**Assignment 4: Client Side Development (Contd.)**

## Part 1: Use Angular Databindingto Populate Pages

**Components** are responsible for:

* Handling user interaction
* Mapping user gestures to logic
* Updating the data
* Providing data to the view

In each of the views and respective components, 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.

**Note:** There’s logic, because data must be retrieved from the database in a certain manner, so it requires logic as to how this will happen.

Clicking on the register button carries no logic other than navigating to the register page, which is already handled by the angular routing configuration.

**Note:** This doesn’t need logic, because it doesn’t need to retrieve or hold any information in order to navigate to the registration page.

**Data-Binding Methods:**

1. **Interpolation** {﻿{...}}- Data flows **from** Typescript code **to** Template (HTML)
2. **Property Binding** [...]- Binds **to** various properties of an HTML component.
3. **Event Binding** (...) - React to user events like click, etc. Data flows **from** Template (HTML) **to** Typescript code
4. **Two-Way-Binding** [(...)] **-** Data flows in **both directions to** and **from** Typescript code **to** Template (HTML)

Here’s an example of how to use data binding concepts in building Components:

|  |
| --- |
| **Note: This is login.component.ts file.**  **Question:** What document is this?  **Question:** What does OnInIt mean/do?  import { Component, OnInit } from '@angular/core';  @Component({  selector: 'app-login',  templateUrl: './login.component.html',  styleUrls: ['./login.component.css']  })  export class LoginComponent implements OnInit {  constructor() { }  login() {    }  ngOnInit() {  }  } |

|  |
| --- |
| **Note: This is the login.component.html file.**  <div class="container-fluid">  <h1>Login</h1>  <form (ngSubmit)="login()">  <div class="form-group">  <input class="form-control" type="text" name="username" placeholder="Username">  </div>  <div class="form-group">  <input class="form-control" type="password" name="password" placeholder="Password">  </div>  <a class="btn btn-primary btn-block" type="submit">Login</a>  <a class="btn btn-success btn-block" routerLink="/register">Register</a>  </form>  </div> |

## Part 2: Use Template-driven Forms to Build Login Form/ Component

(Suggested reading: <https://angular.io/guide/template-syntax>)

1. Use existing HTML login form as created in last assignment
2. Add **FormsModule** in imports array in **app.module.ts** in order to use **ngModel**
3. Add **ngModel** to input field and bind it to a variable in corresponding **login.component.ts** file
4. Add **ngSubmit** event in **form** tag to submit form and invoke **login()** function
5. Add a local reference to the form using **#** syntax
6. Access the form using **@ViewChild** which allows to access a local reference. The instance of **@ViewChild** will be of type **NgForm**
7. Add validation to check user input by adding attributes like **required**, **email**, etc to input fields. See full list of validators available [here](https://angular.io/api/forms/Validators)
8. Disable the Submit button if the form is not valid by using **diable** property binding and local form reference to the form
9. Modify dynamically added CSS classes like **.invalid .touched** of input controls to highlight invalid input fields as red
10. Use a local reference to the input field to display a relevant error message if the field is not valid
11. Access the entered form data in **login()** function using **@ViewChild** reference created in above steps
12. Use structural directive **\*ngIf** to display error message on top of the page if the input credentials do not march

Here’s is the sample code for **LoginComponent**. Use similar pattern to build the complete form for Login and other components:

**login.component.html**

|  |
| --- |
| <div class="container">  <div \*ngIf="errorFlag"  class="alert alert-danger">  {{errorMsg}}  </div>  <h1>Login</h1>  <form (ngSubmit) = "login()" #f="ngForm">  <input placeholder="username"  name="username"  type="text"  class="form-control"  ngModel  #username="ngModel"  required/>  <span class="help-block" \*ngIf="!username.valid && username.touched">  Please enter username!  </span>  <input .../>  <span> ...</span>  <button class="btn btn-primary btn-block"  type="submit"  [disabled]="!f.valid">Login</button>  ...  </form>  </div> |

**login.component.ts**

|  |
| --- |
| import {Component, OnInit, ViewChild} from '@angular/core';  import {NgForm} from '@angular/forms';  @Component({  selector: 'app-login',  templateUrl: './login.component.html',  styleUrls: ['./login.component.css']  })  export class LoginComponent implements OnInit {  @ViewChild('f') loginForm: NgForm;  //properties  username: String;  password: String;  errorFlag: boolean;  errorMsg = 'Invalid username or password !';  constructor(){ }  ngOnInit() { }  login() {  // fetching data from loginForm  this.username = this.loginForm.value.username;  ...  }  } |

## Part 3: Populate Edit Page Form Elements

Views used to edit an existing instance object such as **profile.component.html**, **website-edit.component .html**, **page-edit.component.html**, and **widget-edit.component.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.

A component that we route to has access to something that Angular calls the **ActivatedRoute**. An **ActivatedRoute** is an object that contains information about route parameters, query parameters and URL fragments. **ActivatedRoute** comes with a **params** property which is an **Observable**. To access the user id, we have to subscribe to the parameters **Observable** changes.

The example below illustrates how the **ProfileComponent** retrieves the **userId** as a path parameter and then uses the **UserService** to retrieve the user instance:

|  |
| --- |
| export class ProfileComponent implements OnInit {  //properties  userId: String;  user = {};  username: String;    constructor(private userService: UserService, private activatedRoute: ActivatedRoute) { }  ngOnInit() {  this.activatedRoute.params  .subscribe(  (params: any) => {  **this.userId = params['userId'];**  }  );  this.user = **userService.findUserById(this.userId);**  this.username = this.user['username'];  }  } |

The corresponding view, **profile.component.html**, uses two-way data binding to bind the user details with the form elements:

|  |
| --- |
| ...  ...  <input **ngModel= “{{user.username}}”**  type="text"  class="form-control"  id="username"  placeholder="Alice">  ...  ... |

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**

## Part 4: Populate List Pages

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

The example below illustrates how the **WebsiteListComponent** 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();**  **}** |

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import {WebsiteService} from "../../../services/website.service.client";**  **import {ActivatedRoute} from "@angular/router";**  **@Component({**  **selector: 'app-website-list',**  **templateUrl: './website-list.component.html',**  **styleUrls: ['./website-list.component.css']**  **})**  **export class WebsiteListComponent implements OnInit {**  **userId : String;**  **websites = [{}];**  **constructor(private \_websiteService : WebsiteService, private activatedRoute: ActivatedRoute) { }**  **ngOnInit() {**  this.activatedRoute.params  .subscribe(  (params: any) => {  **this.userId = params['userId'];**  }  );  **this.websites = WebsiteService.findWebsitesByUser(userId);**  **}**  **}** |

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

|  |
| --- |
| **<div \*ngFor="let website of websites">**  **...**  **{{website.name}}**  **...**  **</div>** |

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

## Part 5: Merge Heading, Image and YouTube Widgets

In the previous steps there were three HTML files for editing heading, image, and YouTube videos: **widget-heading.component.html**, **widget-image.component.html**, and **widget-youtube.component.html**. These are the first set of many other widgets that will be implemented. Use the component selector to include them into **widget-edit.component.html** and **widget-list.view.client.html**. These components need to render one of several templates based on the type of the widget. Views can accomplish this by using property [**ngSwitch]** and directive \***ngSwitchCase**.

The example below illustrates how the view **widget-edit.component.html** displays one of the forms based on the widget type:

|  |
| --- |
| <div **[ngSwitch]="widget.type"**>  <div **\*ngSwitchCase**=**'header'**>  **<app-widget-header></app-widget-header>**  </div>  <div **\*ngSwitchCase**=**'image'**>  **<app-widget-image></app-widget-image>**  </div>  <div **\*ngSwitchCase**=**'youtube'**>  **<app-widget-youtube></app-widget-youtube>**  </div>  </div> |

The **widget-list.component.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**:

|  |
| --- |
| <div \***ngFor="let widget of widgets"**>  <div **[ngSwitch]="widget.type"**>  <div **\*ngSwitchCase="'Header'"**>  ...  </div>  <div **\*ngSwitchCase="'YouTube'"**>  ...  </div>  <div **\*ngSwitchCase="'Image'"**>  ...  </div>  </div>  </div> |

## Verify Files and Directory Structure

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.

### flow.png

# Deliverables

## GitHub and Hosted Environment Deliverables

To allow 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 **~/MSIMBO/webdev**:

**$ cd ~/MSIMBO/webdev**

**$ git add .**

**$ git commit -m 'A comment describing your work'**

**$ git push**

If using Heroku, you might need to deploy from within the Heroku dashboard. If you configured Heroku to auto deploy when the repo was updated, then you should be all set.

Verify that the files have copied to the github repository. Also visit your hosted environment 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)

**assignment4 (this 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...

I will grade the very last release. The date/time you create the tag will be considered the date/time of submission.