    public string Type { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
}
```

If you make these changes, your app will continue to work. This is because we are still retrieving the document by its key. But now let's change the Index action to try and get ALL Person documents.

```
public ActionResult Index()
{
    var list = _personRepo.GetAll();
    return View(list);
}
```

We'll implement that new GetAll repository method using Linq2Couchbase:

```
using System.Collections.Generic;
using System.Linq;
using Couchbase.Core;
```

```
using Couchbase.Linq;
using Couchbase.Linq.Extensions;
using Couchbase.N1QL;

public class PersonRepository
{
    private readonly IBucket _bucket;
    private readonly IBucketContext _context;

    public PersonRepository(IBucket bucket, IBucketContext context)
    {
        _bucket = bucket;
        _context = context;
    }

    public List<Person> GetAll()
    {
        return _context.Query<Person>()
           .ScanConsistency(ScanConsistency.RequestPlus)
           .OrderBy(p => p.Name)
           .ToList();
    }
}
```

In this example, I'm telling Couchbase to order all the results by Name. If you'd like, you can experiment with the normal Linq methods that you're used to: `Where`, `Select`, `Take`, `Skip`, and so on.

Just ignore that `ScanConsistency` for now: I'll discuss it more later. But what about that `IBucketContext`? The `IBucketContext` is similar to `DbContext` for Entity Framework, or `ISession` for NHibernate. To get that `IBucketContext`, I'll make some changes to `HomeController`.

```
public HomeController()
{
    var bucket = ClusterHelper.GetBucket("default");
    var bucketContext = new BucketContext(bucket);
    _personRepo = new PersonRepository(bucket, bucketContext);
}
```

I'm doing it this way for simplicity, but I recommend that you use a Dependency Injection (like StructureMap) to handle this, otherwise you'll end up copy/pasting a lot of code into all of your Controllers.

Now, if you compile and run the web app again, it will display "There are no people yet". Hey, where did I go?! I didn't show up because the "foo::123" document doesn't have a "type" field yet. Go to Couchbase Console and add it.

Once you do that, refresh your web page, and the person will appear again.

4.4. A quick note about ScanConsistency

Linq2Couchbase relies on an Index in order to generate and execute queries. When you add new documents, the index must be updated. Until the index gets updates, any documents not yet indexed will not be returned by Linq2Couchbase (by default). By adding in `ScanConsistency` of `RequestPlus` (See Couchbase documentation for the details about scan consistency[17]), Linq2Couchbase will effectively wait until the index is updated before executing a query and returning a response. This is a tradeoff that you will have to think about when designing your application. Which is more important: raw speed or complete accuracy? The Couchbase SDK defaults to raw speed.

5. A Complete ASP.NET CRUD implementation

Let's round out the sample app that I've been building with a full suite of CRUD functionality. The app already shows a list of people. After this post, you'll be able to:

- Add a new person via the web app (instead of directly in Couchbase Console)
- Edit a person
- Delete a person.

Before I start, a disclaimer. I've made some modeling **decisions** in this sample app. I've decided that keys to Person documents should be of the format "Person::{guid}", and I've decided that I will enforce the "Person::" prefix at the repository level. I've also made a decision not to use any intermediate view models or edit models in my MVC app, for the purposes of a concise demonstration. By no means do you have to make the same decisions I did! I encourage you to think through the implications for your particular use case, and I would be happy to discuss the mertis and trade-offs of each approch in the comments or in the Couchbase Forums[18].

[17] http://developer.couchbase.com/documentation/server/4.5/architecture/querying-data-with-n1ql.html
[18] http://forums.couchbase.com

5.1. Adding a new person document

In the previous blog posts, I added new documents through the Couchbase Console. Now let's make it possible via a standard HTML form on an ASP.NET page.

First, I need to make a slight change to the Person class:

```
[DocumentTypeFilter("Person")]
public class Person
{
    public Person() { Type = "Person"; }

    [Key]
    public string Id { get; set; }
    public string Type { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
}
```

I added an "Id" field, and marked it with the `[Key]` attribute. This attribute comes from System.ComponentModel.DataAnnotations, but Linq2Couchbase interprets it to mean "use this field for the Couchbase key".

Now, let's add a very simple new action to `HomeController`:

```
public ActionResult Add()
{
    return View("Edit", new Person());
}
```

And I'll link to that with the bootstrap navigation (which I snuck in previously, and by no means are you required to use):

```
<ul class="nav navbar-nav">
    <li><a href="/">Home</a></li>
    <li>@Html.ActionLink("Add Person", "Add", "Home")</li>
</ul>
```

Nothing much out of the ordinary so far. I'll create a simple `Edit.cshtml` with a straightforward, plain-looking form.

```
@model CouchbaseAspNetExample3.Models.Person
```

```
@{
    ViewBag.Title = "Add : Couchbase & ASP.NET Example";
}

@using (Html.BeginForm("Save", "Home", FormMethod.Post))
{
    <p>
        @Html.LabelFor(m => m.Name)
        @Html.TextBoxFor(m => m.Name)
    </p>

    <p>
        @Html.LabelFor(m => m.Address)
        @Html.TextBoxFor(m => m.Address)
    </p>

    <input type="submit" value="Submit" />
}
```

Since that form will be POSTing to a Save action, that needs to be created next:

```
[HttpPost]
public ActionResult Save(Person model)
{
    _personRepo.Save(model);
    return RedirectToAction("Index");
}
```

Notice that the `Person` type used in the parameter is the same type as before. Here is where a more complex web application would probably want to use an edit model, validation, mapping, and so on. I've omitted all of that, and I send the model straight to a new method in `PersonRepository`:

```
public void Save(Person person)
{
    // if there is no ID, then assume this is a "new" person
    // and assign an ID
    if (string.IsNullOrEmpty(person.Id))
        person.Id = "Person::" + Guid.NewGuid();

    _context.Save(person);
}
```

This repository method will set the ID, if one isn't already set (it won't be now, but it will be later, when we cover 'Edit'). The "Save" method on `IBucketContext` is from Linq2Couchbase. It will add a new document if the key doesn't exist, or update an existing document if it does. It's known as an "upsert" operation. In fact, I can do nearly the same thing without Linq2Couchbase:

```
var doc = new Document<Person>
{
    Id = "Person::" + person.Id,
    Content = person
};
_bucket.Upsert(doc);
```

5.2. Editing an existing person document

Now, I want to be able to edit an existing person document in my ASP.NET site. First, let's add an edit link to each person, by making a change to `_person.cshtml` partial view.

```
<h2 class="panel-title">
    @Model.Name
    @Html.ActionLink("[Edit]", "Edit", new {id = Model.Id.Replace("Person::", "")})
    @Html.ActionLink("[Delete]", "Delete", new {id = Model.Id.Replace("Person::",
 "")}, new { @class="deletePerson"})
</h2>
```

I also added a "delete" link while I was in there, which we'll get to later. One more thing to point out: when creating the routeValues, I stripped out "Person::" from the Id. If I don't do this, ASP.NET will complain about a potentially malicious HTTP request. It would probably be better to give each person a document a more friendly "slug"[19] to use in the URL, or maybe to use that as the document key. That's going to depend on your use case and your data design.

Now I need an `Edit` action in HomeController:

```
public ActionResult Edit(Guid id)
{
    var person = _personRepo.GetPersonByKey(id);
    return View("Edit", person);
}
```

[19] https://en.wikipedia.org/wiki/Semantic_URL#Slug

I'm reusing the same `Edit.cshtml` view, but now I need to add a hidden field to hold the document ID.

```
<input type="hidden" name="Id" value="@Model.Id"/>
```

Alright! Now we can add and edit person documents.

This may not be terribly impressive to those of you already comfortable with ASP.NET MVC. So, next, let's look at something cool that a NoSQL database like Couchbase brings to the table.

5.3. Iterating on the data stored in the person document

My new requirement is that I want to collect more information about a Person. Let's say I want to get a phone number, and a list of that person's favorite movies. With a relational database, that means that I would need to add *at least* two columns, and more likely, at least one other table to hold the movies, with a foreign key.

With Couchbase, there is no explicit schema. Instead, all I have to do is add a couple more properties to the Person class.

```
[DocumentTypeFilter("Person")]
public class Person
{
    public Person() { Type = "Person"; }

    [Key]
    public string Id { get; set; }
    public string Type { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }

    public string PhoneNumber { get; set; }
    public List<string> FavoriteMovies { get; set; }
}
```

That's pretty much it, except that I also need to add a corresponding UI. I used a bit of jQuery to allow the user to add any number of movies. I won't show the code for it here, because the implementation details aren't important. But I have made the whole sample available on Github[20], so you can follow along or check it out later if you'd like.

[20] https://github.com/couchbaselabs/blog-source-code/tree/master/Groves/017MVPBlog/CouchbaseAspNetExample

I also need to make changes to `_person.cshtml` to (conditionally) display the extra information:

```
<div class="panel-body">
    @Model.Address
    @if (!string.IsNullOrEmpty(Model.PhoneNumber))
    {
        <br />
        @Model.PhoneNumber
    }
    @if (Model.FavoriteMovies != null && Model.FavoriteMovies.Any())
    {
        <br/>
        <h4>Favorite Movies</h4>
        <ul>
            @foreach (var movie in Model.FavoriteMovies)
            {
                <li>@movie</li>
            }
        </ul>
    }
</div>
```

And here's how that would look (this time with two Person documents):

I didn't have to migrate a SQL schema. I didn't have to create any sort of foreign key relationship. I didn't have to setup any OR/M mappings. I simply added a couple of new fields, and Couchbase turned it into a corresponding JSON document.

5.4. Deleting a person document

I already added the "Delete" link, so I just need to create a new Controller action…

```
public ActionResult Delete(Guid id)
{
    _personRepo.Delete(id);
    return RedirectToAction("Index");
```

```
}
```

…and a new repository method:

```
public void Delete(Guid id)
{
    _bucket.Remove("Person::" + id);
}
```

Notice that this method is not using Linq2Couchbase. It's using the `Remove` method on `IBucket`. There is a `Remove` available on `IBucketContext`, but you need to pass it an object, and not just a key. I elected to use the `IBucket`, but there's nothing inherently superior about it.

5.5. Wrapping up

Thanks for reading through this blog post. Hopefully, you're on your way to considering or even including Couchbase in your next ASP.NET project. Here are some more interesting links for you to continue your Couchbase journey:

- You might be interested in the ASP.NET Identity Provider for Couchbase[21] (github[22]). If you want to store identity information in Couchbase, this is one way you could do it. At the time of this blog post, it's an early developer preview, and is missing support for social logins.
- Linq2Couchbase is a great project with a lot of features and documentation, but it's still a work in progress. If you are interested, I suggest visiting Linq2Couchbase on Github[23]. Ask questions on Gitter, and feel free to submit issues or pull requests.

6. Conclusion

I've put the full source code for this example on Github[24].

What did I leave out? What's keeping you from trying Couchbase with ASP.NET today? Please leave a comment, [ping me on Twitter](http://twitter.com/mgroves), or email me (matthew.groves AT couchbase DOT com). I'd love to hear from you.

[21] http://blog.couchbase.com/2015/july/the-couchbase-asp.net-identity-storage-provider-part-1
[22] https://github.com/couchbaselabs/couchbase-aspnet-identity
[23] https://github.com/couchbaselabs/Linq2Couchbase
[24] https://github.com/couchbaselabs/blog-source-code/tree/master/Groves/017MVPBlog/CouchbaseAspNetExample