Couchbase with Windows and .NET - Part 2

In this three part series, 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].

In this first part, we installed Couchbase Server and went over the basics of how it works.

In the second part, we'll look at using ASP.NET with Couchbase Server.

In the final part, we'll implement all the CRUD functionality in an ASP.NET application.

1. 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[3], 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).

1.1. 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`.

1 http://blog.couchbase.com/
2 http://developer.couchbase.com
3 https://www.nuget.org/packages/CouchbaseNetClient/

1.2. 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();
    }
}
```

1.3. 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!

2. Laying the groundwork for Linq2Couchbase

Let's do some refactoring before introducing Linq2Couchbase[4]. We'll move Couchbase out of the Controller and put it into a very basic repository[5] class.

2.1. 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[6] (specifically the Single Responsibility Principle), data access should be somewhere in a "model" and not the controller.

[4] https://github.com/couchbaselabs/Linq2Couchbase
[5] http://www.martinfowler.com/eaaCatalog/repository.html
[6] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

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.

2.2. 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.

3. That's it for now

I've put the full source code for this example on Github[7]. Note that this source code represents the final product, so if you check it out now you will have a head start on the next and final blog post in the series.

Please leave a comment, ping me on Twitter[8], or email me (matthew.groves AT couchbase DOT com). I'd love to hear from you.

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

[8] http://twitter.com/mgroves