Web Development

Server Side Development

Description

The first assignment focused on creating static HTML pages linked to one another using static hyperlinks. It allowed practicing rapid prototyping, page layout, navigation, and styling. The second assignment refactored the static Website into a dynamic *Single Page Application* (*SPA*) using *AngularJS*. Static HTML web pages were refactored into dynamic angular views rendering data provided by angular controllers. Angular services provided a central data access point used throughout out the Web application. Data was simulated as arrays of objects in angular services. This assignment focuses on creating Web services with the *Express* Node.js module. Web services will expose server side data sources through HTTP endpoints. The angular services will access the data from the Web services through various HTTP requests such as POST, GET, PUT, and DELETE.

Create Web Services

Create web services for each of the entities: *users*, *websites*, *pages*, *widgets*, *scripts*, and *statements*. These will allow creating scripts that execute when users interact with widgets, e.g., clicking on a button or link. For each the entities create Web services that implement CRUD operations, e.g., create users, update users, delete users. Migrate the object arrays implemented in the angular services over to the Web services. Make sure to use good programming practices discussed in class such as using IIFEs, declaring services, event handlers, variables at the top of a constructor, and implementing them towards, the bottom, using proper indentation, meaningful variable and function names, follow naming conventions, etc. Create a Node.js module called app.js that will serve as the entry point of the server side of the application. Load the module from the server.js file and initialization it. Using the Node.js module express, create the following Web service endpoints that respond to the URL pattern and HTTP method. Map the endpoints to the function listed and implement the logic in the function. Note that some of the URL patterns are ambiguous and will need to be resolved. Note that POST and PUT will need an object passed as argument containing data. Here is a list of all the server side files implementing the Web services of the assignment. They all provide CRUD operations on a data model they expose through HTTP endpoints. They are described further in the sections that follow.

```
/assignment/app.js
/assignment/services/user.service.server.js
/assignment/services/website.service.server.js
/assignment/services/page.service.server.js
/assignment/services/widget.service.server.js
```

Here's a snippet of code that can be used as an example for **`app.js`**

```
module.exports = function(app) {
    require("services/user.service.server.js")(app);
    require("services/website.service.server.js")(app);
    require("services/page.service.server.js")(app);
    require("services/widget.service.server.js")(app);
};
```

The **app.js** file must to be loaded and initialized from the **server.js** file. Towards the bottom of the **server.js** file require the **app.js** file and pass it the app express instance object as shown below

```
var express = require('express');
var app = express();
...
require("/assignment/app.js")(app);
app.listen(port, ipaddress);
```

Since the angular services will be sending JSON objects over the wire to the server, make sure Node.js is configured to parse JSON from the HTTP body using the body parser module. Use the snippet below as an example

```
var express = require('express');
var app = express();

// install, load, and configure body parser module
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Make sure the body parser module is listed as a dependency in the **package.json** file as shown below

```
...
  "dependencies": {
    ...
    "body-parser": "^1.15.0",
    ...
  },
...
```

User web services

Implement the user service in a file called **/assignment/services/user.service.server.js**

HTTP Method	URL Pattern	Function Name
POST	/api/user	createUser
GET	/api/user?username=username	findUserByUsername
GET	/api/user?username=username&password=password	findUserByCredentials
GET	/api/user/:userId	findUserById
PUT	/api/user/:userId	updateUser
DELETE	/api/user/:userId	deleteUser

Website Web Services

Implement the user service in a file called **`/assignment/services/website.service.server.js`**

HTTP Method	URL Pattern	Function Name
`POST`	`/api/user/:userId/website`	`createWebsite`
`GET`	`/api/user/:userId/website`	`findAllWebsitesForUser`
`GET`	`/api/website/:websiteId`	`findWebsiteById`
`PUT`	`/api/website/:websiteId`	`updateWebsite`
`DELETE`	`/api/website/:websiteId`	`deleteWebsite`

Page Web Services

Implement the user service in a file called **`/assignment/services/page.service.server.js`**

HTTP Method	URL Pattern	Function Name
`POST`	`/api/website/:websiteId/page`	`createPage`
`GET`	`/api/website/:websiteId/page`	`findAllPagesForWebsite`
`GET`	`/api/page/:pageId`	`findPageById`
`PUT`	`/api/page/:pageId`	`updatePage`
`DELETE`	`/api/page/:pageId`	`deletePage`

Widget Web Services

Implement the user service in a file called **`/assignment/services/widget.service.server.js`**

HTTP Method	URL Pattern	Function Name
`POST`	`/api/page/:pageId/widget`	`createWidget`
`GET`	`/api/page/:pageId/widget`	`findAllWidgetsForPage`
`GET`	`/api/widget/:widgetId`	`findWidgetById`
`PUT`	`/api/widget/:widgetId`	`updateWidget`
`DELETE`	`/api/widget/:widgetId`	`deleteWidget`

Refactor Angular Services To Use Web Services

The previous assignment implemented angular services as central access points for all things data. Services contained the data in the form of arrays of objects. The user service contained an array of user objects, the website service contained an array of website objects, and the page service contained an array of pages. In this assignment, move those arrays of objects from the angular services to the express Web services. Then, in the Web service functions, implement a similar logic to the one in the angular services. The Web services must parse the path and query parameters from the URL and use them to implement their logic. The angular services will need to be refactored to use the Web services. The angular services will send asynchronous HTTP requests to the Web services, and receive responses from the server. Promises will be used to handle the asynchronous communication.

Refactor Controllers to use Promises

The previous assignment implemented angular controllers to handle events from their respective views, invoke the appropriate angular service, and provide data to the views for rendering. Since the services were implemented to simulate the data on the browser, controllers could call service functions synchronously. In this assignment the services will be refactored to use asynchronous calls to the server instead. Now the angular services will not be returning the data right away. Instead they will be returning promises that allow registering callback functions for asynchronous client/server communication. Refactor the controllers to use the promises returned by the angular services.

Add Sorting to the Widget List View

The widget list view lists all widgets in a page. The developer needs to be able to reorder the widgets on the page. Create a directive called **`jga-sortable`** that uses jQuery and jQueryUI to implement the reordering behavior. Include a reference to jQuery and jQueryUI CDN. Implement the directive in a module called **`jgaDirectives`** in a file called **`/assignment/directives/jga- directives.js`**. Apply jQueryUI's sortable behavior to the widget container. The directive must modify the order of the widgets in the original array in the server. Use the following URL pattern for a Web service endpoint to modify the order of the widgets on the server:

PUT `/page/`**`:pageId`**`/widget?`**`initial`**`=`**`index1`**`&`**`final`**`=`**`index2`**

`pageId`	id of the page whose widgets are being displayed
`initial`	initial index of the widget before being reordered
`final`	final index of the widget after being reordered

Enhance the Image Widget

Currently the image widget allows setting a URL linking to an image on the Web. In this assignment add capabilities to upload images to the server and search for images from Flickr.

Add File Upload Support

Refactor the image widget to support file upload. Use a form element that contains a file input element that allow users to navigate to the image they wish to upload. The form must submit the image to a Web service mapped to **`/api/upload`**. The request must contain the id of the image widget so the image can be related to the image widget when rendering. The Web service must store the image in an upload directory in the public directory, e.g., **`/public/uploads`**. The uploaded file will have a random, unique name with no extension. Use the random name to create the image's URL for the widget's **`url`** field. Here's a snippet of code in **`widget-edit.view.client.html`** illustrating a form used to upload an image from the user's computer.

```
<form action="/api/upload"  method="post" enctype="multipart/form-data">
    <input  name="myFile"   type="file" class="form-control"/>
    <input  name="width"    value="{{model.widget.width}}" style="display: none"/>
    <input  name="widgetId" value="{{model.widget._id}}"   style="display: none"/>
    <button type="submit"   class="btn btn-block btn-primary">Upload Image</button>
</form>
```

The form above submits the image to a service endpoint implemented in **`widget.service.server.js`**

```
module.exports = function (app) {
    var multer = require('multer'); // npm install multer --save
    var upload = multer({ dest: __dirname+'/../../public/uploads' });

    app.post ("/api/upload", upload.single('myFile'), uploadImage);

    function uploadImage(req, res) {

        var widgetId      = req.body.widgetId;
        var width         = req.body.width;
        var myFile        = req.file;

        var originalname  = myFile.originalname; // file name on user's computer
        var filename      = myFile.filename;     // new file name in upload folder
        var path          = myFile.path;         // full path of uploaded file
        var destination   = myFile.destination;  // folder where file is saved to
        var size          = myFile.size;
        var mimetype      = myFile.mimetype;
    }
    ...
}
```

Add Flickr Image Search Support

Refactor the image widget to support searching images from ***Flickr***. Add a button labeled ***Search*** that navigates to an image search view implemented in a file called **`widget-flickr-search.view.client.html`**. The view provides a search box at the top of the screen as shown in the wireframes. When the user clicks on the search button, a request is made to flickr's search API. The API responds with a collection of image objects. Use the object to create a URL to a thumbnail of the image and render them as shown in the wireframes. When the user clicks an image save the URL of the original image as the url field of the new image widget. Here's a snippet of code that can be used in **`widget-flickr-search.view.client.html`** to implement the search text field and button

```
<div class="input-group">
    <input ng-model="searchText" type="text" class="form-control">
    <span class="input-group-btn">
        <a ng-click="model.searchPhotos(searchText)" class="btn btn-default"
type="button">
            <span class="glyphicon glyphicon-search"></span>
        </a>
    </span>
</div>
```

Implement an Angular Flickr Image Search Controller

In a file called **widget-flickr-search.controller.client.js** implement an angular controller called **Flickr ImageSearchController** that uses a **FlickrService** to search for images. The controller must implement an event handlers called **searchPhotos()** invoked when the user clicks on the search button in the view. The view passes the search text to the controller and the controller passes it to the service. The controller unpacks the response from the service and binds the images to a **photos** variable. The view renders the images as shown in the wireframes. The controller must implement an event handler called **selectPhotos()** that is invoked when an image is clicked. The view passes the image instance to the controller as an argument. The controller uses the image object to create a URL for the **url** field and uses the **WidgetService** to update the image instance on the server. Here's an example search input field with a search button

```
vm.searchPhotos = function(searchTerm) {
    FlickrService
        .searchPhotos(searchTerm)
        .then(function(response) {
            data = response.data.replace("jsonFlickrApi(","");
            data = data.substring(0,data.length - 1);
            data = JSON.parse(data);
            vm.photos = data.photos;
    });
}
```

In the **widget-flickr-search.view.client.html** render the results from the search. Here's an example of iterating over the photos. Each of the photos is bound to a click event to notify the controller of the selected photo.

```
<div class="row">
    <div class="col-xs-4" ng-repeat="photo in model.photos.photo">
        <img ng-click="model.selectPhoto(photo)"
             ng-src="https://farm{{photo.farm}}.staticflickr.com/
             {{photo.server}}/{{photo.id}}_{{photo.secret}}_s.jpg"/>
    </div>
</div>
```

The controller can use the **photo** instance to build the URL to the original image in Flickr. Use the **WidgetService** to update the image widget instance in the server. Here's a code snippet that can be used as an example

```
function selectPhoto(photo) {
    var url = "https://farm" + photo.farm + ".staticflickr.com/" + photo.server;
    url += "/" + photo.id + "_" + photo.secret + "_b.jpg";
    WidgetService
        .updateWidget(websiteId, pageId, widgetId, {url: url})
        .then(...);
}
```

Implement an Angular Flickr Image Search Service

In a **flickr.service.client.js** file implement an angular service called **FlickrService** implemented in a function of the same name. The service API must provide a function **searchPhotos** that sends the search request to the Flickr search API and returns a promise. When the promise resolves, the response will contain an array of JSON objects representing the images that match the query

```
...
var key = "your-flickr-key";
var secret = "your-flickr-secret";
var urlBase = "https://api.flickr.com/services/rest/?method=flickr.photos.search
                &format=json&api_key=API_KEY&text=TEXT";
...
function searchPhotos(searchTerm) {
    var url = urlBase.replace("API_KEY", key).replace("TEXT", searchTerm);
    return $http.get(url);
}
```

Implement the HTML Widget

The previous assignment implemented the rendering part of this widget in the widget list view. It used the angular **`$sce`** service to unescape a raw HTML string and display it as formatted HTML. This assignment will implement the editor that will allow creating new instances of the HTML widget that will allow developers to enter and format raw HTML as part of their page. Use the textangular directive to provide a WYSIWYG editor to format the HTML. Download and install the directive to local project and host it from your Node.js server. Copy the textangular JavaScript and CSS files into a js and css folders:

```
js/textAngular.min.js
js/textAngular-rangy.min.js
js/textAngular-sanitize.min.js
css/textAngular.css
```

Load the textangular JavaScript and CSS from the index.html file. The textangular editor is implemented as an angular module that needs to be declared as a dependency of the **`WebAppMakerApp`** module. Use the following snippet as an example:

```
(function () {
    angular.module ("WebAppMakerApp", ["ngRoute", "textAngular"]);
})();
```

In the **`widget-edit.view.client.html`** display the **`text-angular`** directive for the HTML widget, bind the HTML to the widget's **`text`** field, and configure the list of text-angular format buttons. Use the code snippet below as a guide.

```
<div ng-model="model.widget.text"
     text-angular
     ta-toolbar="[['h1','h2','h3'],['bold','italics','underline','strikeThrough'],
        ['ul','ol'],['justifyLeft','justifyCenter','justifyRight','justifyFull'],
        ['indent','outdent'],['html']]">
</div>
```

Alternatively implement the widget in a separate HTML document called **`widget-html-edit.view.client.html`** and include it from the **`widget-edit.view.client.html`** view.

Implement the Text Widget

Implement a brand new text widget that will allow developers to create forms for end users to fill out. In **widget-chooser.view.client.html** add a new **Text Input** link that will add a new widget of type **TEXT** and navigate to the widget editor view **widget-edit.view.client.html** showing the text widget editor. Here's a sample snippet that can be used to implement the editor

```
<div ng-switch-when="TEXT">
    Text
    <input ng-model="model.widget.text" class="form-control"/>
    Rows
    <input ng-model="model.widget.rows" class="form-control" type="number"/>
    Placeholder
    <input ng-model="model.widget.placeholder" class="form-control"/>
    Formatted
    <input ng-model="model.widget.formatted" type="checkbox"/>
</div>
```

Note the new widget properties: **rows** and **formatted**. When the text widget renders in the widget list view, it should render in one of three ways based on the values of **rows** and **formatted**. If the **formatted** property is true, then the widget should render as a **text-angular** input field. If the **formatted** property is false, then the widget can render either as a **input** element or a **textarea** element based on the value of the **rows** property. If the **rows** property is 1, then the widget should render as an **input** element. If the **rows** property is greater than 1, then the widget should render as a **textarea**. Here's an example of how to render the widget in the **widget-list.view.client.html**

```
<div ng-switch-when="TEXT">
    <div ng-if="widget.formatted" text-angular ta-toolbar="..."></div>
    <input ng-if="!widget.formatted && (!widget.rows || widget.rows===1)"
           placeholder="{{widget.textInput.placeholder}}" class="form-control"/>
    <textarea ng-if="!widget.formatted && (widget.rows > 1)"
              rows="{{widget.rows}}" placeholder="{{widget.placeholder}}"
              class="form-control">{{widget.text}}</textarea>
</div>
```

Deliverables

GitHub and OpenShift Deliverables

To allow TAs and instructor to see your changes, please frequently commit and push your work to GitHub and OpenShift repositories. Below is an example of the commands you will use. The example assumes your project is located in **~/summer2016/web-dev**:

```
> cd ~/summer2016/web-dev
> git add .
> git commit -m 'A comment describing your work'
> git push github
> git push openshift
```

Verify that the files have copied to the github repository. Also visit your OpenShift website and verify that your changes are reflected on the remote server.

Tagging a Release

When you consider your work complete and ready for evaluation (ready for release), go to your code repository in GitHub and generate a release by navigating to "releases". Then click on "Create a new release" and type the name of the tag in the input field labeled "Tag version". We will be using the following tags for the various assignments:

assignment3 (previous assignment)
assignment4 (this assignment)
assignment5 (next assignment)
assignment6 (last assignment)

If you need to resubmit the assignment then create a new tag by adding a version number, e.g.,

assignment4.1, assignment4.2, etc...

We will grade the very last release. The date/time you create the tag will be considered the date/time of submission. If you have questions on how to create tags or have any problem at all, please do not hesitate to give me a call at (978) 761-5742 and we can jump on a Google Hangout and I can walk you through the process.

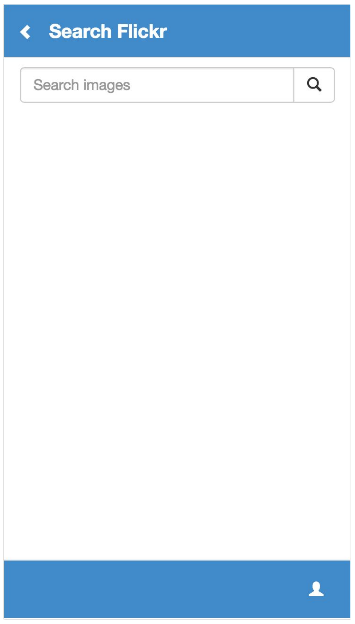
Blackboard Deliverables

Please submit the following in Blackboard

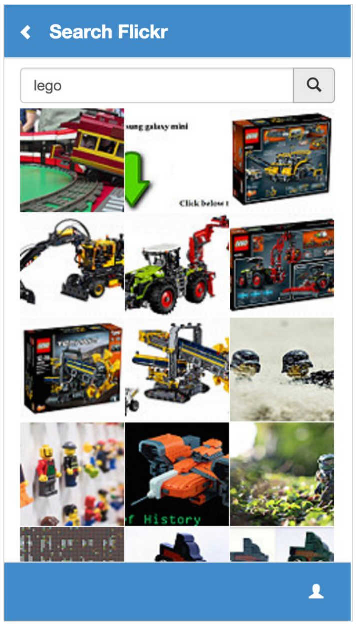
1. GitHub repository URL
2. OpenShift Website URL

Wireframes

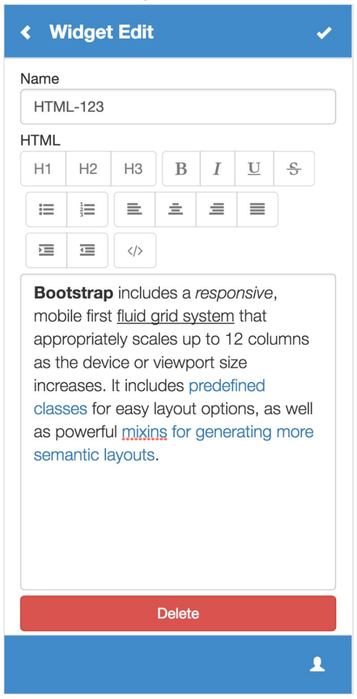
Search Flickr View



Search Results



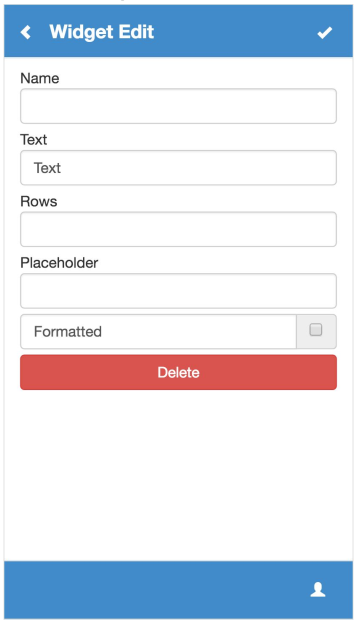
HTML Widget



Name: HTML-123

Bootstrap includes a *responsive*, mobile first fluid grid system that appropriately scales up to 12 columns as the device or viewport size increases. It includes predefined classes for easy layout options, as well as powerful mixins for generating more semantic layouts.

Text Widget



Text: Text