AngularJS Tutorial (SQLite)

In this tutorial we turn to look at server-based web applications where the client-side code is AngularJS.

Although a lot can be done with entirely browser-based (single-page) web applications, it is better to develop a server-based web application if any of the following are true:

- In-company (intranet) client machines may have restricted access to the Internet (important e.g. in banking)
- Modifiable data is not just on the client (belongs to the company or is shared between clients)
- The data sets accessed by the application are (or may eventually become) large
- Authentication and authorisation of clients is important
- The web application consists of many related pages, styles etc and it is important to ensure everything is up to date and works together

For server-based web applications, It is not a good idea to use technologies that draw components from the web at runtime, because most browsers disallow web requests except to the application s hosts (XSS/XDS, cross-site/domain scripting).

After development, our web application will be deployed to a production hosting environment. Since server virtualisation is so popular today, we may suppose that the (virtual) machine that hosts the service has no other applications running. This means that there really is no point in using IIS or Apache. (Deploying to IIS is particularly difficult.) ASP.NET MVC is a great system but as we will see later, it does a great many unnecessary things on the server, and its worst design decision is to have its client-server interactions normally exchange HTML.

A large web application will consist of many scripted pages, will have multiple controllers, modules and databases, that all need to be assembled, so it is a good idea to have a project-based IDE such as Visual Studio, provided we avoid adopting such overly-tempting options as IIS, EF etc.

In our design we will focus on fully-scalable web application architecture. This means avoiding scripting on the web server, and using databases that could scale to distributed and partitioned systems (this will make DBMS such as MongoDB and even PyrrhoDB attractive). All systems start small, so we begin with the smallest possible implementation.

A very simple web server will do fine, so it might as well be a custom web server for our application. So we will use a very simple platform neutral solution written for .NET called AWebSvr. Extract the AWebSvr files in the folder where you got these notes. It supports JSon transfers between client and server, which is useful for AngularJS.

The database technology is a more interesting question: if in the end we want an efficient distributed/partitioned system, but start with an in-process (embedded) database, the alternatives shrink rapidly. And if we want an initially free version, that leaves us with SQLCe, SQLite (assuming we really can import our solution into a full SQL Server) and PyrrhoDB.

As suggested above we will totally avoid Entity Frameworks (Microsoft) and Entity Managers (JEE): they make transactions much harder. We will work to a completely stateless model on the server (no session state), and this makes server side objects (such as data adapters) useless except as transient objects for implementing business logic and for possible communication with other services.

In fact our server-side Views will always be static and cacheable. It is really much more efficient to use AJAX-style requests from the client to get any dynamic data. It is a good idea to repeat any client-side validation on the server, but there is always a design decision about how much business logic to implement in the database. (In the limit of scalability, the database is the best place to apply any rules that might span multiple partitions.)

As an example web application, we will use the Student-Subject one from the AngularJS Overview on tutorialspoint. The code we will study is here.

Getting Started

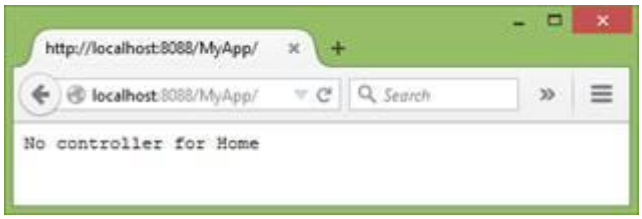
1. Start up any edition of Visual Studio (2005+) as Administrator, and create a new Visual C# Console Application project called AngularJS. (File>New Project..>Visual C#>Console Application, change the location to somewhere you control not your network drive, and give the project name as AngularJS. Click OK.)

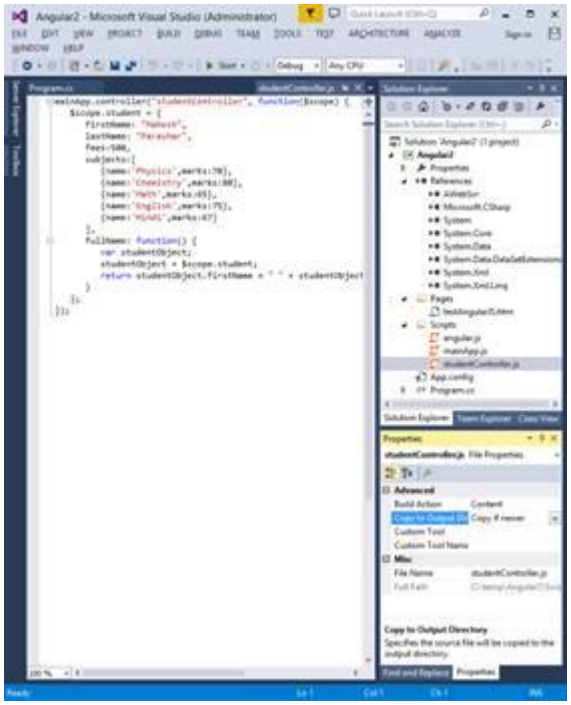
2. In Solution Explorer, right-click References and select Add Reference..>Browse>Browse.. to the folder where you extracted AWebSvr and add the .dll from there, click OK. (It is only 22 KB!). Change the project properties to use .NET 4.5.1.

3. Replace the entire contents of Program.cs with:

```
using AWebSvr;
namespace AngularJS
{
  class Program : WebSvr
  {
    static void Main(string[] args)
    {
      new Program().Server("http://+:8088/MyApp/");
    }
  }
}
```

4. Your program will already run (Click the green arrow): it shows a blank application window. Use a browser to try the web address http://localhost:8088/MyApp . With IE you get a 404 Not Found error at this stage; with Chrome or Firefox you get a message No controller for Home . If you get some other error, check everything and fix it before going any further (you could try a different port number).





5. Close the browser and the application window. (*Closing* the browser is important in this tutorial, because we will be encouraging client-side caching of scripts and credentials. Leaving browsers open will often lead to errors.)

Add AngularJS support

6. In Solution Explorer, right-click the project and select Add> New folder, to add a folder called Scripts.

7. Download angular.js from AngularJS (there is no need to run it).

8. Now in Solution Explorer right-click the Scripts folder, and Add>Existing Item..> browse to where you downloaded to, from the dropdown select Script Files (*.js,*.wsf), select angular.js, and click Add.

9. In Solution Explorer, right-click angular.js and select Properties. In the Properties window find the item Copy to Output, and use the dropdown to change this to Copy if newer.

Add the AngularJS sample

10. In Solution Explorer, right-click the project and select Add>New folder to add a folder called Pages.

11. Extract the following items from near the end of

    http://www.tutorialspoint.com/angularjs/angularjs_modules.htm:

testAngular.htm, mainApp.js, and studentController.js. (using Notepad or the file manager, in either case using all-files to avoid having the file end in .txt). (As mentioned above, for best results work through the whole Google tutorial before this one.)

Change the script elements to read

```
<script src="angular.js"></script>
<script src="mainApp.js"></script>
<script src="studentController.js"></script>
```

12. In Solution Explorer, right-click Pages to Add>Existing Item.. the .htm file (use all files to be able to see it), and right-click Scripts to add the two js files.

13. Repeat step 9 for all 3 of these files.

14. Now run the program, and browse to http://localhost:8088/MyApp/testAngularJS.htm

    Internet Explorer (IE9+), Mozilla or Chrome should be fine. (At this point the sample has data for one student hard-coded.)

15. Close the browser and the application window, and look at the bin\Debug folder below your project folder. You will see the Pages and Scripts folders have been copied and that is where our little web server is finding the files.

16. (Optional) If you open source code of the AWebSvr project with Visual Studio, and look at the Get() method in WebSvr.cs you will see where it tests for and serves .js and .htm files.

```
        if (v.EndsWith(".htm"))
            return new StreamReader("Pages/" + v).ReadToEnd();
```

Adding Database Support

17. Download the appropriate version of System.Data.SQLLite.dll which for me was http://system.data.sqlite.org/blobs/1.0.97.0/sqlite-netFx451-static-binary-bundle-Win32-2013-1.0.97.0.zip. In Solution Explorer, right-click References, Add Reference..>and Browse to System.Data.SQLLite.dll, check the box and click OK.

18. With another Visual Studio create a new Console Application called SQLiteSetup and target .NET 4.5.1. Add Reference.. for System.Data.SQLite.dll as in step 17. Change the Program.cs to the following:

```
using System.Data.SQLite;
using System;

namespace SQLLiteSetup
{
    class Program
    {
        static SQLiteConnection m_dbConnection;
        static void Main(string[] args)
        {
            SQLiteConnection.CreateFile("WebService.sqlite");
            m_dbConnection = new
SQLiteConnection("DataSource=WebService.sqlite;Version=3;");
            m_dbConnection.Open();
            Act("create table \"Student\"(\"Id\" integer primary key,\"firstName\"
varchar(30),\"lastName\" varchar(30))");
            Act("insert into \"Student\" (\"firstName\",\"lastName\")
values('Mahesh','Parashar')");
            Act("create table \"Subject\"(\"student\" references \"Student\",\"name\"
varchar(20),\"marks\" integer)");
            Act("insert into \"Subject\" values (1,'Physics',70)");
            Act("insert into \"Subject\" values (1,'Chemistry',80)");
            Act("insert into \"Subject\" values (1,'Math',65)");
            Act("insert into \"Subject\" values (1,'English',75)");
            Act("insert into \"Subject\" values (1,'Hindi',67)");
            m_dbConnection.Close();
            Console.WriteLine("All worked apparently");
            Console.ReadLine();
        }
        static void Act(string sql)
        {
            var cmd = new SQLiteCommand(sql, m_dbConnection);
            cmd.CommandText = sql;
            cmd.ExecuteNonQuery();
        }
    }
    }
```

And run the program.

19. In Solution Explorer for the Angular 2 project, right-click AngularJS, Add>Existing Item.. and select WebService.sqlite from the bin\Debug folder of the SQLiteSetup project (you will need the All Files option).

20. In Solution Explorer, right-click WebService.sqlite and click Properties. Change the Copy to Output property to Copy always.

The server-side Model

21. Use Solution Explorer to add a New Folder to the project called Models. Right-click Models and Add>Class.. giving the Name as Student.cs (so the class will be called Student), and change the contents to

```
namespace AngularJS.Models
{
    public class Student
    {
        public long Id;
        public string firstName;
        public string lastName;
    }
}
```

Note the class needs to be public, and the fields must exactly match the columns of the Student table. (We use Reflection a lot.)

22. Similarly add a public class to Models called Subject.cs:

```
namespace AngularJS.Models
{
    public class Subject
    {
        public long student;
        public string name;
        public long marks;
    }
}
```

The Database Connection

23. We need to add support for retrieving our data from our database. Add a class to Models called Connect, with the following contents:

```
using System.Collections.Generic;
using System.Data.SQLite;

namespace AngularJS.Models
{
    public class Connect
    {
        public static Connect conn;
        public SQLiteConnection sqlc;
        public Connect(string cs)
        {
            sqlc = new SQLiteConnection(cs);
            sqlc.Open();
            conn = this;
        }
        public void Close()
        {
            sqlc.Close();
         }
      }
    }
```

24. Modify Program.cs to call new Connect(..); inside the Main method, and add using Models; after the first brace:

```
using AWebSvr;

namespace AngularJS
{
```

```
    using Models;
    class Program : WebSvr
    {
        static void Main(string[] args)
        {
            new Connect("DataSource=WebService.sqlite;Version=3");
            new Program().Server("http://+:8088/MyApp/");
            Connect.conn.Close();
        }
    }
    }
```

25. Eventually we will add more code in the Connect class to support our REST/CRUD operations. As a first step, let us add a method for Find, two curly braces in from the end:

```
   public C[] Find<C>(string s) where C:new()
   {
      var r = new List<C>();
      var c = sqlc.CreateCommand();
      var tp = typeof(C);
      c.CommandText = "select * from \"" + tp.Name + "\" where " + s;
      var rdr = c.ExecuteReader();
      while (rdr.Read())
      {
          var ob = new C();
          for (var i = 0; i < rdr.FieldCount; i++)
          {
              var n = rdr.GetName(i);
              var f = tp.GetField(n);
              if (f==null)
                  throw new System.Exception("Table "+tp.Name+" lacks field "+n);
              f.SetValue(ob, rdr[i]);
          }
          r.Add(ob);
      }
      rdr.Close();
      return r.ToArray();
  }
```

This Connect class doesn t contain anything specific to the sample, so it could be made part of the AWebSvr library. This would simplify the tutorial by not having to develop the Connect class as we go. But I haven't done this because (a) AWebSvr should not be specific to any particular DBMS and (b) the code we are putting in Connect is (IMO) interesting and useful knowledge (do try and make sure you understand each bit at each step). We discuss this a bit further in the Appendix.

First steps with CRUD support in the Model

26. Let us use this clever generic method to retrieve a single student from the database. To the Student.cs class add the following (two braces in from the end):

```
        public static Student Find(string fn,string sn)
        {
            var r = Connect.conn.Find<Student>("\"firstName\"='" + fn +
"' and \"lastName\"='" + sn + "'");
            if (r == null || r.Length == 0)
                return null;
            return r[0];
        }
```

27. Let s also get the Student class to hold relevant subject information. In Student.cs, let us add a field subjects:

```
public Subject[] subjects = null;
```

Taking this step is more controversial than it looks. It is building a data model in the middle tier: in general, as mentioned above, this is pointless because of our policy of minimising session state. There are two possible good reasons for doing this (a) there may be some business logic (e.g. required for validation), that is best done on the server but not in the database, and is made easier by this structure; (b) the client needs an API that sends the subject information along with the student details. In this case we will have reason (b) with a bit of generosity over whether the client actually needs an interface at this level (we will see it is more convenient). But we don t have to track changes to this array: the Student object containing it will be immediately forgotten (see step 38).

28. We could have another Find method that supports this list of subjects (FindWithSubjects?) or just also support this field in our Find method:

```
        public static Student Find(string fn, string sn)
        {
            var r = Connect.conn.Find<Student>("\"firstName\"='" + fn +
"' and \"lastName\"='" + sn + "'");
            if (r == null || r.Length == 0)
                return null;
            r[0].subjects = Connect.conn.Find<Subject>("\"student\"=" + r[0].Id);
            return r[0];
        }
```

29. A Find based on the ID field will also be useful later:

```
        public static Student Find(long id)
        {
            var r = Connect.conn.Find<Student>("\"Id\"=" + id);
            if (r == null || r.Length == 0)
                return null;
            r[0].subjects = Connect.conn.Find<Subject>("\"student\"=" + id);
            return r[0];
    }
```

Using AJAX to get data

30. We need to change the client-side application code to use all this new machinery. All dynamic data will come from AJAX calls, in the client-side controller. At the moment studentController.cs contains hardwired data. So let s replace everything in studentController.js with:

```
mainApp.controller("studentController", function ($scope, $http) {
  $scope.findStudent = function () {
    $scope.fullName = '';
    var url = 'http://localhost:8088/MyApp/Student/' + JSON.stringify($scope.student);
    $http.get(url).success(function (response) {
        $scope.student = response;
        $scope.fullName = $scope.student.firstName + " " + $scope.student.lastName;
        $scope.$digest();
    });
  };
});
```

There is quite a lot of JavaScript machinery here, and some AngularJS stuff that you should be relatively happy with if you have done Google s AngularJS tutorial before starting this one. The JavaScript mainApp object here was constructed in mainApp.js (loaded in the .htm file) and named mainApp . mainApp is connected to the web page in the first of the page, which also connects to a studentController which is defined in the above code extract. The controller uses the AngularJS $scope service, as all of the controllers in the tutorial do; and also uses the AngularJS $http service so we can make AJAX calls.

The A in AJAX stands for Asynchronous. The get method returns immediately with a promise object for handling the results of the server request, and it is normal to define a success function on this return value whose main job is to collect the response. If the request fails, the promise object will receive the error information (http status codes etc), NOT the browser. By default the error information is discarded: it is good practice to also define an error function for handling the error. It is defined in the same way: we follow .success (function (response) { ..}) with .error(function(response){..}) .

The JSON.stringify function serialises the student for us as a document in the url. This is fine in this application since the document is not large. In a more complex example we would use a specially constructed JavaScript object to send the bits we want.

Finally the $digest() call ensures that the web page is fully synchronised with the data that has just arrived from the web server.

31. We also add a Search button to the web page, as we really only want to call the server when we have finished typing the firstName and LastName. In testAngularJS.htm, change the 2 lines containing Enter last name: and {{student.fullName}} to read:

```
<tr><td>Enter first name:</td><td><input type="text" ng-model="student.firstName">
</td></tr>
<tr><td>Enter last name:</td><td><input type="text" ng-model="student.lastName">
</td></tr>
<tr><td><input type="button" ng-click="findStudent()" value="Search" /></td>
<td>{{fullName}}</td></tr>
```

A server-side Controller

32. The server now needs to be able to handle the url in step 33. We need to create our first server-side Controller. In Solution Explorer, right-click the project and Add>New Folder to create a Controllers folder.

33. Right-click the Controllers folder and select Add>Class.., give the name as StudentController.cs, and click Add.

34. Change the StudentController.cs contents to

```
using AWebSvr;

namespace AngularJS.Controllers
{
    using Models;
    public class Name
    {
        public string firstName;
        public string lastName;
    }
    class StudentController : WebCtlr
    {
        public static string GetStudent(WebSvc ws,Document d) // Name
        {
            var nm = d.Extract<Name>()[0];
            var s = Student.Find(nm.firstName, nm.lastName);
            return new Document(s).ToString();
        }
    }
}
```

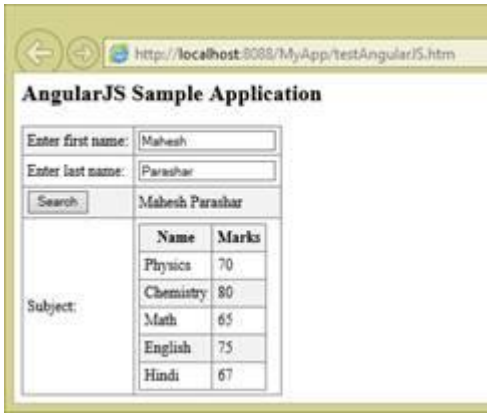
Note that as promised in step 31, the Student object s created and filled by Find is immediately forgotten.

35. We also need to get our WebSvr class to use this controller. In Program.cs just after using Models; add

```
using Controllers;
```

and add a line at the start of the Main method:

```
Add(new StudentController());
```



36. Now run the program, and browse to the same address as before. Type the correct name, and click Search.

37. Close the application window and the browser.

Adding a new Student

Now that we have got this far with our web application, let us follow a slightly more usual development cycle. This would proceed as follows: make a change to the page, support it in the client-side controller, and support it in a server-side controller. Some of these steps may not be required in some cases. Just now, there are extra complications since at some stage we need to enable the Connect class and our Server to post new data to the database. This will give an extra step or two.

38. Add a new button to testAngularJS.htm:

```
<tr><td><input type="button" ng-click="findStudent()" value="Search" />
  <input type="button" ng-click="addStudent()" value="Add" />
            </td><td>{{fullName}}</td></tr>
```

39. Add an AJAX post method to studentController.js. We want to do the computation about fullname each time, so let s rewrite the studentController.js as follows:

```
mainApp.controller("studentController", function ($scope, $http) {
  $scope.setStudent = function (r) {
    $scope.student = r;
    $scope.fullName = $scope.student.firstName + " " + $scope.student.lastName;
    $scope.$digest();
  };
  $scope.findStudent = function () {
    $scope.fullName = '';
    var url = 'http://localhost:8088/MyApp/Student/' + JSON.stringify($scope.student);
    $http.get(url).success(function (response) {
       $scope.setStudent(response);
    });
  };
  $scope.addStudent = function () {
     var url = 'http://localhost:8088/MyApp/Student/';
     $http.post(url, $scope.student).success(function (response) {
         $scope.setStudent(response);
     });
  };
});
```

40. Add a Post handler to Connect.cs (for other DBMS this might be provided in a REST interface):

```
public void Post(object ob)
{
```

```
            var c = sqlc.CreateCommand();
            var tp = ob.GetType();
            var sc = new StringBuilder();
            var sv = new StringBuilder();
            var cm = "";
            foreach (var f in tp.GetFields())
            {
                var v = f.GetValue(ob);
                if (v!=null)
                {
                    sc.Append(cm + "\"" + f.Name + "\"");
                    if (f.FieldType == typeof(string))
                        v = "'" + v + "'";
                    sv.Append(cm); sv.Append(v);
                    cm = ",";
                }
            }
            c.CommandText = "insert into \"" + tp.Name + "\"(" + sc + ")values(" + sv
+ ")";
            c.ExecuteNonQuery();
        }
```

You will also need using System.Text; at the top of Connect.cs.

41. We need to compute a new value for the primary key of Student. Add a GetScalar method in Connect.c:

```
        public object GetScalar(string s)
        {
            var c = sqlc.CreateCommand();
            c.CommandText = s;
            var rdr = c.ExecuteReader();
            object r = null;
            if (rdr.Read())
                r = rdr[0];
            rdr.Close();
            return r;
        }
```

42. Add a Post method to StudentController.cs:

```
 public static string PostStudent(WebSvc ws,Document d) // Name
 {
  //    Connect.conn.BeginTransaction();
     var nm = d.Extract<Name>()[0];
     var s = Student.Find(nm.firstName, nm.lastName);
     if (s != null)
        throw new Exception("We already have this");
     var m = (long)Connect.conn.GetScalar("select max(\"Id\") from \"Student\"");
     s = new Student();
     s.Id = m + 1;
     s.firstName = nm.firstName;
     s.lastName = nm.lastName;
     Connect.conn.Post(s);
     s = Student.Find(m + 1);
   //   Connect.conn.Commit();
     return new Document(s).ToString();
  }
```

You will also need using System; at the top of StudentController.cs. The last bit looks like overkill why not just return the new Id field? but s will contain some other automatic stuff later on.

43. Now try running the program, and use the web application to add yourself as a Student. If everything works, your fullName should get added to the form when your details are posted. We will add ways of handling new subject data in a moment.

Note that the running program is adding things to the copy of the database in bin\Debug, not the database that Server Explorer sees. This is appropriate during development: when we are re-running the program to find bugs it is useful if the database always starts with the same contents. And in fact we are going to retest adding yourself as a student in step 45.

Supporting transactions

As you can see in the above code, if we ever move to a multi-threaded web server or a server-based database we really should support transactions in this operation. (Or we could use a stored procedure in the database to encapsulate the steps involved.) But since it is quite instructive we will add transaction support to our SqlClient library class Connect.cs.

44. We want some extra things in Connect.cs. At the top add using System; then as highlighted:

```
class Connect
{
    public static Connect conn;
    public SQLiteConnection sqlc;
    public SQLiteTransaction tr = null;
    public Connect(string cs)
    {
        sqlc = new SQLiteConnection(cs);
        sqlc.Open();
        conn = this;
    }
    public void Close()
    {
        sqlc.Close();
    }
    public void BeginTransaction()
    {
        tr = sqlc.BeginTransaction();
    }
    public void Commit()
    {
        tr.Commit();
        tr = null;
    }
    public void Rollback()
    {
        tr.Rollback();
        tr = null;
    }
    public C[] Find<C>(string s) where C:new()
    {
        var r = new List<C>();
        var c = sqlc.CreateCommand();
        if (tr != null)
            c.Transaction = tr;
        var tp = typeof(C);
        c.CommandText = "select * from " + tp.Name + " where " + s;
        var rdr = c.ExecuteReader();
        while (rdr.Read())
```

```
            {
                var ob = new C();
                for (var i = 0; i < rdr.FieldCount; i++)
                {
                    var n = rdr.GetName(i);
                    var f = tp.GetField(n);
                    f.SetValue(ob, rdr[i]);
                }
                r.Add(ob);
            }
            rdr.Close();
            return r.ToArray();
        }
        public void Post(object ob)
        {
            var c = sqlc.CreateCommand();
            if (tr != null)
                c.Transaction = tr;
            var tp = ob.GetType();
            var sc = new StringBuilder();
            var sv = new StringBuilder();
            var cm = "";
            foreach (var f in tp.GetFields())
            {
                var v = f.GetValue(ob);
                if (v!=null)
                {
                    sc.Append(cm + "[" + f.Name + "]");
                    if (f.FieldType == typeof(string))
                        v = "'" + v + "'";
                    sv.Append(cm);  sv.Append(v);
                    cm = ",";
                }
            }
            c.CommandText = "insert into ["+tp.Name+"]("+sc+")values("+sv+")";
            try
            {
                c.ExecuteNonQuery();
            }
            catch (SQLiteException e)
            {
                Console.WriteLine(e.Message);
                if (tr != null)
                    Rollback();
            }
        }
        public object GetScalar(string s)
        {
            var c = sqlc.CreateCommand();
            if (tr != null)
                c.Transaction = tr;
            c.CommandText = s;
            var rdr = c.ExecuteReader();
            object r = null;
            if (rdr.Read())
                r = rdr[0];
            rdr.Close();
            return r;
        }
    }
```

You will also need using System; in this file.

45. Now uncomment the lines about transactions in StudentController and retest the application. (This is only a first step in adding transaction safety to this application! We will return to this point at the end of this tutorial at step 74.) When you are happy with this (and have closed the application and browser), you can copy WebService.mdf and WebServioce_log.ldf from the Debug folder to the project folder, if you want to start future runs of the application from this point.

46. For handling Delete and Put, let s add a new method in Connect.cs:

```
public void Act(string s)
{
    var c = sqlc.CreateCommand();
    if (tr != null)
        c.Transaction = tr;
    c.CommandText = s;
    c.ExecuteNonQuery();
}
```

Changing the Subjects information

47. In order to add or modify the subjects information, in this application it seems best to open a form at the foot of the page. Change testAngularJS.htm by adding

```
        <span ng-show="fullName!=''">
        <input type="button" ng-click="delStudent()"
value="Delete">
        <input type="checkbox" ng-
model="editing">Edit</span>
        <div ng-show="editing">
            <table ng-show="editing">
                <tr><td>Subject:</td><td><input
type="text" ng-model="edit.name" /></td></tr>
                <tr><td>Mark:</td><td><input
type="text" ng-model="edit.marks" /></td></tr>
            </table>
            <input type="button" ng-
click="addSubject()" value="+">
        <input type="button" ng-click="updateMark()" value="!">
        <input type="button" ng-click="delSubject()" value="-">
    </div>
    <input type="button" ng-click="clear()" value="Clear">
```

between the second </table> and . Notice we are not placing these elements in a HTML Form: this is fairly standard practice for AJAX.

48. Our plan now is to add POST, DELETE and PUT for Subjects. Normal REST semantics would simply treat each subject entry in isolation, and then we would get our client to update the subject list alongside the server calls, but for better transactional behaviour, let us build a findStudent() into the server-side controllers, so that we refresh our entire picture of the student each time. Then studentController.js can become:

```
mainApp.controller("studentController", function ($scope, $http) {
    $scope.student = {};
    $scope.edit = {};
    $scope.setStudent = function (r) {
        $scope.student = r;
        $scope.fullName = $scope.student.firstName + " " + $scope.student.lastName;
        $scope.edit.student = $scope.student.Id;
        $scope.$digest();
```

```
    };
    $scope.findStudent = function () {
        $scope.fullName = '';
        var url = 'http://localhost:8088/MyApp/Student/' +
JSON.stringify($scope.student);
        $http.get(url).success(function (response) {
            $scope.setStudent(response);
        });
    };
    $scope.addStudent = function () {
        var url = 'http://localhost:8088/MyApp/Student/';
        $http.post(url, $scope.student).success(function (response) {
            $scope.setStudent(response);
        });
    };
    $scope.addSubject = function () {
        var url = 'http://localhost:8088/MyApp/Subject/';
        $http.post(url, $scope.edit).success(function (response) {
            $scope.setStudent(response);
        });
    };
    $scope.updateMark = function () {
        var url = 'http://localhost:8088/MyApp/Subject/';
        $http.put(url, $scope.edit).success(function (response) {
            $scope.setStudent(response);
        });
    };
    $scope.delSubject = function () {
        var url = 'http://localhost:8088/MyApp/Subject/' +JSON.stringify($scope.edit);
        $http.delete(url).success(function (response) {
            $scope.setStudent(response);
        });
    };
    $scope.delStudent = function () {
        var url = 'http://localhost:8088/MyApp/Student/' + $scope.student.Id + '/';
        $http.delete(url);
        $scope.clear();
    };
    $scope.clear = function () {
        $scope.student = {};
        $scope.edit = {};
        $scope.editing = false;
        $scope.fullName = '';
    };
});
```

49. In Solution Explorer, right-click Controllers and Add>Class.. giving the file name as SubjectController.cs. Replace the contents with:

```
using AWebSvr;

namespace AngularJS.Controllers
{
    using Models;
    class SubjectController : WebCtlr
    {
        public static string PostSubject(WebSvc ws, Document d)
        {
            Connect.conn.BeginTransaction();
            var su = d.Extract<Subject>()[0];
            Connect.conn.Post(su);
            var s = Student.Find(su.student);
            Connect.conn.Commit();
```

```
            return new Document(s).ToString();
        }
        public static string PutSubject(WebSvc ws, Document d)
        {
            Connect.conn.BeginTransaction();
            var su = d.Extract<Subject>()[0];
            Connect.conn.Act("update \"Subject\" set \"marks\"=" + su.marks +
              " where \"student\"=" + su.student + " and \"name\"='" + su.name + "'");
            var s = Student.Find(su.student);
            Connect.conn.Commit();
            return new Document(s).ToString();
        }
        public static string DeleteSubject(WebSvc ws, Document d)
        {
            Connect.conn.BeginTransaction();
            var su = d.Extract<Subject>()[0];
            Connect.conn.Act("delete from \"Subject\" where \"student\"=" + su.student
                     + " and \"name\"='" + su.name + "'");
            var s = Student.Find(su.student);
            Connect.conn.Commit();
            return new Document(s).ToString();
        }
    }}
```

This time it was clearer to use Act to do the necessary updates and deletions rather than adding Put() or Delete() to Connect.cs.

50. In Program.cs add as the second line of Main():

```
            Add(new SubjectController());
```

Handling DELETE Student

51. We should remove the dependent Subject information if a student is deleted. Add a Delete method to StudentController.cs:

```
        public static string DeleteStudent(WebSvc ws, string id)
        {
            Connect.conn.BeginTransaction();
            Connect.conn.Act("delete from [Subject] where [student]=" + id);
            Connect.conn.Act("delete from [Student] where [Id]=" + id);
            Connect.conn.Commit();
            return "OK";
        }
```

52. All of this machinery should work now. Edit some subjects for your Student entry.

In the above code, all of the URLs were the same, there was only one AngularJS controller, and only one module. If you have more than one, you can use different URLs, controllers, modules, method names etc. And of course, your web application can have many Views. If they are all in the Pages folder, just use hyperlinks in the ordinary way.

When we get to multi-threading later on in this tutorial there will be a way of sharing data about the current request between different parts of the server-side code.

Authentication

In keeping with normal practice the server doesn t trust clients: the server will just check the user identity on each service call. There is some basic support in AWebSvr for authentication which we will exploit.

53. To activate AWebSvr s authentication mechanism, all we need to do is to have a Login page. To get started, let s make this very simple. In Solution Explorer, right-click Pages and Add>New Item..>Web>HTML page and give the name as Login.htm. Change the contents to:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Login Page</title>
</head>
  <body>
    Sorry: You are not registered on this site.
  </body>
</html>
```

54. Don t forget to change the Copy to Output Property of this file to Copy if newer . We need to do this for every new Page.

The server-side authentication list

Let s set up a User table in the database.

55. In another Visual Studio, open the SQLiteSetup project. In Program.cs add the lines

```
Act("create table \"User\"(\"userName\" varchar(20) primary key," +
"\"password\" varchar(20))");
Act("insert into \"User\" values ('aUser','whosOk')");
```

Run the program to recreate WebService.sqlite.

56. In the AngularJS project, delete the old WebService.sqlite, add the new one from the SQLiteSetup project s bin\Debug folder, and fix the Copy to Output property to be Copy always.

57. Let s tell our application about this class. In Solution Explorer, right-click Models and Add>Class.. User.cs and change its contents to:

```
using System.Net;
using AWebSvr;

namespace AngularJS.Models
{
    public class User
    {
        public string userName;
        public string password;

        public bool Authenticated()
        {
            var us = Connect.conn.Find<User>("\"username\"='" + userName + "' and
\"password\"='" + password +"'");
            return us!=null && us.Length == 1;
        }
        public static bool Authenticated(HttpListenerContext cx)
        {
            var b = cx.User.Identity as HttpListenerBasicIdentity;
            var u = new User();
            u.userName = b.Name;
            u.password = b.Password;
            return u.Authenticated();
        }
    }
   }
```

58. Now change Program.cs to use Basic Http authentication and add an override to use our new authentication method by default:

```
using AWebSvr;
using System.Net;

namespace AngularJS
{
    using Models;
    using Controllers;
    class Program : WebSvr
    {
        static void Main(string[] args)
        {
            Add(new StudentController());
            Add(new SubjectController());
            new Connect("DataSource=WebService.sqlite;Version=3;");
            new Program().Server(AuthenticationSchemes.Basic, "http://+:8088/MyApp/");
            Connect.conn.Close();
        }
        public override bool Authenticated()
        {
            return base.Authenticated() || User.Authenticated(context);
        }
    }
}
```

This option will make the browser pop up a window to collect Name and Password from the user. The browser does this because the client side code has not supplied user credentials. (For our application Basic authentication is fine as we are still on localhost, and we will surely use https for the production version.)

59. Check this works. You should be able to access the application with the User information we entered in step 55, and check that other users are not allowed to access the site. (You need to close the browser to change identity.)

There are alternatives to Basic Authentication. (As an exercise, come back to here and use IntegratedWindowsAuthentication instead.)

An authentication service

Let s now allow users to register themselves on the site.

60. Let s add a method to User.cs to register a user:

```
public void Register()
{
    Connect.conn.Act("insert into \"User\" values ('"+userName+"','"+password+"')");
}
```

61. We now need an authentication service. In Solution Explorer, right-click Controllers and Add>Class..>UserController.cs, and change it to:

```
using AWebSvr;

namespace AngularJS.Controllers
{
    using Models;
    public class UserController : WebCtlr
    {
        public static string GetUser(WebSvc ws, Document d)
```

```
        {
            var u = d.Extract<User>();
            if (u != null && u.Length == 1 && u[0].Authenticated())
                return "OK";
            throw new System.Exception("Not authenticated");
        }
        public static string PostUser(WebSvc ws, Document d)
        {
            var u = d.Extract<User>();
            if (u!=null && u.Length==1)
                u[0].Register();
            return "Done";
        }
        public override bool AllowAnonymous()
        {
            return true;
        }
    }
}
```

The override here is so that the Register button is allowed for unauthenticated users!

62. Add at the start of Main in Program.cs

```
            Add(new UserController());
```

63. We need a more sophisticated login page. Change the contents of Login.htm to read:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Login Page</title>
<script src="angular.js"></script>
<script src="mainApp.js"></script>
<script src="loginController.js"></script>
</head>
<body>
    <div ng-app="mainApp" ng-controller="loginController">
        <table border="0">
            <tr><td>User name:</td><td><input type="text" ng-
model="user.userName"></td></tr>
            <tr><td>Password:</td><td><input type="password" ng-
model="user.password"></td></tr>
            <tr>
                <td>
                    <input type="button" ng-click="register()" value="Register" />
                </td>
                </tr>
            </table>
        {{loginStatus}}
        </div>
</body>
</html>
```

Client-side authentication control

64. The above login page needs a loginController to work. In Solution Explorer right-click Scripts and Add>New Item..>Web>JavaScript file called loginController.js. Change its contents to read:

```
mainApp.controller("loginController", function ($scope, $http) {
    $scope.loginStatus = 'Your credentials are not recognised';
    $scope.register = function () {
```

```
        var url = 'http://localhost:8088/MyApp/User/';
        $http.post(url, $scope.user).success(function (response) {
            $scope.user = response;
            $scope.loginStatus = 'Registration is successful! Reload to continue.';
            $scope.$digest();
        });
    };
    });
```

Don t forget to change its properties to Copy if newer.

65. Try out this new machinery. Register new users and add some subject information for them. (The browser will ask for credentials on the next round trip after registering.)

MultiThreading

66. As part of the scalability development, it is quite possible that we should implement multi-threading in our web server. Up to now, the web clients have all shared the one thread, so that if a client takes a long time to serve, other clients need to wait. In fact LocalDB (and Pyrrho), despite being embedded database engines, will support multithreading, so it is reasonable to use a multithreaded web server. This means that each client request spawns a new service thread with its own connection to the local database.

    AWebSvr can handle multithreading too, so there are two stages in moving to multi-threading. The first is to make the database connection (potentially) different for each request. The second is to make the service thread different for each request (with its own database connection).

    The first step is to *remove* the following lines in Connect,cs:

```
    public class Connect
    {
        public static Connect conn;
        public SQLiteConnection sqlc;
        public SQLiteTransaction tr = null;
        public Connect(string cs)
        {
            sqlc = new SQLiteConnection(cs);
            sqlc.Open();
            conn = this;
           }
```

67. We now break the Program.cs class in two. Around the Main method, change the contents to look like:

```
using AWebSvr;
using System.Net;

namespace AngularJS
{
    using Models;
    using Controllers;
    class Program : WebSvr
    {
        static void Main(string[] args)
        {
            Add(new StudentController());
            Add(new SubjectController());
            Add(new UserController());
            new Program().Server(AuthenticationSchemes.Basic, "http://+:8088/MyApp/");
        }
        public override WebSvc Factory()
        {
```

```
            return new MySvc();
        }
    }
    public class MySvc: WebSvc
    {
        public Connect conn;
        public override void Open(HttpListenerContext cx)
        {
            conn = new Connect("DataSource=WebService.sqlite;Version=3;");
        }
        public override bool Authenticated()
        {
            return base.Authenticated() || User.Authenticated(context);
        }
   }
}
```

(We will change the last method here in the very next step 71.) This works because of the following code in WebSvr.cs

```
var ws = Factory();
ws._Open(listener);
if (ws == this)
    Serve();
else
  new Thread(new ThreadStart(ws.Serve)).Start();
```

so that because we override the Factory method, the WebSvr now make a new thread to handle each request. We have also carefully written our Models code so that Model instances are not shared between the resulting threads.

68. Next, make the new service class MySvc a parameter in each of the methods in each model and each controller, and change Connect.conn to ws.conn, e.g. in Student.cs :

```
        public static Student Find(MySvc ws,string fn, string sn)
        {
            var r = ws.conn.Find<Student>("\"firstName\"='" + fn +
"' and \"lastName\"='" + sn + "'");
            if (r == null || r.Length == 0)
                return null;
            r[0].subjects = ws.conn.Find<Subject>("\"student\"=" + r[0].Id);
            return r[0];
        }
```

and similarly for the other Find; in the methods in User.cs, and in the controllers we need things like:

```
    class UserController
    {
        public static string PostUser(MySvc ws, Document d)
        {
            if (u!=null && u.Length==1)
                u[0].Register(ws);
            return "Done";
        }
    }
```

In Program.cs the Authenticated() override becomes

```
        public override bool Authenticated()
        {
            return base.Authenticated() || User.Authenticated(this,context);
         }
```

69. Work through these changes until everything compiles again without errors. If you get stuck, look at the zip file for this stage of the tutorial. Check the application still works.

Connect pooling

70. The effect of the last set of changes is that a new Connect is created for each service call. This is a bit wasteful. Assuming that the database engine is implemented properly, we should be able to re-use the Connect once the service request is over. Since all the Connects have the same connection string, there is an obvious saving in processing and temporary storage. So let us implement a ConnectPool. In Program.cs add using System.Collections.Generic; at the top and modify the first few lines of class MySvc to:

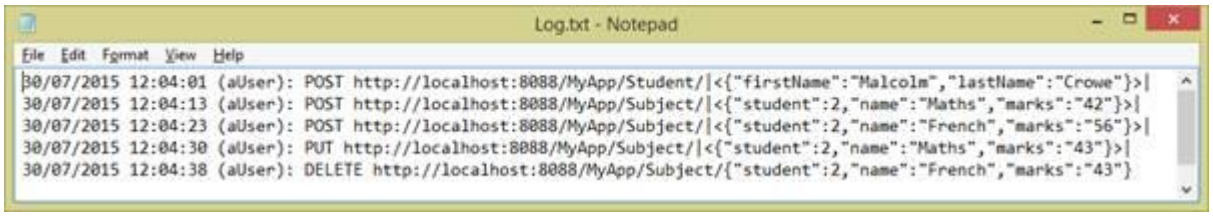
```
        static Stack<Connect> connectPool = new Stack<Connect>();
        public Connect conn;

        public override void Open(HttpListenerContext cx)
        {
            lock (connectPool)
            {
                if (connectPool.Count > 0)
                    conn = connectPool.Pop();
                else
                     conn = new Connect("DataSource=WebService.sqlite;Version=3;");
            }
        }
        public override void Close()
        {
            connectPool.Push(conn);
            base.Close();
        }
```

A Transaction Log

AWebSvr has some support for a transaction log. The default behaviour is that the existence of a file called Log.txt triggers logging.

71. To log all REST events in your application, use Solution Explorer to add a text file called Log.txt to your project, and ensure as usual that it is copied to the output folder: I recommend changing the Copy to Output property to Copy always. Each log entry has a timestamp, the user Name, the verb, the Url and the posted data for each request.

Here is an example of the Log.txt file generated automatically by AWebSvr (remember to look for it in the debug folder):



This actually highlights some untidy code in the sample: the last step shown will delete the French subject even though the mark is not 43!

Detecting Transaction Conflicts

Finally, let us consider the issue of transaction conflicts between users of this system. We began to provide some support for transactions at step 44. However, suppose that two users of the web application are making updates to the same student. Our application is not too bad at this, since on each update we retrieve the list of all of the subjects. Consider the following scenario:

(1) Teachers A and B are both working on student S, so they have the same subject information on their screen: Maths 42

(2) Teacher A adds a mark for French 56, and now sees Maths 42, French 56. But Teacher B still sees only Maths.

(3) Teacher B adds a mark for English 71, and now sees Maths 42, French 56 and English 71, while teacher A does not see the English mark.

Possibly this is fine in this application. Teacher B will think oh, someone has just added a mark for French . But we can imagine situations where there would be irritation and incomprehension: Administrator C deletes the student meantime, or both teachers have different marks for the same subject.

One solution to this problem would be for the clients to poll the server every few seconds to check for the current state of the student whose details are on the screen. If we don t do this, we should avoid making changes before discovering the changes made by someone else. Good practice is to use row versioning as described in Laiho and Laux s paper at www.dbtechnet.org/papers/RVV_Paper.pdf .

72. Since the only changes to our database are made by our controllers we could implement our own row versioning as follows. In another Visual Studio open SQLiteSetup.sln and make the following changes:

```
            Act("create table RVV(SEQ integer)");
            Act("insert into RVV values(0)");
            Act("create table \"Student\"(\"Id\" integer primary key,"+
    "\"firstName\" varchar(30),\"lastName\" varchar(30), VERSION integer default 0)");
```

Run the program. In the AngularJS project delete WebService.sqlite, and add it back from SQLiteSetup\bin\Debug. Remember to chang the Copy to output property to Copy always.

73. Whenever a change is made to data about a student we want to update this SEQ and assign it to the student. So let us add the VERSION field to Student.cs and create a method to do the update in Student.cs as follows:

```
public long VERSION;
public static void Update(MySvc ws, long id, long version)
{
    var cm = ws.conn.sqlc.CreateCommand();
    cm.Transaction = ws.conn.tr;
    cm.CommandText = "update RVV set SEQ=SEQ+1";
    cm.ExecuteNonQuery();
    var rv = (long)ws.conn.GetScalar("select SEQ from RVV");
    cm.CommandText = "update [Student] set VERSION="+rv+" where [Id]=" + id +
        " and VERSION="+version;
    if (cm.ExecuteNonQuery()!=1)
        throw new System.Exception("Data has changed unexpectedly");
}
```

Thus we will detect an error when someone else has already updated the student we are working on.

74. In studentController.js, change the first few lines to:

```
mainApp.controller("studentController", function ($scope, $http) {
    $scope.student = {};
    $scope.edit = {};
```

```
    $scope.error = '';
    $scope.setStudent = function (r) {
        $scope.student = r;
        $scope.fullName = $scope.student.firstName + " " + $scope.student.lastName;
        $scope.edit.student = $scope.student.Id;
        $scope.edit.version = $scope.student.VERSION;
        $scope.$digest();
     });
 };
```

This temporarily adds the *student s* VERSION field as a field called version in the document we will be sending when we update *subjects*.

75. Add a field **version** of type long to our Subject.cs class, so that the middleware knows what version of the student we have been sent for this request. Add the [Exclude] attribute since the database table hasn t got this field:

```
    [Exclude]
    public long version;
```

You will need using AWebSvr; at the top of this file. (It is deliberate to use VERSION for the student table and version for the Subject class: they are not the same.)

76. We need to make sure our Post method in Connect.cs pays attention to the Exclude attribute

```
            foreach (var f in tp.GetFields())
            {
                if (f.GetCustomAttributes(false).Length != 0) // [Exclude]
                    continue;
                var v = f.GetValue(ob);
                if (v != null)
                {
                    sc.Append(cm + "[" + f.Name + "]");
                    if (f.FieldType == typeof(string))
                        v = "'" + v + "'";
                    sv.Append(cm); sv.Append(v);
                    cm = ",";
                }
            }
```

77. In SubjectController call our Student.Update method each time before the call to Find:

```
            Student.Update(ws, su.student,su.version);
            var s = Student.Find(ws, su.student);
            ws.conn.Commit();
            return new Document(s).ToString();
```

78. In studentController.js, make the following modification to addSubject, updateMark and delSubject to handle the error

```
    $scope.addSubject = function () {
        var url = 'http://localhost:8088/MyApp/Subject/';
        $http.post(url, $scope.edit).success(function (response) {
            $scope.setStudent(response);
        }).error(function (response) {
            $scope.error = response;
        });
       };
```

79. Let us display this message. In testAngularJS.htm,make these changes after the second </table>:

```
    <span ng-show="fullName!=''">
        <input type="button" ng-click="delStudent()" value="Delete">
        <input type="checkbox" ng-model="editing">Edit
    </span>
```

```
    <span ng-show="error!=''">{{error}}
        <input type="button" ng-click="confirm()" value="Continue" >
       </span>
```

80. In studentController.js add a confirm() function to restore normal service

```
$scope.confirm = function () {
    $scope.error = '';
    $scope.findStudent();
   };
```

81. Now try out the above scenario. I get

(1) 

(2) 

(3) 

If you find any bugs or omissions in this tutorial, please email me: Malcolm.Crowe@tawqt.com .

Appendix: The AWebSvr library

AWebSvr is an open source contribution by Malcolm Crowe and University of the West of Scotland. It is a simple platform-independent customisable RESTful web server designed for use with JavaScript clients. It is for the .NET framework: Platform independence means that it does not depend on any particular DBMS. The web server and interfaces are designed for scalability. For best results, use an in-process (embedded) DBMS at first, because the web server and DBMS are just for a single web application. As the project scales up and out, the database will become distributed and partitioned, so you really also want this capability in your DBMS.

According to this approach, multi-tenancy, server scripting, entity frameworks and middleware data models are deprecated. In the accompanying tutorial the DBMS drivers are included for several DBMS including SQLServer.

The REST URl design is illustrated by the following two examples (both http and https are permitted):

http://*host*:*port*/*AppName*/*ControllerName*/{*JSON*}

http://*host*:*port*/*AppName*/*ControllerName*/[*param*{/*param*}]

where in the first example the curly brackets are part of JSON syntax, and in the second they indicate recurrence. By default, Controller names have the suffix Controller (e.g. *MyOwn*Controller), and the rest of the name is used in method names Get*MyOwn*, Put*MyOwn*, Post*MyOwn*, and Delete*MyOwn*.

Class	SubClass Of	Description
DocArray	DocBase	Handles arrays in JSON/BSON serialisation
DocBase		Common processing for JSON/BSON parsing
Document	DocBase	Handles JSON/BSON serialisation and class extraction
DocumentException	Systyem.Exception	For trapping document and serialisation errors
*ObjectId**		*Handles the ObjectId type in MongoDb-style documents*
WebCtlr		Base class for authenticated services
WebSvc		Base class for a custom web service
WebSvr	WebSvc	Base class for a custom web server

*ObjectId is not documented here.

DocArray

DocArray ()	Constructor
DocArray(string s)	Constructor: a document array deserialised from JSON
C[] Extract<C>(params string[] p)	Traverses the fields of this Document recursively for each path starting with p, and returns any objects of class C found.
List<object> items	The items in the document array. This is an indexer for the DocArray.
byte[] ToBytes()	A BSON version of the document array
string ToString()	A JSON version of the document array (Inherited from Object).

Document

There are two sorts of controller methods supported by this library: For any complex data, use a Document/ parameter for JSON serialisation. Simple data (e.g. a key) can be supplied as a sequence of parameters. The first parameter is reserved for the Service.

Document()	Constructor
Document(byte[] b)	Constructor: a document deserialised from BSON
Document(object ob)	Constructor: a document from a given object, representing its public fields.
Document(string s)	Constructor: a document deserialised from JSON
bool Contains(string k)	Returns true if k is the name of a field in the document.
C[] Extract<C>(params string[] p)	Traverses the items of this DocArray recursively for each path starting with p, and returns any objects of class C found.
List<KeyValuePair<string,object>> fields	The fields in the document (name-value pairs). This is an indexer for the Document.
byte[] ToBytes()	A BSON version of the document
string ToString()	A JSON version of the document (Inherited from Object).

DocumentException

string Message	(Inherited from System.Exception)

ExcludeAttribute

This is a custom attribute to tell AWebSvr not to send a field of a model class to the corresponding database table. Reflection is used (a) to collect fields from the database table to a model class and (b) to collect fields from JavaScript documents. The model class should always have all of the table columns but may have additional public fields for client-side communication. [Exclude] has no properties or methods and is simply placed in front of the field declaration in the model class. In the DBMS driver for POST or PUT no attempt should be made to supply values for excluded fields.

WebCtlr

Derived classes (e.g. XXController) should provide one of more the standard HTTP methods GetXX, PutXX, PostXX, DeleteXX according to one or both of the following templates:

public static string *VERBXX*(WebSvc ws,Document d)

public static string *VERBXX*(WebSvc ws,params object data)

virtual bool AllowAnonymous()	Can be overridden by a subclass. The default implementation returns false, but anonymous logins are always allowed if no login page is supplied (Pages/Login.htm or Pages/Login.html).

WebSvc

Your custom web server/service instance(s) will indirectly be subclasses of this class, so will have access to its protected fields and methods documented here. Controllers should be added in a static method, e.g. in Main()

Derived classes typically organise a connection to the DBMS being used. The connection can be for the service or for the request, and so should be set up in an override of the Open method.

static void Add(WebCtlr wc)	Install a controller for the service.
virtual bool Authenticated()	Override this to discriminate between users. By default the request will be allowed to proceed if AllowAnonymous

	is set on the controller or there is no login page. Get user identities etc from the context.
virtual void Close()	Can be overridden to release request-specific resources.
System.Net.HttpListenerContext context	Gives access to the current request details.
static System.Collections.Generic.Dictionary <string,WebCtlr> controllers	The controllers defined for the service.
string GetData()	Extracts the HTTP data supplied with the request: a URL component beginning with { will be converted to a Document.
virtual void Log(string verb, System.Uri u, string postData)	Write a log entry for the current controller method. The default implementation appends this information to Log.txt together with the user identity and timestamp.
virtual void Open (System.Net.HttpListenerContext cx)	Can be overridden by a subclass, e.g. to choose a database connection for the current request. The default implementation does nothing.
Serve()	*Calls the requested method using the above templates. Don t call this method directly.*

WebSvr

Your custom web server should be a subclass of WebSvr, and WebSvr is a subclass of WebSvc. It defines the URL prefixes (including hostnames and port numbers) for the service. If your service is multi-threaded, you can override the Factory method to returning a new instance of your WebSvc subclass. Finally, call either of the two Server methods to start the service loop.

virtual WebSvc Factory ()	Can be overridden by a subclass to create a new service instance. The default implementation returns this (for a single-threaded server).
void Server(params string[] prefixes)	Starts the server listening of a set of HTTP prefixes (up to the appName), with anonymous authentication.
void Server(System.Net.AuthenticationSchemes au, params string[] prefixes)	Starts the server listening of a set of HTTP prefixes (up to the appName), with the given authentication scheme(s).