SENG365 Assignment 2, 2016: Poll website

Worth: 15% of final grade

Due: 11:55pm, Friday 3rd June

Drop dead: 11:55pm Friday 10th June, with 15% penalty.

Goals

This assignment aims to give you experience with building a website in which most of the functionality is in the JavaScript client, with the webserver providing just the initial download of the page and its scripts plus the database, which is to be accessed by RESTful web services.

1. Introduction

Your task is to provide a poll website that manages a set of *opinion polls*. Each poll has a question and a set of possible answers, exactly one of which must be selected by any user voting on a poll. To keep the task reasonably simple, we omit many features that would be essential in a real-world site, such as administrator login, account management, poll opening and closing dates. We're also omitting the ability to create and edit polls through the website; you will have to set up the polls by direct manipulation of the database, e.g. using *phpmyadmin.*

There is an Assignment 2 forum on Learn. You may ask questions about the assignment on that forum, and any updates/clarifications will be posted to that forum.

2. Database requirements

You need to record *polls* and *votes*. You must choose appropriate database representations for these.

In addition to the usual *id,* a poll has the following attributes:

- A short title.
- The question that the poll asks.
- A set of two or more answers, of which the voting user must select exactly one. There should not be a limit on the number of possible answers a poll may have.

A vote records how a particular IP number voted on a particular poll.

3. Functionality to be provided

You are required to implement the site as a Single Page Application (SPA), that is, using a JavaScripted client that manages all user interaction, using web services to access the database on the server. You should use *AngularJS*, plus *jQuery* if you need it. See section 6 for more information on the use of Angular in this assignment.

Your site must have a base URI like `http://csse-studweb3.canterbury.ac.nz/~usercode/365/polls` and should provide the following functionality.

1. Voting. A user (assumed in this case to be a normal unprivileged user) is presented with a list of all polls and can then choose to vote in any of them. The IP number of the machine from which the vote request comes is recorded in the vote. One would normally wish to ensure that a particular IP number only voted once on a given poll, but you shouldn't actually implement that, partly because it's not very instructive to do so but mostly because, once you'd implemented it, you (or we) wouldn't easily be able to test your site!
2. Viewing. The user (assumed now to be a poll administrator, though this not enforced) is shown a list of all polls in such a way that they can click any one of the polls to see the question, possible answers and a summary of all votes recorded so far.
3. A poll reset. A user (assumed to be a poll administrator) must be able to reset the poll, i.e., discard all the votes. [This is primarily so we can test your assignment.]
4. An *About this site* view that documents the site in some detail, as in the first assignment: the author's name; date; functionality that's known to be buggy or defective; code and data design; discussion of any other key design decisions; acknowledgements (including links) of any code

or ideas brought into your code from outside; any extra features you've implemented; a personal perspective of your achievement.

4. The RESTful web services

We have already used some simple JSON web services to fetch database information. In this assignment you will also need to update the database. Appendix 1 provides a more general introduction to RESTful web services than has been provided so far in the course.

Table 1 defines the services to be provided in this application. **You should return 404 (Not Found) for errors like a poorly-formed or non-existent ID or any other unspecified error conditions.** The greyed rows are not needed for the specified functionality but are included for completeness.

Method	URI	Request data	Response	Comments
GET	polls	-	200 OK and a JSON list of polls. Each poll is a JSON object with at least *id* and *title* attributes.	
GET	polls/id	-	200 OK and a single JSON object representing the requested poll, with at least *id*, *title*, *question* and *answers* attributes; *answers* is a list of strings.	id is an integer, e.g. polls/23, which is the ID of the required poll.
POST	polls	A JSON poll object in the same form as for GET.	201 (Created) + 'Location' header with link to new poll	Creates a poll. Not needed; included for completeness only.
PUT	polls/id	A JSON poll object in the same form as for GET.	200 OK if the PUT succeeds.	The given poll object replaces the existing poll object of the given id. Not needed; included for completeness only.
DELETE	polls/id	-	200 OK if the delete succeeds.	Not needed; included for completeness only.
POST	votes/pollId/vote	-	200 OK if the POST succeeds, i.e. the vote is valid. [Do not implement the code to prevent multiple votes from the same IP number.]	pollId and vote are integers, e.g. votes/23/3, representing the pollId and the selected answer number.
GET	votes/pollId	-	200 OK; response data is a JSON object containing at least a *votes* attribute which is a list of integers, being the number of votes for each answer.	
DELETE	votes/pollId	-	200 OK.	Deletes all votes in the given poll (essentially a reset of the poll).

Table 1: Specification of the RESTful services.

The base URI for all restful services in that table is of the form:

```
http://csse-studweb3.canterbury.ac.nz/~usercode/365/polls/index.php/services/
```

[If you wish to set up a .htaccess to remove the '/index.php' from all urls that's fine, too, but we'll ignore that possibility in what follows.]

For example, the URI for a GET of a poll with id 23 should be of the form:

```
http://csse-studweb3.canterbury.ac.nz/~rjl83/365/polls/index.php/services/polls/23
```

Where the table specifies that an object must have “at least” the following attributes you may add additional attributes if you wish but you must supply at least the ones specified.

6. Implementation Notes

A bit of care over the database schema (what tables? what columns?) could save a significant amount of time. Also, how many *models* do you need? [Hint: one model per table is almost certainly too many.]

Within your API you will need to check the request method (which is given by `$this->input->server('REQUEST_METHOD')`) within each controller method in order to determine what sub-action to take.

The *curl* command is useful for checking out web services from the command line. *man curl* to find out more.

Your use of Angular in the lab involved just a single view. This application requires at least three views, though one of these is just a single static page. Because we haven't had time to cover multiple views in labs, you are provided with a rudimentary two-view Angular application. Experiment with this before starting the assignment. See Appendix 2 for more information. Also, a tutorial chapter on the use of Angular routes is available from `https://docs.angularjs.org/tutorial/step_07`.

The base URI of your application, i.e. the URI from which a user loads the SPA, should be of the form

```
http://csse-studweb3.canterbury.ac.nz/~usercode/365/polls
```

See if you can obtain the counts of all the votes for a given poll in a single SQL query.

There is a pitfall to be avoided when setting up the voting form. You'll probably want to use *ng-repeat* over the set of answers, binding the different options back to some viewModel variable like *$scope.choice.* However, *ng-repeat* clones a new child scope for each iteration and if you just use *ng-model* to bind to the *choice* variable, you're binding to *choice* within the child scope, not the one you want, which is in the parent scope. So you either need to bind to *$parent.choice* or ensure that the choice variable is an attribute of an object (e.g. $scope.stuff.choice); the latter works because the child scope is only a shallow clone of the parent so that both the parent's and child's *stuff* attributes refer to the same object.

There is a Learn forum for Assignment 2, which will be used for discussions about this assignment. All official announcements relating to this assignment will be posted to that forum, so you are expected to keep in touch with it.

7. Marking

We don't wish to be too prescriptive in giving a marking breakdown (e.g. 73% for this, 27% for that) because it is always the case that a particularly good effort in one aspect can outweigh deficiences in another. However, as a guideline we envisage a mark breakdown something like the following:

- Web service functionality (tested separately): 30%
- Web site functionality: 30%
- Usability and appearance (you are expected to use some CSS in this assignment): 15%
- Design and code quality: 15%
- Quality of the “About this site” presentation: 10%

As in assignment 1, some percentage of these marks (somewhere between 5% and 10%) will be reserved for “going the extra mile”, i.e., some evidence of extra enthusiasm and dedication, such as additional functionality or a particularly nicely crafted site. You should draw our attention to any features of which you're particularly proud in your *About* page.

8. Submitting

First, make sure your database has at least 3 vaguely plausible polls in it, each with at least two answers and one with at least five. Then zip up your solution folder, naming it *polls-usercode.zip*, and submit it to the assignment 2 drop box on *Learn*.

PS: the Usual COSC Rule applies ...

Have fun!

Appendix 1: a brief introduction to RESTful web services

Web services of various sorts will be covered in lectures towards the end of the course, but this appendix should tell you enough for this assignment. Also, for a good tutorial on RESTful services, see http://www.restapitutorial.com/lessons/httpmethods.html

The term *web service* generally refers to a web-based service that is intended for use by an external computer program rather than by a human user. The *productListJSON* and *productDetailsJSON* programs in lab 4 provided simple web services and the services needed in this assignment are not much more complicated than that.

Client programs access RESTful web services using unadorned HTTP. The POST, GET, PUT and DELETE verbs are seen as operations on (virtual) collections of resources; they correspond to Create-Read-Update-Delete respectively in the CRUD pattern of database use. Files in a folder, or rows in a database table are simple examples of such resource collections. The meaning of the HTTP verb varies depending on whether the URI references a collection or an individual resource. Thus, for example, a GET of a collection should return a list of the resources in the collection whereas a GET of a resource returns the details of the resource itself.

Appendix 2: the skeleton 2-view Angular app

To run the supplied skeleton site, you must use a web server: the *route* module in Angular loads the separate view pages using XMLHttpRequests, which doesn't work if you open the home-page .html file directly in the browser.

Note the following:

1. When you run the app you'll see URLs like *http://localhost:8383/app/index.html#/items* and *http://localhost:8383/app/index.html#/items/1*. The bit of the URL from the '#' onwards is a *fragment identifier* (see https://en.wikipedia.org/wiki/Fragment_identifier). In its simplest form this can be used to select portions of a document but in JavaScript frameworks it's often used as a “sub-url”, evaluated by the framework to determine what view to present to the user just as the main bit of the url is used on the server. This allows bookmarking within the JavaScript app as well as normal forward/backward web-browser commands.
2. The *scripts* folder contains the main *angular.js* script plus a new *angular-route.js* module that provides routing services for multi-view sites.
3. The *index.html* is the site “home page”. The head portion of this loads the various scripts required but the body consists just of a *div* element annotated with the *ng-view* directive. This is a place-holder for the various views.
4. The *partials* folder contains the sub-view templates - two in this case. These are like the Angular views in the lab but without the DOCTYPE, head element etc. You can think of them as providing the *content* portions to a master template.
5. *app.js* defines the top-level Angular module ('itemsApp'), which is essentially a “container” for

the application. The second parameter to the constructor is a list of the names of other components needed by the module, namely the router ('ngRoute') and the separate controllers module ('itemsControllers'). Also in this file is the all-important route configuration. The call to the module's *config* method configures the '$routeProvider' service, telling it how to map from a given URL to a (view, controller) pair. Thus for example URLs of the form <baseUrl>#/items/<itemId> result in the item-detail view being loaded, with its associated *ItemDetailCtrl* controller being contructed with the *itemId* as a constructor parameter. [The ':' in the URL results in the rest of the url being bound to the name *itemId*].

6. *controllers.js* defines another module (“container”) into which are plugged two controllers: 'ItemListCtrl' for the list-of-items view and 'ItemDetailCtrl' for the detail view. Each is defined as in the lab except that the second has an extra service provided to it in the *controller* call: the *$routeParams* service which allows the router to pass parameters from the URL into the controller when it is constructed. Also in this module is a pseudo-database of items, which is accessed by the item-list controller to get the whole list and the item-detail controller to get details on any one item. [In your application you'll connect to an actual database with JSON services to get such information.]