Web Development

Client Side Development

Description

This is the third of several assignments guiding you through the creation of a single page MEAN stack application. The skills you practice in the assignments are meant to be used on a project of your own. Do all your work in a directory called ***assignment*** off of the ***public*** directory. Commit your work frequently throughout the day and push your changes at the end of the day to deploy and restart your server. When you are ready to submit this assignment, ***tag*** your last commit you wish to be graded on as ***assignment3*** on GitHub. You can keep working and committing and pushing. The TAs and instructors can always view your assignment using the ***assignment3*** tag.

In the previous assignment you created a static set of pages that served as a prototype of the Website. The static pages allowed us to explore content related issues such as page layout, styling, navigation, and other presentation related issues. In this assignment you will extend the prototype by making the pages dynamic. Angular.js will be used to implement data services, controllers, and single page navigation.

Create an Angular Application

The first step will be to create an angular application. Angular applications are created by declaring an angular module and tying it to an HTML document. Do all your work in the **`assignment`** directory. Remember to use good programming practices such as IIFEs.

1. Create an **`app.js`** JavaScript file and declare an angular module called **`WebAppMaker`** with no dependencies
2. Create, or modify, the **`index.html`** page
    a. Declaring an angular application called **`WebAppMaker`**
    b. Link to a bootstrap CDN
    c. Load the angular core library from a CDN
    d. Load the **`app.js`** file

Later in this assignment we will add and configure routing dependencies.

Create Angular Views

In the previous assignment several static HTML pages were created. In this assignment pages will be refactored into angular views. Rename the files to capture the fact that they are *views* and are dynamically rendered on the *client* side using JavaScript on the browser.

1. Group the HTML files into folders **`user`**, **`website`**, **`page`**, and **`widget`**
    a. Move **`login`**, **`register`**, and **`profile`** pages into the **`user`** folder
    b. Move **`website-list`**, **`website-new`**, and **`website-edit`** pages into the **`website`** folder
    c. Move **`page`** pages into the **`page`** folder
    d. Move **`widget`** pages into the **`widget`** folder
2. Rename all the HTML files so that they all end with **`.view.client.html`**. For example **`page-list.html`** becomes **`page-list.view.client.html`**. These files will often be referred to as *templates* or *views*. This will break the hyperlinks and navigation will not work anymore. Do not fix this yet. Navigation will be refactored in a later step

3. Create a new folder called **views**. Move the **user**, **website**, **page**, and **widget** folders into the **views** folder
4. Refactor the HTML files into angular fragments. Remove the following tags: **html**, **head**, **meta**, **title**, **link**, **script**, **body**. Keep only the main content of the pages

These new files are not meant to be viewed independently. They are instead meant to be dynamically included as part of a *single page* that already provides these tags. The **index.html** file created earlier will serve as the single page container of the Website.

Configure Angular Routing

In the previous assignment we created several static web pages linked together using hyperlinks. In this assignment we will refactor navigation using Angular's routing module and implement a *single page application* or SPA. Refactor all hyperlinks in the views to use *hash fragments* instead of referring to actual files. For instance, change the hyperlink from the login page to the register page, **<a href="register.html">**, to use hash fragment: **<a href="#/register">**. Use the routes listed in the routing table below. Create a **config.js** file to configure the angular application routing and implement the navigation described in the page flow diagram. Use the routing provider to map the hash fragments to the views. For instance, the following snippet of code binds the **/login** route to its template **login.view.client.html**

```
(function() {
  angular
    .module("WebAppMaker")
    .config(Config);
  function Config($routeProvider) {
    $routeProvider
      .when("/login", {
            templateUrl: "/views/user/login.view.client.html"
      })
      ...
  }
})();
```

Use the example above to implement navigation as described in page flow diagram and the following routes

	Route	View
1.	/login, /, default	login.view.client.html
2.	/register	register.view.client.html
3.	/user/:uid	profile.view.client.html
4.	/user/:uid/website	website-list.view.client.html
5.	/user/:uid/website/new	website-new.view.client.html
6.	/user/:uid/website/:wid	website-edit.view.client.html
7.	/user/:uid/website/:wid/page	page-list.view.client.html
8.	/user/:uid/website/:wid/page/new	page-new.view.client.html
9.	/user/:uid/website/:wid/page/:pid	page-edit.view.client.html
10.	/user/:uid/website/:wid/page/:pid/widget	widget-list.view.client.html
11.	/user/:uid/website/:wid/page/:pid/widget/new	widget-chooser.view.client.html
12.	/user/:uid/website/:wid/page/:pid/widget/:wgid	widget-edit.view.client.html

Where **:uid**, **:wid**, **:pid**, and **:wgid** are path parameters encoding the IDs of particular users, websites, pages, and widgets.

In the **index.html** page

1. Load the angular routing module from a CDN (**angular-route.min.js**)
2. Load the configuration file **config.js**
3. Declare an angular routing view using the **ng-view** directive where all views will be dynamically included

Create Angular Services

Create angular services to provide a central place to access/update data. Create a separate service for each type of entity: **user**, **website**, **page**, and **widget**. Each service must provide CRUD operations to manipulate the corresponding entity: **create**, **read**, **update**, and **delete**. Create the following services with the listed CRUD operations. Implement all services in a **services** folder. Make sure to follow good practice such as using IIFEs (immediately invoked function expressions) and declaring APIs at the top of the service. Remember to load all new service files from the **index.html** page. Create the following service files

1. **assignment/services/user.service.client.js**
2. **assignment/services/website.service.client.js**
3. **assignment/services/page.service.client.js**
4. **assignment/services/widget.service.client.js**

UserService

Implement **UserService** in a file called **user.service.client.js**. Declare the service in a function of the same name. In the service declare a local array called **users** that will be used to simulate data from a database. The local **users** array is only temporary and will be removed in the next assignment where data will be fetched from the server. Use the following data to initialize the **users** array

```
[
   {_id: "123", username: "alice",    password: "alice",    firstName: "Alice",  lastName: "Wonder"  },
   {_id: "234", username: "bob",      password: "bob",      firstName: "Bob",    lastName: "Marley"  },
   {_id: "345", username: "charly",   password: "charly",   firstName: "Charly", lastName: "Garcia"  },
   {_id: "456", username: "jannunzi", password: "jannunzi", firstName: "Jose",   lastName: "Annunzi" }
]
```

Implement the following API in the **UserService** service

1. **createUser(user)** - adds the **user** parameter instance to the local **users** array
2. **findUserById(userId)** - returns the user in local **users** array whose **_id** matches the **userId** parameter
3. **findUserByUsername(username)** - returns the user in local **users** array whose **username** matches the parameter **username**
4. **findUserByCredentials(username, password)** - returns the user whose **username** and **password** match the **username** and **password** parameters
5. **updateUser(userId, user)** - updates the user in local **users** array whose **_id** matches the **userId** parameter
6. **deleteUser(userId)** - removes the user whose **_id** matches the **userId** parameter

Here's an example snippet of code that declares the **UserService** in **user.service.client.js** and a couple of API functions. Follow the same pattern for all other services

```
(function() {
    angular
        .module("WebAppMaker")
        .factory("UserService", UserService);
    function UserService() {
        var users = [ ... ];
        var api = {
            "createUser"   : "createUser",
            "findUserById" : "findUserById",
            ...
        };
        return api;
        function createUser(user) { ... }
        function findUserById(id) { ... }
        ...
    }
})();
```

WebsiteService

Implement **WebsiteService** in a file called **website.service.client.js**. Declare a service called **Website Service** implemented in a function of the same name. In the service, declare a local array called **websites** that will be used to simulate data from a database. The local **websites** array is only temporary and will be removed in the next assignment where data will be fetched from the server. Use the following data to initialize the **websites** array

```
[
  { "_id": "123", "name": "Facebook",    "developerId": "456" },
  { "_id": "234", "name": "Tweeter",     "developerId": "456" },
  { "_id": "456", "name": "Gizmodo",     "developerId": "456" },
  { "_id": "567", "name": "Tic Tac Toe", "developerId": "123" },
  { "_id": "678", "name": "Checkers",    "developerId": "123" },
  { "_id": "789", "name": "Chess",       "developerId": "234" }
]
```

Implement the following API in the **WebsiteService** service

1. **createWebsite(userId, website)** - adds the **website** parameter instance to the local **websites** array. The new website's **developerId** is set to the **userId** parameter
2. **findWebsitesByUser(userId)** - retrieves the websites in local **websites** array whose **developerId** matches the parameter **userId**
3. **findWebsiteById(websiteId)** - retrieves the website in local **websites** array whose **_id** matches the **websiteId** parameter
4. **updateWebsite(websiteId, website)** - updates the website in local **websites** array whose **_id** matches the **websiteId** parameter
5. **deleteWebsite(websiteId)** - removes the website from local **websites** array whose **_id** matches the **websiteId** parameter

PageService

Implement **PageService** in a file called **page.service.client.js**. Declare the service in a function of the same name. In the service declare a local array called **pages** that will be used to simulate data from a database. The local **pages** array is only temporary and will be removed in the next assignment where data will be fetched from the server. Use the following data to initialize the **pages** array

```
[
  { "_id": "321", "name": "Post 1", "websiteId": "456" },
  { "_id": "432", "name": "Post 2", "websiteId": "456" },
  { "_id": "543", "name": "Post 3", "websiteId": "456" }
]
```

Implement the following API in the **PageService** service

1. **createPage(websiteId, page)** - adds the **page** parameter instance to the local **pages** array. The new page's **websiteId** is set to the **websiteId** parameter
2. **findPageByWebsiteId(websiteId)** - retrieves the pages in local **pages** array whose **websiteId** matches the parameter **websiteId**
3. **findPageById(pageId)** - retrieves the page in local **pages** array whose **_id** matches the **pageId** parameter
4. **updatePage(pageId, page)** - updates the page in local **pages** array whose **_id** matches the **pageId** parameter
5. **deletePage(pageId)** - removes the page from local **pages** array whose **_id** matches the **pageId** parameter

WidgetService

Implement **WidgetService** in a file called **widget.service.client.js**. Declare the service in a function of the same name. In the service declare a local array called **widgets** that will be used to simulate data from a database. The local **widgets** array is only temporary and will be removed in the next assignment where data will be fetched from the server. Use the following data to initialize the **widgets** array

```
[
  { "_id": "123", "widgetType": "HEADER", "pageId": "321", "size": 2, "text": "GIZMODO"},
  { "_id": "234", "widgetType": "HEADER", "pageId": "321", "size": 4, "text": "Lorem ipsum"},
  { "_id": "345", "widgetType": "IMAGE", "pageId": "321", "width": "100%",
    "url": "http://lorempixel.com/400/200/"},
  { "_id": "456", "widgetType": "HTML", "pageId": "321", "text": "<p>Lorem ipsum</p>"},
  { "_id": "567", "widgetType": "HEADER", "pageId": "321", "size": 4, "text": "Lorem ipsum"},
  { "_id": "678", "widgetType": "YOUTUBE", "pageId": "321", "width": "100%",
    "url": "https://youtu.be/AM2Ivdi9c4E" },
  { "_id": "789", "widgetType": "HTML", "pageId": "321", "text": "<p>Lorem ipsum</p>"}
]
```

Implement the following API in the **WidgetService** service

1. **createWidget(pageId, widget)** - adds the **widget** parameter instance to the local **widgets** array. The new widget's **pageId** is set to the **pageId** parameter

2. **findWidgetsByPageId(pageId)** - retrieves the widgets in local **widgets** array whose **pageId** matches the parameter **pageId**
3. **findWidgetById(widgetId)** - retrieves the widget in local **widgets** array whose **_id** matches the **widgetId** parameter
4. **updateWidget(widgetId, widget)** - updates the widget in local **widgets** array whose **_id** matches the **widgetId** parameter
5. **deleteWidget(widgetId)** - removes the widget from local **widgets** array whose **_id** matches the **widgetId** parameter

Create Angular Controllers

In the previous assignment the static pages contained static content to illustrate representative data and layout of the pages. In this assignment views will be refactored to instead render dynamic content provided by *controllers* that provide the view with data through a *model*. Controllers will also implement event handlers to map user gestures to logic that manipulates a data model. Data retrieval and manipulation will be done through the services created earlier. Service will be shared across controllers responsible for a particular type of entity. Make sure to use good programming practices discussed in class such as declaring event handlers at the top of the controller, using IIFEs, use view models instead of **$scope**, avoid using **$rootScope**, access/update data through a service. Create a controller file for each of the entity types, e.g., user, website, page, and widget. Create the following controller files

1. **assignment/views/user/user.controller.client.js**
2. **assignment/views/website/website.controller.client.js**
3. **assignment/views/page/page.controller.client.js**
4. **assignment/views/widget/widget.controller.client.js**

In each of the controller files declare controllers for each of the views in their respective directories. For instance, in the **assignment/views/user** directory declare the following controllers in **user.controller.client.js**, one for each of the views that deal with user entities: **LoginController**, **RegisterController**, and **ProfileController**. Here's an example of declaring the **LoginController** and **RegisterController** in **user.controller.client.js**

```
(function() {
    angular
        .module("WebAppMaker")
        .controller("LoginController", LoginController)
        .controller("RegisterController", RegisterController)
        ...
    function LoginController() { ... }
    function RegisterController() { ... }
    ...
})();
```

Use the example above to create all other controllers. Declare the following controllers for websites, pages and widgets, one for each of the views that deal with the website, page and widget entities:

assignment/views/website/ website.controller.client.js	**assignment/views/page/ page.controller.client.js**	**assignment/views/widget/ widget.controller.client.js**

WebsiteListController	**PageListController**	**WidgetListController**
NewWebsiteController	**NewPageController**	**NewWidgetController**
EditWebsiteController	**EditPageController**	**EditWidgetController**

Bind Controllers to their Routes and Views

Once all the controllers have been declared in their respective files, use the **controller** and **controllerAs** properties to bind the controllers to their respective views in **config.js**. For instance, in **config.js**, bind the **LoginController** to the login view as follows

```
function Config($routeProvider) {
    $routeProvider
        .when("/login", {
               templateUrl: "/views/user/login.view.client.html",
               controller: "LoginController",
               controllerAs: "model"
        })
        ...
}
```

Bind the **ProfileController**, and all other controllers, to their respective views.

Bind View Models to Controllers and Templates

In each of the controllers, declare a view model (**vm**) variable bound to the controller instance. The view model will allow controllers and views to exchange data and events. In the corresponding views, use the **ng-model** directive to bind the form elements to the view model declared in the controller. For instance, in the **LoginController** declare a view model bound to the controller instance as follows:

```
function LoginController() {
    var vm = this;
}
```

In the corresponding view, **login.view.client.html**, use **ng-model** to bind the **username** and **password** input elements:

```
<input ng-model="model.user.username" class="form-control" type="text"/>
<input ng-model="model.user.password" class="form-control" type="password"/>
```

Do the same for other controllers and views that have form elements:

	Controllers	Views
1.	RegisterController	register.view.client.html
2.	ProfileController	profile.view.client.html
3.	NewWebsiteController	website-new.view.client.html
4.	EditWebsiteController	website-edit.view.client.html

5.	NewPageController	page-new.view.client.html
6.	EditPageController	page-edit.view.client.html
7.	EditWidgetController	widget-edit.view.client.html

Populate Edit Page Form Elements

Views used to edit an existing instance object such as **profile.view.client.html**, **website-edit.view.client.html**, **page-edit.view.client.html**, and **widget-edit.view.client.html** need to display a form already populated with values from the instance object they are editing. For instance when a user logs in, the user instance needs to be retrieved from the server and the current user properties must be displayed in the profile page. The user can then update the values and submit the changes to the server. The example below illustrates how the **ProfileController** retrieves the **userId** as a path parameter and then uses the **UserService** to retrieve the user instance. The user is then bound to the view model (**vm**) for the template view to render

```
function ProfileController($routeParams, UserService) {
  var vm = this;
  var vm.userId = $routeParams["userId"];
  function init() {
     vm.user = UserService.findUserById(vm.userId);
  }
  init();
}
```

The corresponding view, **profile.view.client.html**, uses **ng-model** to bind the view model with the form elements

```
...
<input ng-model="model.user.firstName" class="form-control" type="text"/>
...
<input ng-model="model.user.lastName" class="form-control" type="password"/>
...
```

Use the example above to complete the other views used for editing existing instances

	Controllers	Views
1.	EditWebsiteController	website-edit.view.client.html
2.	EditPageController	page-edit.view.client.html
3.	EditWidgetController	widget-edit.view.client.html

Populate List Pages

Views that display lists of entities such as **website-list.view.client.html**, **page-list.view.client.html**, and **widget-list.view.client.html** need to iterate over collections of objects. Controllers need to retrieve the data collection from the respective service and bind the collection to the view model. The view can then iterate over the collection rendering each instance using an HTML template.

The example below illustrates how the **WebsiteListController** retrieves the **userId** as a parameter in the path and then uses the **WebsiteService** to retrieve all the websites for a given **userId**

```
function WebsiteListController($routeParams, WebsiteService) {
   var vm = this;
   var vm.userId = $routeParams["userId"];
   function init() {
      vm.websites = WebsiteService.findWebsitesByUser(userId);
   }
   init();
}
```

The corresponding view, **website-list.view.client.html**, uses directive **ng-repeat** to iterate over the collection of websites and use the HTML as a template for each instance in the collection

```
<div ng-repeat="website in model.websites">
  ...
  {{website.name}}
  ...
</div>
```

Use the example above to implement **page-list.view.client.html** and **widget-list.view.client.html**.

Implement Event Handlers

Controllers are responsible for handling user interaction, mapping user gestures to logic, updating the data, and providing data to the view. In each of the views and respective controllers, create event handlers to deal with each of the human gestures in each of the views. For instance, in the login page there's only one gesture: clicking on the login button. Clicking on the register button carries no logic other than navigating to the register page, which is already handled by the angular routing configuration. The example below declares event handler **login()** in the **LoginController** to handle the login button click in the corresponding view **login.client.view.html**

```
function LoginController($location, UserService) {
    var vm = this;
    vm.login = login;
    ...
    function login(user) {
        user = UserService.findUserByCredentials(user.username, user.password);
        if(user) {
            $location.url("/user/" + user._id);
        } else {
            vm.alert = "Unable to login";
        }
    }
}
```

In the corresponding view, **login.view.client.html**, can use **ng-click** directive to bind the login link to the **login()** event handler in the controller

```
<a ng-click="model.login(user)">Login</a>
```

In the example below, **EditWebsiteController** declares event handlers **updateWebsite()** and **deleteWebsite()**

```
function EditWebsiteController($routeProvider, WebsiteService) {
    var vm = this;
    vm.websiteId = $routeProvider.websiteId;
    vm.updateWebsite = updateWebsite;
    vm.deleteWebsite = deleteWebsite;

    function updateWebsite(website) {
        WebsiteService.updateWebsite(vm.websiteId, website);
    }

    function deleteWebsite() {
        WebsiteService.deleteWebsite(vm.websiteId);
    }
}
```

The corresponding view, **website-edit.view.client.html**, can use the **ng-click** directive to bind the update link and delete button to the **updateWebsite()** and **deleteWebsite()** event handlers in the controller

```
<a ng-click="model.updateWebsite(website)">Update</a>
<a ng-click="model.deleteWebsite()">Delete</a>
```

Use the examples above to implement event handlers for the rest of the controllers and views

Merge Heading, Image and YouTube Widgets

In the previous assignment there were three HTML files for editing heading, image, and YouTube videos: **widget-heading.html**, **widget-image.html**, and **widget-youtube.html**. These are the first set of many other widgets that will be implemented. Use the directive **ng-include** to include them into **widget-edit.view.client.html** and **widget-list.view.client.html**. These views need to render one of several templates based on the type of the widget. Views can accomplish this by using the directives **ng-switch** and **ng-switch-when**. The example below illustrates how the view **widget-edit.view.client.html** displays one of the forms based on the widget type

```
<div ng-switch="widget.widgetType">
    <div ng-switch-when="HEADING">
        <ng-include src="'widget-heading.view.client.html'">
    </div>
    <div ng-switch-when="IMAGE">
        <ng-include src="'widget-image.view.client.html'">
    </div>
```

```
    <div ng-switch-when="YOUTUBE">
        <ng-include src="'widget-youtube.view.client.html'">
    </div>
</div>
```

The **widget-list.view.client.html** view iterates over the list of widgets and needs to render different widgets based on the widget type of each of the widget instances in the collection of widgets. The example below iterates over the array of widgets and chooses a different template based on the widget type

```
<div ng-repeat="widget in widgets">
    <div ng-switch-when="HEADING">
        <ng-include src="'widget-heading.view.client.html'">
    </div>
    <div ng-switch-when="IMAGE">
        <ng-include src="'widget-image.view.client.html'">
    </div>
    <div ng-switch-when="YOUTUBE">
        <ng-include src="'widget-youtube.view.client.html'">
    </div>
</div>
```

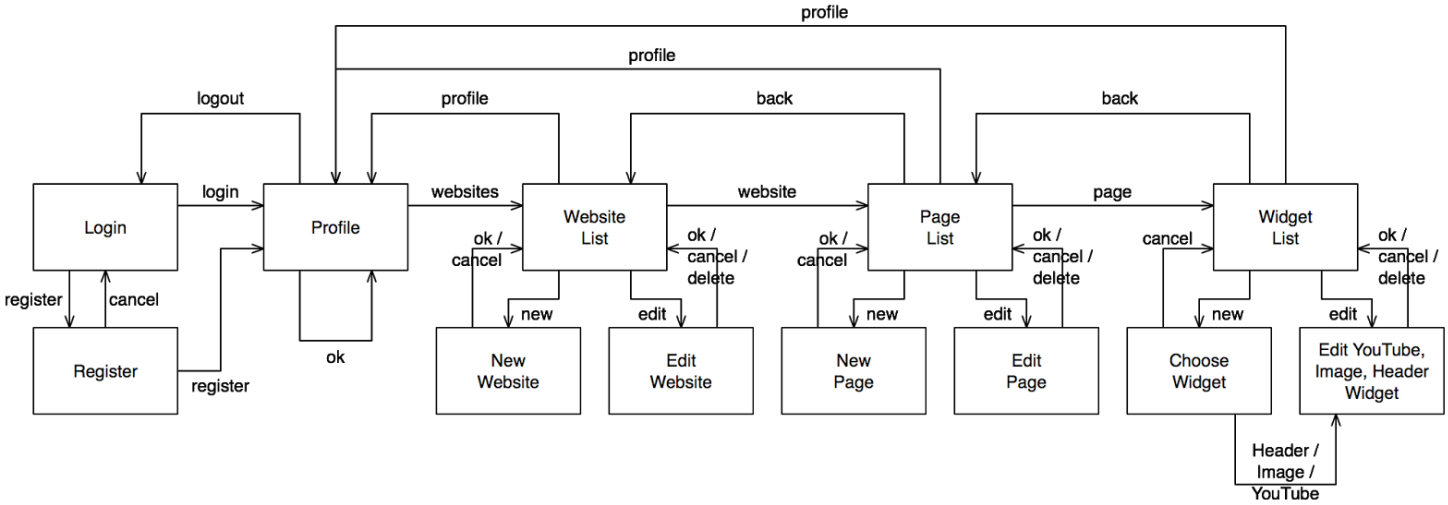
Verify Files and Directory Structure

With all these changes from the previous assignment, below is a list of all the files worked on this assignment. Verify the name and location of the files. All files should be under the **assignment** directory

- **index.html**
- **app.js**
- **config.js**
- **services/**
  - **user.service.client.js, website.service.client.js, page.service.client.js, widget.service.cient.js**
- **views/**
  - **user/**
    - **user.controller.client.js**
    - **login.view.client.html, register.view.client.html, profile.view.client.html**
  - **website/**
    - **website.controller.client.js**
    - **website-list.view.client.html, website-new.view.client.html, website-edit.view.client.html**
  - **page/**
    - **page.controller.client.js**
    - **page-list.view.client.html, page-new.view.client.html, page-edit.view.client.html**
  - **widget/**
    - **widget.controller.client.js**
    - **widget-list.view.client.html, widget-chooser.view.client.html, widget-edit.view.client.html, widget-heading.view.client.html, widget-image.view.client.html, widget-youtube.view.client.html**

Page Flow Diagram

Implement navigation as shown in the page flow diagram and table below. Ignore links and buttons not listed here. Other links and buttons will be addressed in subsequent assignments.



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

We will be using code repository tags (or releases) to "submit" assignments. 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:

assignment1 (previous assignment)
assignment2 (previous assignment)
assignment3 (this assignment)
Assignment4 (next assignment)

assignment5
assignment6 (last assignment)
project

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

assignment3.1, assignment3.2, 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