Couchbase with Windows and .NET

We're going to look at the basics of interacting with Couchbase for .NET developers on Windows. We'll start with the basics, and build towards a "vertical" slice of a complete ASP.NET MVC app on the .NET 4.x framework. For a deeper dive, 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:

1.1. Download and install Couchbase Server[3].

Windows 10, Mac OSX and Linux packages are all available. Go to the Downloads page on the Couchbase website[4]. We'll use the Enterprise Edition for this blog post. Let's proceed with a Windows 10 installation.

Run the downloaded installation file, and go through a 4-step wizard.

At this point, Couchbase should be running as a Windows Service (verify by opening services.msc[5]).

We'll automatically be taken to the Couchbase Console via web browser to customize the Couchbase setup. We can always return to this console via http://localhost:8091.

Click "Setup" to start the configuration wizard. Let's stick with the default settings (mostly) for now.

1. In step 1, we need to specify how much RAM to give to Couchbase. It might be a good idea to dial down some of the default RAM Quotas (we can always change them later). Since we only need the one node for development, let's enabled all services on this node

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

(Index, Query, and Full Text). I would recommend checking out some of the free Couchbase training[6] available to dive deeper into tweaking these settings.

1. In step 2, we can choose to install some sample data. For this blog post, we 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, let's create a "default" bucket, to follow along with this blog post.
3. In step 4, we can elect to receive notifications from Couchbase and register the product.
4. Finally, in step 5, we need to enter a username and password to create an administrator account.

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

Feel free to play around with the Couchbase Console and check out the Couchbase Console documentation[7].

2. Some NoSQL lingo

But before we dive into some code, let's go over some of the lingo with Couchbase. It's not a difficult tool to use, but it is different from RDBMS systems like SQL Server.

Node

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

Cluster

Nodes are joined together into a cluster[8] 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 developers, we don't generally have to worry about node management and clustering in code: it's all handled by Couchbase and the Couchbase .NET SDK.

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

Bucket

A bucket[9] 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. We could store a document that contains information about a user, and a document with information about a building, both in the same bucket.

Document

In a basic sense, a Couchbase bucket is 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

With Couchbase Server, we can write queries in a language called N1QL[10] (N1QL stands for "Non-first Normal Form Query Language, 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 we'll see later, the Linq2Couchbase library leverages N1QL into a LINQ provider that is similar to Entity Framework, NHibernate.Linq, etc.

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

Indexes

Indexes[11] in Couchbase are as important as in relational databases. Probably more so, because buckets contain a variety of documents.

To enable N1QL queries on a bucket, we need to at least 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``

We can create indexes based on the fields in the documents. For instance, if we have a lot of documents that have "name" or "author" fields, and we end up querying based on this fields often, we can create indexes for them. These are "secondary indexes." As the size of a bucket grows, so does the importance of indexing.

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[12], and get the infrastructure in place to start using Couchbase.

Let's start in Visual Studio with a File→New, and select ASP.NET Web Application, then select "MVC". I'm going to assume you have some familiarity with ASP.NET MVC (we're using .NET 4.x, but a .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. We can do this with the NuGet UI by right-clicking on "References", clicking "Manage NuGet Packages", clicking "Browse", and then searching for "CouchbaseNetClient". (We could search for "Linq2Couchbase" instead. Installing that will also install CouchbaseNetClient, but we won't actually be using Linq2Couchbase until later).

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

[11] http://developer.couchbase.com/documentation/server/4.5/indexes/n1ql-in-couchbase.html
[12] https://www.nuget.org/packages/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` when the application starts. At a minimum, we need to specify one node in the cluster, and give that to the `ClusterHelper`. We only need this 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 don't want an IBucket directly in MVC controllers, more on that later).

Next, let's add a document to the bucket, directly in Couchbase Console. Use anything for a key, but make a note of it.

Now, let's add an action to `TestController`. 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 (in this case, "name" and "address") match up to the JSON document in the bucket. Run the MVC site in a browser, and we should see something like this:

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

4. Introducing Linq2Couchbase

Let's build on what we've already done by introducing Linq2Couchbase[13]. We'll also move Couchbase out of the Controller and put it into a very basic repository[14] class. My goal is to have you feeling comfortable with the basics of Couchbase and Linq2Couchbase, and be able to start applying it in a web application.

[13] https://github.com/couchbaselabs/Linq2Couchbase
[14] http://www.martinfowler.com/eaaCatalog/repository.html

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[15] (specifically the Single Responsibility Principle), data access should be somewhere in a "model" and not the controller.

The first step is to refactor the existing code. We can keep the 'really simple example', but let's move it 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.

[15] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

4.2. Refactoring to use a Person class

In the above example, we're 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. We can do this with a C# class.

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

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

Couchbase Server supports a query language known as N1QL[16]. It's a superset of SQL, and allows us to leverage existing knowledge of SQL syntax 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[17], and is not yet part of the core, supported Couchbase .NET SDK library. However, if you're used to Entity Framework, NHibernate.Linq,

[16] http://www.couchbase.com/n1ql
[17] https://github.com/couchbaselabs

or any other LINQ provider, it's a great way to introduce yourself to Couchbase. For some operations, we 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).

To use N1QL (and therefore Linq2Couchbase), the bucket must be indexed[18]. Go into Couchbase Console, click the 'Query' tab, and create a primary index on the `default` bucket.

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

If we don't have an index, Linq2Couchbase will throw 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, we're only going to have "person" documents, but it's a good idea to do this from the start. We'll create a `Type` field, and set it to "Person". We'll also put an attribute on the C# class so that Linq2Couchbase understands that this class corresponds to a certain document type.

```
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; }
}
```

After we make these changes, the 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.

[18] http://developer.couchbase.com/documentation/server/4.5/n1ql/n1ql-language-reference/createprimaryindex.html

```
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, we're telling Couchbase to order all the results by Name. At this point, we can experiment with the normal LINQ methods: `Where`, `Select`, `Take`, `Skip`, and so on.

Just ignore that `ScanConsistency` for now: we'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`, we'll make some changes to `HomeController`.

```
public HomeController()
```

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

We're doing it this way for simplicity, but I recommend that you use a dependency injection[19] framework (like StructureMap[20]) to handle this, otherwise you'll end up copy/pasting a lot of code into your Controllers.

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

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

4.4. A quick note about ScanConsistency

Linq2Couchbase relies on an Index to generate and execute queries. Adding a new documents triggers an index update. Until the index finishes updating, 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[21]), 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 we've been building with a full suite of CRUD functionality. The app already shows a list of people. We'll next want to:

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

[19] https://msdn.microsoft.com/en-us/library/ff921152.aspx
[20] http://structuremap.github.io/
[21] http://developer.couchbase.com/documentation/server/4.5/architecture/querying-data-with-n1ql.html

- 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 we 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 merits and trade-offs of each approch in the comments or in the Couchbase Forums[22].

5.1. Adding a new person document

Up until now, we've used the Couchbase Console to create new documents. Now let's make it possible via a standard HTML form on an ASP.NET page.

First, we 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; }
}
```

We 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());
}
```

[22] http://forums.couchbase.com

And We'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. We'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, let's create that 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 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[23]. In fact, we 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, we 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>
```

We also added a "delete" link while we were in there, which we'll get to later. One more thing to point out: when creating the routeValues, we stripped out "Person::" from `Id`. If we don't do

[23] http://developer.couchbase.com/documentation/server/current/n1ql/n1ql-language-reference/upsert.html

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"[24] 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 we need an `Edit` action in HomeController:

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

We're reusing the same `Edit.cshtml` view, but now we 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 anyone 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

A new requirement is that we want to collect more information about a Person. Let's say we want to get a phone number, and a list of that person's favorite movies. With a relational database, that means that we 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 we 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; }
```

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

```
    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 we 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[25], so you can follow along or check it out later if you'd like.

We 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):

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

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

5.4. Deleting a person document

We already added the "Delete" link, so we 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`. A `Remove` method is available on `IBucketContext`, but we need to pass it an entire document, 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:

- There is a ASP.NET Identity Provider for Couchbase[26] (github[27]). At the time of this blog post, it's an early developer preview, and is missing support for social logins.

[26] http://blog.couchbase.com/2015/july/the-couchbase-asp.net-identity-storage-provider-part-1
[27] https://github.com/couchbaselabs/couchbase-aspnet-identity

- 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[28]. Ask questions on Gitter, and feel free to submit issues or pull requests.

I've put the full source code for this example on Github[29].

What did I leave out? What's keeping you from trying Couchbase with ASP.NET today? Please leave a comment, ping me on Twitter[30], or email me (matthew.groves AT couchbase DOT com). I'd love to hear from you.

[28] https://github.com/couchbaselabs/Linq2Couchbase
[29] https://github.com/couchbaselabs/blog-source-code/tree/master/Groves/017MVPBlog/CouchbaseAspNetExample
[30] http://twitter.com/mgroves