# Web Development

Database Development

## Description

This assignment adds database persistent storage to the ongoing class assignment. Previous assignments have used client side and server side data management. This assignment uses mongoose data models and schemas to persist and interact with a MongoDB instance. Web services will be refactored to use the data models instead of previous mock data collections. Communication with the database will need to be asynchronous using promises to register callbacks from the database.

## Implement a Data Model

Use mongoose to implement schemas and models for all the assignment's entities: users, websites, pages, and widgets. Refactor Web services to use the mongoose models instead of the static mock data used in previous assignments. Mongoose models must return promises to interact asynchronously with MongoDB. Web services must handle promises returned by the mongoose models. Implement schemas and models under a directory called **`/assignment/model`**. Make sure to apply good programming practices discussed in class such as declaring module APIs towards the top of the module and implementation functions towards the bottom of the module, format source code with white spaces to improve readability and follow naming conventions. Group schemas and models under a directory for the specific entity. Here's a list of all the schemas and models required for this assignment

**`/assignment/model/models.server.js`**
**`/assignment/model/user/user.schema.server.js`**
**`/assignment/model/user/user.model.server.js`**
**`/assignment/model/website/website.schema.server.js`**
**`/assignment/model/website/website.model.server.js`**
**`/assignment/model/page/page.schema.server.js`**
**`/assignment/model/page/page.model.server.js`**
**`/assignment/model/widget/widget.schema.server.js`**
**`/assignment/model/widget/widget.model.server.js`**

### Implement a User Data Model

#### Implement a Mongoose User Schema

Implement a Mongoose schema with the following properties and data types

| Property | Data Type | |
|---|---|---|
| `username` | `String` | |
| `password` | `String` | |
| `firstName` | `String` | |
| `lastName` | `String` | |
| `email` | `String` | |
| `phone` | `String` | |

| | | |
|---|---|---|
| `websites` | `[Website]` | Array of references to child website instances |
| `dateCreated` | `Date` | Current date |

### Implement a Mongoose User Model

| Function Signature | Description |
|---|---|
| `createUser(user)` | Creates a new **`user`** instance |
| `findUserById(userId)` | Retrieves a user instance whose **`_id`** is equal to parameter **`userId`** |
| `findUserByUsername(username)` | Retrieves a user instance whose **`username`** is equal to parameter **`username`** |
| `findUserByCreadentials(username, password)` | Retrieves a user instance whose **`username`** and **`password`** are equal to parameters **`userId`** and **`password`** |
| `updateUser(userId, user)` | Updates user instance whose **`_id`** is equal to parameter **`userId`** |
| `deleteUser(userId)` | Removes user instance whose **`_id`** is equal to parameter **`userId`** |

## Implement a Website Data Model

### Implement a Mongoose Website Schema

| Property Name | Data Type | Default Value / Description |
|---|---|---|
| `_user` | Reference to **`User`** | Refers to parent user |
| `name` | `String` | |
| `description` | `String` | |
| `pages` | `[Page]` | Array of references to child page instances |
| `dateCreated` | `Date` | Current date |

### Implement a Mongoose Website Model

| Function Signature | Description |
|---|---|
| `createWebsiteForUser(userId, website)` | Creates a new **website** instance for user whose **`_id`** is **`userId`** |
| `findAllWebsitesForUser(userId)` | Retrieves all website instances for user whose **`_id`** is **`userId`** |
| `findWebsiteById(websiteId)` | Retrieves single website instance whose **`_id`** is **`websiteId`** |
| `updateWebsite(websiteId, website)` | Updates website instance whose **`_id`** is **`websiteId`** |
| `deleteWebsite(websiteId)` | Removes website instance whose **`_id`** is **`websiteId`** |

# Implement a Page Data Model

## Implement a Mongoose Page Schema

| Property Name | Data Type | Default Value / Description |
|---|---|---|
| `_website` | Reference to `Website` | Refers to parent website |
| `name` | `String` | |
| `title` | `String` | |
| `description` | `String` | |
| `widgets` | `[Widget]` | Array of references to child widget instances |
| `dateCreated` | `Date` | Current date |

## Implement a Mongoose Page Model

| Function Signature | |
|---|---|
| `createPage(websiteId, page)` | |
| `findAllPagesForWebsite(websiteId)` | |
| `findPageById(pageId)` | |
| `updatePage(pageId, page)` | |
| `deletePage(pageId)` | |

# Implement a Widget Data Model

## Implement a Mongoose Widget Schema

| Property Name | Data Type | Default Value / Valid Values |
|---|---|---|
| `_page` | Reference to `Page` | Refers to parent page |
| `type` | `String, enum` | `['HEADING', 'IMAGE', 'YOUTUBE', 'HTML', 'INPUT']` |
| `name` | `String` | |
| `text` | `String` | |
| `placeholder` | `String` | |
| `description` | `String` | |
| `url` | `String` | |
| `width` | `String` | |
| `height` | `String` | |
| `rows` | `Number` | |
| `size` | `Number` | |
| `class` | `String` | |
| `icon` | `String` | |
| `deletable` | `Boolean` | |
| `formatted` | `Boolean` | |
| `dateCreated` | `Date` | Current date |

Implement a Mongoose Widget Model

| Function Signature | |
|---|---|
| `createWidget(pageId, widget)` | Creates new **`widget`** instance for parent page whose **`_id`** is **`pageId`** |
| `findAllWidgetsForPage(pageId)` | Retrieves all widgets for parent page whose **`_id`** is **`pageId`** |
| `findWidgetById(widgetId)` | Retrieves widget whose **`_id`** is **`widgetId`** |
| `updateWidget(widgetId, widget)` | Updates widget whose **`_id`** is **`widgetId`** |
| `deleteWidget(widgetId)` | Removes widget whose **`_id`** is **`widgetId`** |
| `reorderWidget(pageId, start, end)` | Modifies the order of widget at position **`start`** into final position **`end`** in page whose **`_id`** is **`pageId`** |

# 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 **`wam-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 **`wamDirectives`** in a file called **`/public/assignment/directives/wam-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. Refactor the **`widget.service.client.js`** and **`widget.service.server.js`** as needed. Use the following URL pattern for a Web service endpoint to modify the order of the widgets on the server:

PUT `/page/:pageId/widget?start=index1&end=index2`

| | |
|---|---|
| **`pageId`** | id of the page whose widgets are being displayed |
| **`start`** | initial index of the widget before being reordered |
| **`end`** | final index of the widget after being reordered |

# Style the YouTube Widget to be Responsive

Currently the YouTube widget width is responsive, but not the height. Style the widget so that it's size is responsive and maintains the aspect ratio.

```
<div ng-switch-when="YOUTUBE" class="youtube-widget">
    <iframe ... >
    </iframe>
</div>
```

This can be accomplished by applying the following CSS to the DIV including the YouTube iframe

```
.youtube-widget {
    position: relative;
    padding-bottom: 56.25%; /* 16:9 */
    height: 0;
}
```

```
.youtube-widget iframe {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
}
```

# Implement Client Side Validation and Error Handling

Use angular validation and controller logic to enforce the following form element requirements. Highlight invalid fields and display an error message below the invalid field. Display an error alert box at the top of the view styled as a bootstrap alert box, e.g., **`class="alert alert-danger"`**. Only display error messages after a form has been submitted. Turn off autocorrect and auto capitalization for username field in the login view.

| View | Field | Constraint |
|---|---|---|
| Login | **`username, password`** | required |
| Register | **`username, password, verify password`** | required |
| | **`password, verify password`** | must match |
| New Website | **`name`** | required |
| Edit Website | **`name`** | required |
| New Page | **`name`** | required |
| Edit Page | **`name`** | required |
| New Widget | **`name`** | required |
| Edit Widget | **`name`** | required |

# 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:

assignment4 (previous assignment)
**assignment5 (this assignment)**
assignment6 (last assignment)

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

assignment5.1, assignment5.2, 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