Couchbase with Windows and .NET - Part 3

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 looked at using ASP.NET with Couchbase Server.

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

1. Linq2Couchbase

Couchbase Server supports a query language known as N1QL[3]. 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[4], 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, 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[5]. Go into Couchbase Console, click the 'Query' tab, and create a primary index on the `default` bucket.

1 http://blog.couchbase.com/
2 http://developer.couchbase.com
3 http://www.couchbase.com/n1ql
4 https://github.com/couchbaselabs
5 http://developer.couchbase.com/documentation/server/4.5/n1ql/n1ql-language-reference/createprimaryindex.html

```
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.

```
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[6] framework (like StructureMap[7]) to handle this, otherwise you'll end up copy/pasting a lot of code into your Controllers.

[6] https://msdn.microsoft.com/en-us/library/ff921152.aspx
[7] http://structuremap.github.io/

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.

1.1. 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[8]), 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.

2. 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
- 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[9].

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

2.1. 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());
}
```

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[10]. 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);
```

2.2. 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 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"[11] 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);
}
```

[10] http://developer.couchbase.com/documentation/server/current/n1ql/n1ql-language-reference/upsert.html
[11] https://en.wikipedia.org/wiki/Semantic_URL#Slug

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.

2.3. 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; }
    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 [12], so you can follow along or check it out later if you'd like.

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

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):

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.

2.4. 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.

2.5. Wrapping up

Thanks for reading through this blog post series. 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[13] (github[14]). 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[15]. 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[16].

What did I leave out? What's keeping you from trying Couchbase with ASP.NET today? Please leave a comment, ping me on Twitter[17], or email me (matthew.groves AT couchbase DOT com). I'd love to hear from you.

[13] http://blog.couchbase.com/2015/july/the-couchbase-asp.net-identity-storage-provider-part-1
[14] https://github.com/couchbaselabs/couchbase-aspnet-identity
[15] https://github.com/couchbaselabs/Linq2Couchbase
[16] https://github.com/couchbaselabs/blog-source-code/tree/master/Groves/017MVPBlog/CouchbaseAspNetExample
[17] http://twitter.com/mgroves