Angular2 Tutorial (Python, 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 scalable web application architecture. This means 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. As a novelty, this tutorial will use Python 3.4 as the implementation language. Although it is written as a scripting language, compilers exist to speed up execution.

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 called AWebSvr. Extract the AWebSvrPy.zip 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: the advantage of SQLite is that it can be an embedded database, and the embedded version is installed along with Python, so we will use that.

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 Overiew on tutorialspoint. The code we will study is here.

Getting Started

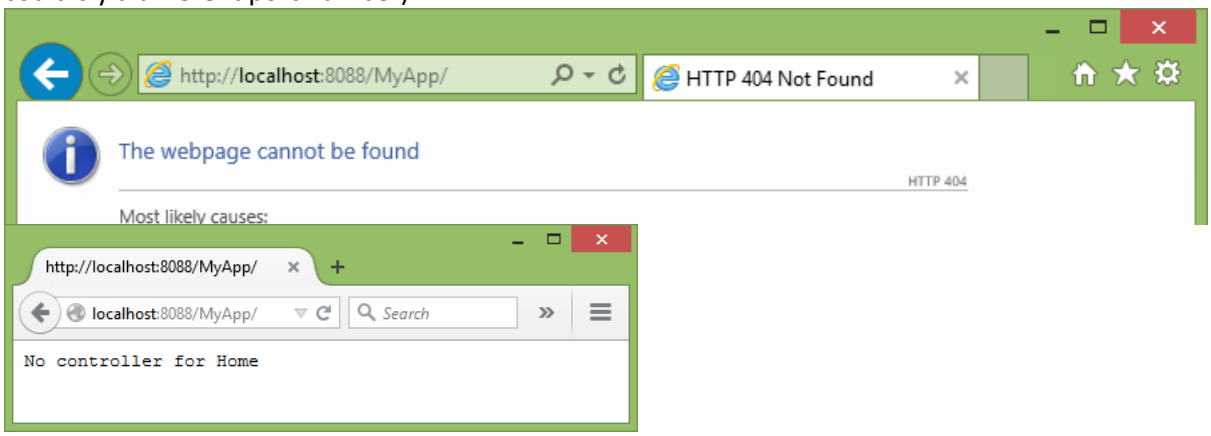
1. This tutorial assumes you have installed Python 3.4 and Visual Studio Tools for Python. You can add these tools to any edition of Visual Studio – her we assume Visual Studio 2015. It also assumes you have downloaded AWebSvrPy.zip (e.g. from TAWQT.com) and extracted AWebSvr.py somewhere.
2. Start up Visual Studio as Administrator, and create a new Python Application project called Angular3. (File>New Project..>Templates>Python>Python Application, change the location to somewhere you control – not your network drive, and give the project name as Angular3. Click OK).
3. In Solution Explorer, right-click Angular3 and select Add Existing Item Browse.. to the folder where you extracted AWebSbrPy, select AWebSvr.py, and click OK.
4. Replace the entire contents of Angular3.py with:

```
from AWebSvr import *

class Angular3(WebSvr):
    pass

Angular3().server('localhost',8088)
```

5. 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 Edge it's 400); 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).



6. 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 sometimes lead to different behaviour.)

Add AngularJS support

7. In Solution Explorer, right-click the project and select Add> New folder, to add a folder called Scripts.
8. Download angular.js from AngularJS (there is no need to run it).
9. 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.

Add the AngularJS sample

10. In Solution Explorer, right-click the project and select Add 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. r (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.)

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. In Solution Explorer, double-click testAngularJS.htm and change the first few lines to read:

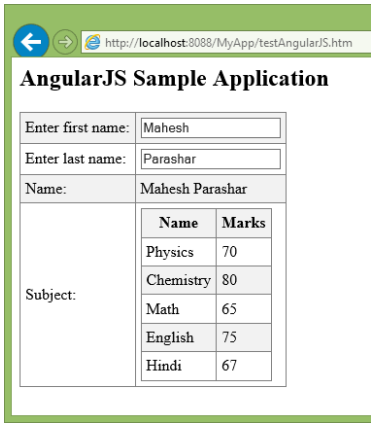
```
<html>

<head>
    <title>Angular JS Modules</title>
    <script src="angular.js"></script>
    <script src="mainApp.js"></script>
    <script src="studentController.js"></script>


        <style>
```

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.) If your screen looks different from this then maybe you need to download the files again!



Adding Database Support

15. Let's build the database first. In Solution Explorer>HaikusSample, Right-click the project, select Add>New Item..> Empty Python file, Change the Name to Setup.py and click Add.
16. Change the contents of Setup.py to:

```
import sqlite3
from builtins import print
conn = sqlite3.connect('WebService.db')
conn.execute("create table RVV(SEQ integer)");
conn.execute("insert into RVV values(0)");
conn.execute("create table \"Student\"(\"Id\" integer primary key,\"firstName\""+
    " varchar(30),\"lastName\" varchar(30), VERSION integer default 0)");
conn.execute("insert into \"Student\" (\"firstName\",\"lastName\")"+
             " values('Mahesh','Parashar')");
conn.execute("create table \"Subject\"(\"student\" references \"Student\","+
             "\"name\" varchar(20),\"marks\" integer)");
conn.execute("insert into \"Subject\" values (1,'Physics',70)");
conn.execute("insert into \"Subject\" values (1,'Chemistry',80)");
conn.execute("insert into \"Subject\" values (1,'Math',65)");
conn.execute("insert into \"Subject\" values (1,'English',75)");
conn.execute("insert into \"Subject\" values (1,'Hindi',67)");
conn.execute("create table \"User\"(\"userName\" varchar(20) primary key," +
"\"password\" varchar(20))");
conn.execute("insert into \"User\" values ('aUser','whosOk')");
conn.commit()
print('Done')
```

There are some aspects here whose purpose will not be clear at this stage: RVV, VERSION and User. We will come to them later.

17. Click Start (the green arrow in the toolbar). Then in the FileManager verify that WebService.db has been created in the project folder (it is

about 23KB). Close the application window.

The server-side Model

18. Use Solution Explorer to add a empty Python file using New Item.. to the project called Models.py.
19. Add the following code to Models.py:

```
class Name:
    def __init__(self):
        self.firstName = ''
        self.lastName = ''
        self.VERSION = 0
class Student (Name):
       def __init__(self):
              self.Id = 0
              self.subjects = list()
              super().__init__()
class Subject:
    def __init__(self):
        self.student = 0
        self.name = ''
        self.marks = 0
```

The Database Connection

20. Modify Angular3.py to open the database. Set Angular3.py to be the Startup File.

```
from AWebSvr import *
import sqlite3
class Angular3(WebSvr):
    pass
Angular3().connect(sqlite3.connect('WebService.db')).server('localhost',8088)
```

First steps with CRUD support in the Model

21. Before we start to build database support into our Models.py file, let us develop a class to help retrieve strongly-typed data from the database. In Solution Explorer, Add>New Item..>Empty Python File, given the Name as Reflection.py, and click Add. Change its contents to:

```
from sqlite3 import *
class Reflect:
    def stringify(ob):
        if isinstance(ob,list):
            r = '['
            c = ''
            for a in ob:
                r += c+Reflect.stringify(a)
                c = ','
            return r+']'
        r = '{'
        c = ''
        for f in dir(ob):
            if f[0]=='_':
                continue
            v = getattr(ob,f)
            if v==None:
                continue
            if isinstance(v,str):
                v = '"'+v+'"'
            elif isinstance(v,list):
                v = Reflect.stringify(v)
            else:
                v = repr(v)
            r += c+'"'+f+'":'+v
            c = ','
```

```
        return r+'}'
    def findOne(con,cls,cond=None):
        s = 'select * from "'+cls.__name__+'"'
        if cond!=None:
            s += ' where '+cond
        try:
            c = con.cursor()
            x = c.execute(s)
            a = c.fetchone()
            e = cls()
            for i in range(len(a)):
                f = x.description[i][0]
                setattr(e,f,a[i])
            c.close()
            return e
        except DatabaseError as e:
            return repr(e)
        except Exception as e:
                return repr(e)
    def findAll(con,cls,cond=None,num=1000):
        s = 'select * from "'+cls.__name__+'"'
        if cond!=None:
            s += ' where '+cond
        try:
            c = con.cursor()
            x = c.execute(s)
            r = list()
            for a in x:
                if len(r)>=num:
                    break
                e = object.__new__(cls)
                for i in range(len(a)):
                    f = x.description[i][0]
                    setattr(e,f,a[i])
                r.append(e)
            c.close()
            return r
        except DatabaseError as e:
            return repr(e)
        except Exception as e:
                return repr(e)
```

We will add some more methods to this class later.

22. Let's use Reflect.findOne to create Student given a name. In Models.py add

```
from Reflection import *
```

at the top, and to the Student class in Models.py add the following method:

```
    def _find(ws,fn,sn):
        s = Reflect.findOne(ws.conn,Student,"\"firstName\"='"+
                                fn+"' and \"lastName\"='"+sn+"'")
        s.subjects = Reflect.findAll(ws.conn,Subject,cond="\"student\"="+str(s.Id))
        return s
```

(Note that you need to remove the line break Word has added here!) The initial underscore is so that fields in the Models classes are the attributes without underscores.

Using AJAX to get data

23. 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 data for a single student has been hardwired. So let's change studentController.js to get it from the database instead:

```
mainApp.controller("studentController", function ($scope, $http) {
    $scope.student = {};
    $scope.findStudent = function () {
        $scope.fullName = '';
        $scope.student.subjects = [];
        var url = 'http://localhost:8088/MyApp/Student/'+
              $scope.student.lastName +'/'+  $scope.student.firstName;
        $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){..}) .

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

24. 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 lines containing Name:: and {{student.fullName()}} to read:

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

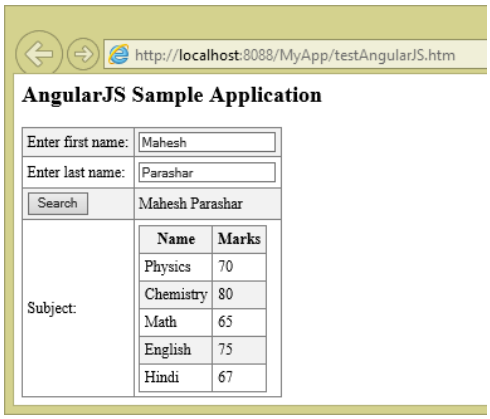
25. A server-side Controller

26. The server now needs to be able to handle the url in step 31. We need to create our first server-side Controller. In Solution Explorer, right-click the project and Add>New Item.. to create a new Python file, give the name as Controllers.py, and click Add.
27. Change the Controllers.py contents to

```
from AWebSvr import *
from Models import *
from Reflection import *
class StudentController(WebCtlr):
    def get(self,ws,ps):
        s = Student._find(ws,ps[1],ps[0])
        return Reflect.stringify(s)
```

28. We also need to get our WebSvr class to use this controller. In Angiular3.py just after import sqlite3, add

```
from Models import *
from Controllers import *
WebSvc.controllers['Student'] = StudentController()
```



29. Now run the program, and browse to the same address as before. Type the correct name, and click Search.
30. Close the application window and the browser. (The browser will probably cache our script files etc, so changes you make to the project might not be reflected in your next browsing session.)

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 add a method to post new data to the database. This will give an extra step or two.

31. 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>
```

32. Add an AJAX post method to studentController.js as follows)(before the final });):

```
$scope.addStudent = function () {
    $scope.student.subjects = [];
    var url = 'http://localhost:8088/MyApp/Student/';
    $http.post(url, $scope.student).success(function (response) {
        x = response;
        $scope.findStudent();
    });
    };
```

The useless looking x-=response here is to tell Angular that we have used the response string that has been passed to us.

33. Add an insert method to the class Reflect in Reflection.py:

```
    def insert(con,mod,doc):
        s = 'insert into "'+mod.__class__.__name__+'"('
        w = ') values ('
        c = ''
        for f in doc.fields:
            if not f[0] in dir(mod):
                continue
            m = getattr(mod,f[0])
            v = f[1]
            if v==None or m==None or isinstance(m,list):
                continue
            if isinstance(m,str):
                v = "'"+v.replace('\n','|')+"'"
            else:
                v = repr(v)
            s += c+'"'+f[0]+'"'
            w += c+v
            c = ','
        try:
            con.execute(s+w+')')
            con.commit()
            return "OK"
        except DatabaseError as e:
            return repr(e)
        except Exception as e:
                return repr(e)
```

34. Add a Post method to StudentController in Controllers.py:

```
    def post(self,ws,ps):
        return Reflect.insert(ws.conn,Student(),ps[0])
```

The AWebSvr adds the posted data (in Json) as a Document to the parameter list, and this passed to the insert method. Note that Document assignment and comparison is field-based. Whatever JSON has been provided from the client will be posted: the post will fail if the primary key constraint from step 17 is violated (and we haven't yet implemented error handling in the client-side controller to report on this).

35. 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.

Changing the Subjects information

36. 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.

37. We also will need methods in Reflection.py for update and delete:

```
    def update(con,mod,doc,key):
        s = 'update "'+mod.__class__.__name__ +'" set '
        w = ' where '
        c1 = ''
        c2 = ''
        for f in doc.fields:
            if not f[0] in dir(mod):
                continue
            v = f[1]
            if isinstance(v,str):
                v = '"'+v+'"'
            else:
                v = repr(v)
            a = '"'+f[0]+'"='+v
            if f[0] in key:
                w += c2+a
                c2 = ' and '
            else:
                s += c1+a
                c1 = ','
        try:
            con.execute(s+w)
            con.commit()
            return "OK"
        except DatabaseError as e:
            return repr(e)
        except Exception as e:
            return repr(e)
    def delete(con,mod,id,key):
        s = 'delete from "'+mod.__class__.__name__ +'" where'
        c = ''
        for i in range(len(id)):
            m = getattr(mod,key[i])
            v = id[i]
            if isinstance(m,str):
                v = '"'+v+'"'
            else:
                v = str(v)
            s += c+ '"'+key[i]+'"='+v
            c = ' and '
        try:
            con.execute(s)
            con.commit()
            return "OK"
        except DatabaseError as e:
            return repr(e)
        except Exception as e:
            return repr(e)
```

38. Our plan now is to add POST, DELETE and PUT for Subjects. We could get all of them to return the updated Student information but that would be quite a departure from normal REST semantics. We could get our client to update the subject list alongside the server calls, but for better assurance, let's redo findStudent() each time. Then studentController.js can become:

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

```
    $scope.edit = {};
    $scope.findStudent = function () {
        $scope.fullName = '';
        $scope.student.subjects = [];
        var url = 'http://localhost:8088/MyApp/Student/' +
              $scope.student.lastName +'/'+  $scope.student.firstName;
        $http.get(url).success(function (response) {
               $scope.student = response;
               $scope.fullName = $scope.student.firstName + " " +
       $scope.student.lastName;
               $scope.edit.firstName = $scope.student.firstName;
               $scope.edit.lastName = $scope.student.lastName;
               $scope.$digest();
        });
    };
    $scope.addStudent = function () {
      $scope.student.subjects = [];

        var url = 'http://localhost:8088/MyApp/Student/';
        $http.post(url, $scope.student).success(function (response) {
            x = response;
            $scope.findStudent();
        });
    };
    $scope.addSubject = function () {
        $scope.edit.student = $scope.student.Id;
        var url = 'http://localhost:8088/MyApp/Subject/';
        $http.post(url, $scope.edit).success(function (response) {
            x = response;
            $scope.findStudent();
        });
    };
    $scope.updateMark = function () {
        $scope.edit.student = $scope.student.Id;
        var url = 'http://localhost:8088/MyApp/Subject/' ;
        $http.put(url, $scope.edit).success(function (response) {
            x = response;
            $scope.findStudent();
        });
    };
    $scope.delSubject = function () {
        var url = 'http://localhost:8088/MyApp/Subject/' + $scope.student.Id +
            '/' + $scope.edit.name;
        $http.delete(url).success(function (response) {
            $scope.findStudent();
        });
    };
    $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 = '';
    };
});
```

39. In Solution Explorer, add the following class to Controllers.py:

```
class SubjectController(WebCtlr):
```

```
    def post(self,ws,ps):
        return Reflect.insert(ws.conn,Subject(),ps[0])
    def put(self,ws,ps):
        return Reflect.update(ws.conn,Subject(),ps[0],['student','name'])
    def delete(self,ws,ps):
        return Reflect.delete(ws.conn,Subject(),ps,['student','name'])
```

40. In Angular3.py add after the reference to StudentController:

```
WebSvc.controllers['Subject'] = SubjectController()
```

Handling DELETE Student

41. Add a Delete method to class StudentController in Controllers.py:

```
    def delete(self,ws,ps):
        return Reflect.delete(ws.conn,Student(),ps,['Id'])
```

42. All of this machinery should work now. Edit some subjects for your Student entry. Then close the application window and browser.

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.

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.

43. To activate the OSP/AWebSvr 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..>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>
```

The server-side authentication list

```
class User:
    def __init__(self):
        self.userName = ''
        self.password = ''
    def _authenticated(ws):
        us=Reflect.findOne(ws.conn,User,"userName='"+ws.user+"',password='"+
              ws.password+"'")
        return us!=None
```

44. Now change Angular3.py to use our new authentication method:

```
from OSPLink import *
from Models import *
from Controllers import *
WebSvc.controllers['Student'] = StudentController()
```

```
WebSvc.controllers['Subject'] = SubjectController()
class Angular3(WebSvr):
    def __init__(self):
        super().__init__()
    def authenticated(self):
        return super().authenticated() and User._authenticated(self)
Angular3().server('localhost',8088)
```

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.)

45. Check this works. You can also start a new browser to try out the aUser credentials we added in step 16.

An authentication service

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

46. Let's add a method to the User class in Models.py to register a user:

```
    def _register(self,ws):
        ws.conn.execute('insert into "User" values '+"('"+self.userName+
"','"+self.password+"')")
```

47. We now need an authentication service. In Controllers.py, add a new class:

```
class UserController(WebCtlr):
    def get(self,ws,ps):
        d = ps[0]
        u = d._extract(User)
        if u!=None and u._authenticated():
            return "OK"
        raise Exception("Not authenticated")
    def post(self,ws,ps):
        d = ps[0]
        u = d._extract(User)
        if u!=None:
            u._register()
        return "Done"
    def allowAnonymous(self):
        return True
```

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

48. Add in Angular3.py:

```
WebSvc.controllers['User'] = UserController()
```

49. 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

50. The above login page needs a loginController to work. In Solution Explorer right-click Scripts and Add>New Item..>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();
        });
    };
});
```

51. We also need to load this script in testAngularJS.htm, near the top:

```
...<script src="studentController.js"></script>
<script src="loginController.js"></script>
<style>...
```

52. Try out this new machinery. Register new users and add some subject information for them.

(1) 

If you find any bugs or omissions in this tutorial, please email me: Malcolm.Crowe@uws.ac.uk .