Developing Web Applications

Dr. Jose Annunziato

Introduction

This is the first of several assignments that together will build on each other to create an online course management system where faculty can author courses and students can register for online tutorials. In this assignment you will create a multi tier, dynamic Web application for admin users to manage other users of a multi user Web application. System administrators will be able to list, create, update and remove users.

Several examples are provided throughout the assignment. All examples are illustrative and not meant to be copied and pasted verbatim. You are responsible for using the code snippet examples for what they are, just examples. Do not expect the examples to work as is. You are meant to write all your code from scratch, using the code snippets provided here and in class as guidance. Ask your instructors for further clarification if needed.

The assignment requires the use of various naming conventions and other common best practices that you are required to employ throughout all assignments. If you are familiar with alternative conventions and industry standards you might have acquired from co-ops or your professional practice, feel free to use those if you feel they are an improvement over the ones suggested here. Confirm with your instructor(s) and TA(s) as it may affect your assignment score.

The wireframes, styles, and look and feel used through the assignment are a minimum requirement and are only illustrative suggestions. You are free to improve on the styling, layout, and look and feel, as long as it is a clear improvement over the illustrations provided. Equivalent use of white space, justification, wrapping, padding, and margins is required. Confirm with your instructor(s) and TA(s) if you have doubts.

Features

The following features will be implemented in this assignment:

1. (40pts) User admin allows admin users to create, find, update and delete users
2. (20pts) Registration allows anyone to register as a user
3. (20pts) Login allows anyone to login
4. (20pts) Profile allows logged in users to update their profile information

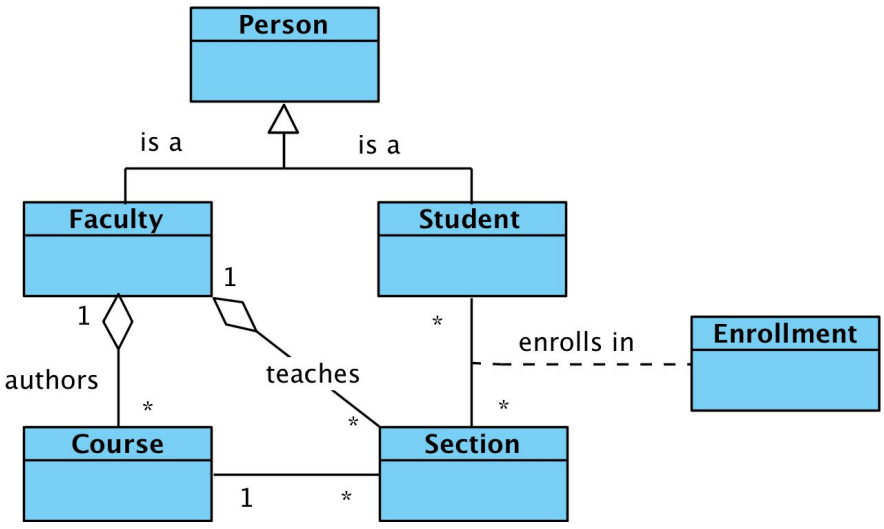
Learning objectives

This assignment will introduce the student to the following concepts and skills:

- Creating webpages with the Hypertext Markup Language (HTML)
- Styling HTML webpages with the Bootstrap Cascading Style Sheet (CSS) library
- Programming webpages with the JavaScript programming language
- Dynamically manipulating a webpage's Document Object Model (DOM) with the jQuery JavaScript library
- Integrating with Representational State Transfer (REST) webservices
- Implementing a Java middle tier using Spring boot
- Representing data using JavaScript Object Notation (JSON)
- Storing and retrieving data from a database with the Java Persistence API (JPA)
- Data modeling with the Unified Modeling Language (UML)

Design

The ***UML class diagram*** below shows the relation between several ***entities*** implemented in this assignment. Entities describe related attributes of an object or concept that we wish to store in a database. The ***base class*** Person captures common ***attributes*** describing all users of the system. The Person ***class*** contains attributes such as id (which uniquely identifies a Person's record), username, password, firstName, lastName, dateOfBirth, role, email, and phone. Faculty and Student classes ***inherit*** attributes from Person, capturing additional, more specific, attributes of faculty and students. We say that Person is a more general form, or a ***generalization*** of Faculty and Student. The inverse is also true, that is, faculty and student are special cases of a person, or they are ***specializations***. Student class captures attributes such as gpa and graduation year. Faculty captures attributes such as whether the faculty is tenured, and their office.



Class diagram

The class diagram further captures the one to many relationship “authors” between faculty and course. That is, one faculty can author many courses. Similarly, there’s a one to many relationship teaches between faculty and section, describing the fact that one faculty can teach several sections. There’s also a one to many relation between courses and sections, that is, one course may have many sections associated with it. And finally, enrollment describes a many to many relation between sections and students, capturing the fact that a student might be enrolled in many sections and that many sections might have many students enrolled in it. The enrollment ***association*** can further describe the relation between a particular student and a particular section such as the grade the student received when enrolled in that section.

Front end

The front end will consist of HTML pages dynamically rendered using JQuery, a popular JavaScript library for programmatically manipulating the DOM. The HTML and JQuery will be used to create a browser based client application.

Middle tier and database

The client application will use AJAX to communicate with a Java based middle tier running on a tomcat server. The Spring boot framework will provide useful abstractions and boilerplate behaviors to define Web services, Java data modeling, and object relational mapping to a relational database. JPA will map Java data models to a MySQL database.

User admin requirements (40pts)

User admin front end requirements (20pts)

User admin front end wireframes

Wireframes capture the content and layout intent of a visual user interface. The styling shown here are only for illustration purposes. You are free to improve on the styling suggested here, or implement the proposed style.

Use CSS libraries such as bootstrap and fontawesome to provide a consistent look and feel across the application. These libraries provide generic example code snippets that can be copied and tailored for a specific use. Although reusing code from well known authoritative sources is encouraged, avoid copying whole entire pages or algorithms. Make sure to provide clear references.

The user admin page must provide a system administrator with the capability to create new users, retrieve all users, retrieve a user for editing, edit existing users, and delete existing users. The following wireframe suggests a user interface that provides all these use cases, but you are free to improve on the design as long as you provide the minimum functionality.

User admin wireframe

User admin front end implementation

The front end will be implemented using HTML, CSS and jQuery. HTML (HyperText Markup Language) is a declarative language used to create and layout the content of Web pages. CSS (Cascading Style Sheet) is a declarative language used to style Web pages, that is, configure the look and feel of Web pages. jQuery is a popular JavaScript library used to dynamically manipulate the DOM easily. The DOM (Document Object Model) is the in-memory representation a browser creates when it parses an HTML document. The browser uses the DOM to represent and then render it as a Web page.

Each Web page will consist of two files: a template file, and a controller file. Template files are implemented as HTML documents and will render the user interface with which the user interacts. Controllers files are implemented as JavaScript files using jQuery to dynamically interact with corresponding HTML documents. Controllers are responsible for listening to user events and inputs, processing those events, communicating with the server, and then rendering dynamic portions of the page. Additional JavaScript files implement services and models shared

among several controller files. Service JavaScript files will encapsulate all data access to and from the server. Model JavaScript files will implement the data model and data structures needed on the client.

The user administration page will provide user administrators with the functionality of creating, retrieving, updating and removing users of the system. The following files implement the user administration page.

File	Description	Points
`user-admin.template.client.html`	implements the user interface using HTML	5
`user-admin.style.client.css`	implements the styling of the user admin page using CSS	1
`user-admin.controller.client.js`	implements the controller handling user events and rendering dynamic portions of the user admin page	5
`user.model.client.js`	implements the user data model	1
`user.service.client.js`	implements the user service client encapsulating all data communication with the server (8pts)	8

Notice the file naming convention. The naming follows the pattern

`{feature}.{role}.{tier}.{extension}`

where the file parts mean the following

File part	Description	Example values
`feature`	describes a feature, component, or page	`user-admin, login, register, profile`
`role`	describes the role of the file in the overall architecture of the application	`model, service, template, controller, style`
`tier`	describes where in the multi tier application does the file execute	`client, server`
`extension`	file extension describing the language the file is written in	`js, html, css, ts, sql, json, xml`

This naming convention allow developers to infer the purpose of a file by its name, and quickly orient themselves where in the application are they working on. All files must follow a naming convention agreed by the development team. Follow the above naming convention unless you have a better one. Check with your instructor if you prefer using a different naming convention.

User admin template (5pts)

The user admin template renders the user interface for the user admin page using HTML and CSS as shown in the user admin wireframes. Implement the template in a file called `user-admin.template.client.html` in the `jquery/components/admin` folder. The following are code snippet samples that can be used as a starting point. Feel free to change the markup or provide an entirely better implementation

Loading relevant JavaScript libraries and stylesheets

The first thing to do is to load all relevant scripts and stylesheets. Use the link and script elements to load these dependencies. Here's an example:

`jquery/components/admin/user-admin.template.client.html`

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>User Admin</title>
<link rel="stylesheet" href=
  "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"/>
<link rel="stylesheet" href=
  "https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"/>
<link href="user-admin.style.client.css" rel="stylesheet" />
<script
  src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script src="../services/user.service.client.js"></script>
<script src="user-admin.controller.client.js"></script>
</head>
<body>
...
</body>
```

The `user.service.client.js`, `user-admin.controller.client.js`, and `user-admin.style.client.css` files are discussed later. The following sections describe the content of the `body` element.

Table headings

A table head can render meta data, such as titles, for columns containing data of the same type. For instance, the following table head renders column headers for username, password, first and last name. The last column is left intentionally empty (` `) so that each row below it can render its own action links or buttons.

jquery/components/admin/user-admin.template.client.html

```
<table class="table">
  <thead>
    <tr>
      <th>Username</th>
      <th>Password</th>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Role</th>
      <th> </th>
    </tr>
```

Create/update user form

The second row of the table head can be used as a form with input fields for each of the properties of a new user, e.g., username, password, first and last name. The last column renders three action icons to search, update, and create a user.

jquery/components/admin/user-admin.template.client.html

```
<tr class="wbdv-form">
  <td><input id="usernameFld" class="form-control"
    placeholder="Username" /></td>
  <td><input id="passwordFld" class="form-control"
    placeholder="Password" /></td>
  <td><input id="firstNameFld" class="form-control"
    placeholder="First Name" /></td>
  <td><input id="lastNameFld" class="form-control"
    placeholder="Last Name" /></td>
  <td><select id="roleFld" class="form-control">
      <option value="FACULTY">Faculty</option>
      <option value="STUDENT">Student</option>
  </select></td>
```

```
  <td><span class="float-right" style="white-space: nowrap">
      <i class="fa-2x fa fa-search wbdv-search"></i> <i
      class="fa-2x fa fa-plus wbdv-create"></i> <i
      class="fa-2x fa fa-check wbdv-update"></i>
  </span></td>
</tr>
```

Notice the naming convention of adding Fld at the end of the input fields and prepending wbdv to our own classes to distinguish them from classes implemented by other libraries such as bootstrap or fontawesome.

User template row

The list of users will be rendered in the table body as a collection of rows, one row per user. Each column of the row will render a different user property. An invisible row van be used as a template to be cloned and populate for every instance in a user array. Here's an example of what the table row template might look like.

```
jquery/components/admin/user-admin.template.client.html
```

```
  <tbody class="wbdv-tbody">
    <tr class="wbdv-template wbdv-user wbdv-hidden">
      <td class="wbdv-username">ada</td>
      <td> </td>
      <td class="wbdv-first-name">Ada</td>
      <td class="wbdv-last-name">Lovelace</td>
      <td class="wbdv-role">FACULTY</td>
      <td class="wbdv-actions">
        <span class="float-right">
          <i id="wbdv-remove" class="fa-2x fa fa-times wbdv-remove"></i>
          <i id="wbdv-edit" class="fa-2x fa fa-pencil wbdv-edit"></i>
        </span>
      </td>
   </tr>
 </tbody>
</table>
```

Commit your work into source control

As you complete a significant amount of work, and reach a stable state of your project, commit your work into source control. Add and commit the files created in this section. Use a commit command and message similar to the one below.

```
Git commit command and message
```

```
git add .
git commit -m 'Created user admin template'
```

Do the same for every section that follows where a new file is created, or new functionality is added. Every missing commit will deduct a point from the final score.

User admin styling (1pt)

The user admin page should use the bootstrap css library and the fontawesome font library (or equivalent) to provide a consistent look and feel across the application. For instance, notice that the table stretches across the entire page as opposed to the default HTML table styling. Also notice that all content has consistent padding and margins, avoiding content to be flush with their containers. Bootstrap provides many code snippets that can be reused without having to reinvent the layout and styling. Feel free to make use of predefined code snippets, but avoid copying entire pages. Implement the user admin styling in a file called `user-admin.style.client.css` in the folder `jquery/components/admin`. Here's a snippet of code on how to load the CSS files from the HTML template file. Don't forget to commit your work.

`jquery/components/admin/user-admin.template.client.html`

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>User Admin</title>
<link rel="stylesheet" href=
  "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"/>
<link rel="stylesheet" href=
  "https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"/>
<link href="user-admin.style.client.css" rel="stylesheet" />
<script
  src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script src="admin-user.service.client.js"></script>
<script src="admin-user.controller.client.js"></script>
</head>
```

User model (1pt)

To represent users on the client, create a JavaScript file that implements a User JavaScript class with the following properties: username, password, email, firstName, lastName, phone, role, and dateOfBirth. Use this class to create instances of users whenever sending or receiving data from the server, or rendering users in the UI. Create the class in a file called `user.model.client.js` in the `jquery/models` folder. Here's a snippet of code to get you started. Don't forget to commit your work.

jquery/models/user.model.client.js

```
function User(username, password, firstName, lastName) {
  this.username = username;
  this.password = password;
  // ...same for rest of properties…

  this.setUsername = setUsername;
  this.getUsername = getUsername;
  // ...same for rest of properties…

  function setUsername(username) {
    this.username = username;
  }
  function getUsername() {
    return this.username;
  }
  // ...same for rest of properties…
}
```

User admin controller (5pts)

Controllers are responsible for handling user generated input and events, and providing data for rendering the view. The user admin controller will handle events such as create user, find all users, find user by id, update user, and delete user. Implement the user admin controller in a file called `user-admin.controller.client.js` in the `jquery/components/admin` folder. The controller should get a reference to the form elements in corresponding template, bind event handlers to create a new user, find, update and delete an existing user, and retrieve all users. Here's a template for the user admin controller. You'll need to implement the functions listed in the controller. Commit your work when done.

jquery/components/admin/user-admin.controller.client.js

```
(function () {
    var $usernameFld, $passwordFld;
    var $removeBtn, $editBtn, $createBtn;
    var $firstNameFld, $lastNameFld;
    var $userRowTemplate, $tbody;
    var userService = new AdminUserServiceClient();
    $(main);

    function main() { … }
    function createUser() { … }
    function findAllUsers() { … }
    function findUserById() { … }
```

```
    function deleteUser() { ... }
    function selectUser() { ... }
    function updateUser() { ... }
    function renderUser(user) { ... }
    function renderUsers(users) { ... }
})();
```

The user admin controller will have to implement the following event handlers.

`jquery/components/admin/user-admin.controller.client.js`

Function	Description
`main()`	executes on document load, when the browser is done parsing the html page and the dom is ready. Retrieved the dom elements needed later in the controller such as the form elements, action icons, and templates. Binds action icons, such as create, update, select, and delete, to respective event handlers
`createUser()`	handles create user event when user clicks on plus icon. Reads from the form elements and creates a user object. Uses the user service createUser() function to create the new user. Updates the list of users on server response
`findAllUsers()`	called whenever the list of users needs to be refreshed. Uses user service findAllUsers() to retrieve all the users and passes response to renderUsers
`findUserById()`	called whenever a particular user needs to be retrieved by their id, as in when a user is selected for editing. Reads the user is from the icon id attribute. Uses user service findUserById() to retrieve user and then updates the form on server response
`updateUser()`	handles update user event when user clicks on check mark icon. Reads the user is from the icon id attribute. Reads new user values form the form, creates a user object and then uses user service updateUser() to send the new user data to server. Updates user list on server response
`deleteUser()`	handles delete user event when user clicks the cross icon. Reads the user is from the icon id attribute. Uses user service deleteUser() to send a delete request to the server. Updates user list on server response
`renderUser()`	accepts a user object as parameter and updates the form with the user properties
`renderUsers()`	accepts an array of user instances, clears the current content of the table body, iterates over the array of users, clones a table row template for

	each user instance, populates the table row with the user object properties, adds the table row to the table body

User admin service client (8pts)

The user admin service implements all data access to the server. Implement the user admin service in a file called user.service.client.js in the jquery/services folder. Here's a template for the user service. **Last reminder: commit your work when done.**

jquery/services/user.service.client.js

```
function UserServiceClient() {
    this.createUser = createUser;
    this.findAllUsers = findAllUsers;
    this.findUserById = findUserById;
    this.deleteUser = deleteUser;
    this.updateUser = updateUser;
    this.url = 'http://localhost:8080/api/user';
    var self = this;
    function createUser(user, callback) { ... }
    function findAllUsers(callback) { ... }
    function findUserById(userId, callback) { ... }
    function updateUser(userId, user, callback) { ... }
    function deleteUser(userId, callback) { ... }
}
```

The user service should implement the following functions to interact with the user web service.

Function	Description	API
`createUser()`	accepts a user object and POSTs it to a user Web service. Receives status	`POST /api/user`
`findAllUsers()`	sends a GET request to user Web service. Receives a JSON array of all users	`GET /api/user`
`findUserById()`	sends a GET request to user Web service with userId as path parameter. Receives a single JSON object for the userId	`GET /api/user/{userId}`
`updateUser()`	accepts a user id and user object with new property values for the user with user id. Sends PUT request with user object and user id as path parameter	`PUT /api/user/{userId}`

`deleteUser()`	sends a DELETE request to user Web service with user id as path parameter for user to remove. Receives status	`DELETE /api/user/{userId}`

User admin middle tier requirements (20pts)

User admin middle tier implementation

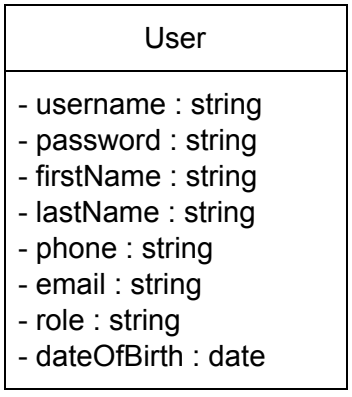
The middle tier (server side) will be implemented using Java supported by various Spring Boot libraries that encapsulate Java APIs such as JPA (Java Persistence API) and JAX-RS (Java API for RESTful Web Services)

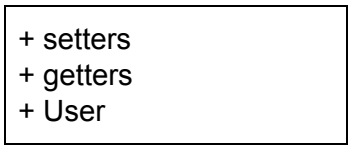
The following files will implement the user administration middle tier.

File	Description	Points
`User.java`	Implements the user data model including properties that describe all users	5
`UserService.java`	Implements the RESTful Web service that exposes behavior and data through a set of URL patterns and actions	10
`UserRepository.java`	Implements mapping between Java classes and instances and their corresponding relational schemas, tables, and records	4
`user.sql`	Implements the relational schema that maps to the corresponding Java class	1

User admin model (5pts)

In a package called webdev.models, implement a class called User based on the following UML class diagram





All classes must have public setters and getters adhering to the JavaBeans naming convention.

src/java/webdev/models/User.java

```
package webdev.models;

import javax.persistence.*;

@Entity
public class User {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private int id;
  private String username;
  private String password;
  private String firstName;
  private String lastName;
  // ... same for rest of properties

  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getUsername() {
    return username;
  }
  public void setUsername(String username) {
    this.username = username;
  }
  // ... same for rest of the properties
}
```

Did you remember to commit your work? For ALL the new files and new features from the last reminder?

User admin repository (4pts)

In package webdev.repositories, create a JPA CRUD repository called UserRepository that provides the default CRUD operations plus the following queries

src/java/webdev/repositories/UserRepository.java

```
package webdev.repositories;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;

import webdev.models.User;
public interface UserRepository extends CrudRepository<User, Integer> {}
```

User database schema (1pts)

Using MySQL workbench, verify that the User.java class has mapped to an equivalent relational schema similar to the one shown below. Copy or export the SQL CREATE statement to a file called user.sql. Save it to the same directory where User.java lives.

src/java/webdev/models/user.sql

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(255) DEFAULT NULL,
  `last_name` varchar(255) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  `username` varchar(255) DEFAULT NULL,
  `role` varchar(255) DEFAULT NULL,
  `phone` varchar(255) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `date_of_birth` DATETIME DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=latin1;
```

User admin Web service (10pts)

Web services are HTTP endpoints that expose a behavior or data and allow integrating various distributed systems. The user admin front end will be executing on the browser as a client application. The client application will need to integrate with a server side application through Web services executing on the server application. The Web service endpoints expose the URL patterns and actions needed to support the user admin client application described earlier

Function	Description	API
`createUser()`	accepts a POST with a user object embedded in the HTTP BODY. Parses the user from the BODY and then uses	`POST /api/user`

	the UserRepository to insert it in the database	
`findAllUsers()`	accepts a GET request and responds with a JSON array of all users	`GET /api/user`
`findUserById()`	accepts a GET request and parses the userId from the path. Retrieves the user whose primary key is equal to the userId, and then returns to user a single JSON object	`GET /api/user/{userId}`
`updateUser()`	accepts an HTTP PUT request, parses the userId as a path parameter and a User.java instance from the HTTP BODY. Uses the UserRepository to find the user by its userId and then updates the record with the new values in the user object	`PUT /api/user/{userId}`
`deleteUser()`	accepts an HTTP DELETE action, parses the userId as a path parameter and uses the UserRepository to remove the user whose id is equal to the userId path parameter	`DELETE /api/user/{userId}`

Web services use repositories to implement their respective behavior. For instance, the `findAllUsers()` service endpoint uses the user repository `findAll()` method to retrieve and return all user instances from the database. Below is a sample snippet illustrating the implementation of `findAllUsers()`

src/java/webdev/services/UserService.java

```
package webdev.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.*;

import webdev.models.User;
import webdev.repositories.UserRepository;

@RestController
public class UserService {
      @Autowired
      UserRepository userRepository;
```

```
	@GetMapping("/api/user")
	public List<User> findAllUsers() {
		return userRepository.findAll();
	}
}
```

User signup requirements (20pts)

The user signup feature will allow all users to register with the system so that they can login into they personal profiles.

User signup front end requirements (10pts)

The user signup front end will provide a form for users to enter their preferred username and password. If the username is not already taken, the application will create the new user and then automatically logged them in. The following files will implement the user administration page.

File	Description	Points
`register.template.client.html`	implements the user interface using HTML	3
`register.style.client.css`	implements the styling of the user admin page using CSS	1
`register.controller.client.js`	implements the controller handling user events and rendering dynamic portions of the user admin page	4
`user.service.client.js`	implements the user service client encapsulating all data communication with the server (8pts)	2

User signup wireframes

Users can sign up using the registration page show below. The page is responsive having a desktop and a mobile version.

Desktop registration wireframe

Mobile registration wireframe

Clicking on the Login link navigates to the login page. Clicking on the Sign up button, navigates to the profile page, with the newly registered user currently logged in.

User signup template (4pts)

The register template implements the user sign up user interface as described in the registration wireframes. Create the register template in a file called `register.template.client.html` in the `jquery/components/register` folder. Below is a snippet of code for the template. Add additional markup and styling as shown in the login template.

```
jquery/register/register.template.client.html
```

```
<h1>Register</h1>
Username: <input id="usernameFld"/>
Password: <input id="passwordFld"/>
Verify Password: <input id="verifyPasswordFld"/>
<button id="registerBtn">Register</button>
```

User signup styling (1pt)

Implement any register specific styling in a file called `register.style.client.css` in the same folder.

User signup controller (4pts)

The register controller is very similar to the login controller. The register controller should bind to the form elements, and bind the sign up button to a register event handler. The register event

handler should use the user service's register function to post a user object with the username and password for the new user. Create the controller in a file called `register.controller.client.js` in the `jquery/components/register` folder. Here's a snippet of the controller to get you started.

`jquery/register/register.controller.client.js`

```
(function () {
    var $usernameFld, $passwordFld, $verifyPasswordFld;
    var $registerBtn;
    var userService = new UserService();
    $(main);

    function main() { … }
    function register() { … }
})();
```

User signup service (2pts)

Find user by username (2pts)

To support the registration behavior, add a register service to the already existing user service.

Register – posts to a register web service

`jquery/service/user.service.client.js`

```
function UserService() {
  this.register = register;
  ...
  function register() { ... }
}
```

User signup front middle tier requirements (10pts)

The following files will implement the user signup middle tier.

File	Description	Points
`UserRepository.java`	add findUserByUsername() to verify if the username is already taken	2
`UserService.java`	add register() and findUserByUsername() to use the repository to verify the username and then create the	8

user in the database

Implement findUserByUsername in user repository (2pts)

In the UserRepository.java file created earlier, add a findUserByUsername() interface function and corresponding JPQL to retrieve a user by their username.

Implement findUserByUsername in user service (3pts)

In UserService.java, add a findUserByUsername() method that parses the username from a query parameter called username, then uses the repository's findUserByUsername() method to retrieve the user and returns the user instance. Note that you might need to refactor the findAllUsers() method.

Implement register in user service (5pts)

In UserService.java, implement a register() web service endpoint method mapped to a POST action on /api/register URL pattern. The endpoint should use the user repository findUserByUsername() method to verify that the username is not already in use. If not in use, the endpoint should use the user repository createUser() to insert the new user into the database and then add the new user to the session attribute "user" to set the new user as currently logged in. Subsequent HTTP requests can check for the "user" session attribute to verify that there's a logged in user.

`src/java/webdev/services/UserService.java`

```
@PostMapping("/api/register")
public User register(@RequestBody User user, HttpSession session) { ... }
```

Login requirements (20pts)

Login front end requirements (10pts)

The following files will implement the user administration page.

File	Description	Points
`login.template.client.html`	implements the user interface using HTML	3
`login.style.client.css`	implements the styling of the login page using CSS	1
`login.controller.client.js`	implements the controller handling user	4

	events and rendering dynamic portions of the user admin page	
`user.service.client.js`	add login() web service client	2

Login wireframes

The login page allows users to login to the system. Clicking on the sign in button validates whether the user is a valid user and if so, navigates the user to the profile page. The login page, as many other pages must be responsive to the size of the screen. The images below illustrate the login page as rendered in a desktop and in a smaller mobile screen. You are free to make improvements on the design and styling as long as the basic functionality is supported. The forgot password feature is not required for this assignment, but can be completed as a bonus (See Bonus section). Clicking on the Sign up link navigates to the registration page.

Desktop login wireframe

Mobile login wireframe

Login template (3pts)

The login template implements the user interface for the login wireframe. The template renders a sign in title, username and password labels and inputs, a sign in button, and links to forgot password and sign up. The sign up link navigates to the sign up page described elsewhere. The forgot password link navigation is a stretch bonus described in the Stretch section at the end of this document. Here's a snippet of code to get you started on the login page.

`jquery/components/login/login.template.client.html`

```
<div class="container">
  <h1>Sign In</h1>
  <form>
    <div class="form-group row">
      <label for="username" class="col-sm-2 col-form-label">
```

```
        Username </label>
      <div class="col-sm-10">
        <input class="form-control" id="username" placeholder="Alice">
      </div>
    </div>
    <div class="form-group row">
      <label for="password" class="col-sm-2 col-form-label">
        Password </label>
      <div class="col-sm-10">
        <input type="password" class="form-control wbdv-password-fld"
          id="password" placeholder="123qwe#$%">
      </div>
    </div>
    <div class="form-group row">
      <label class="col-sm-2 col-form-label"></label>
      <div class="col-sm-10">
        <button class="btn btn-primary btn-block">Sign in</button>
        <div class="row">
          <div class="col-6">
            <a href="#">Forgot Password?</a>
          </div>
          <div class="col-6">
            <a href="#" class="float-right">Sign up</a>
          </div>
        </div>
      </div>
    </div>
  </form>
</div>
```

Login styling (1pt)

Implement styling specific to the login page in a CSS file called `login.style.client.css` and link to it from the template file.

Login controller (4pts)

The login controller handles user events, dynamic DOM updates, and navigation. Implement the login controller in a file called login.controller.client.js in the directory jquery/components/login. Here's a snippet of code to get you started.

jquery/components/login/login.controller.client.js

```
(function () {
    var $usernameFld, $passwordFld;
    var $loginBtn;
```

```
    var userService = new AdminUserServiceClient();
    $(main);

    function main() { … }
    function login() { … }
})();
```

Login service (2pt)

To support the logging behavior, add a login() service function to the user service client. The login() function posts credentials including username and password to a user web service.

```
jquery/services/user.services.client.js
```

```
function UserService() {
  …
  this.login = login;
  …
  function login() { … }
  …
}
```

Login middle tier requirements (10pts)

Implement findUserByCredentials() Web service endpoint (4pts)

To support the login functionality, implement a finder in the user repository that can retrieve users by username and password. Call the finder method findUserByUsernameAndPassword() which takes username and password strings and queries the database for a matching user record. Declare the finder method in the UserRepository interface implemented earlier. Provide the JPQL that the interface method will need. The JPQL statement should use the username and password parameters provided in the method call to parameterize the query that retrieves the user.

Implement the login() Web service endpoint (6pts)

To support the login functionality, implement a login() HTTP service endpoint that handles the login post request. Since this is a user related functionality, the request handler can live in the user web service UserService.java. The post request handler will parse a User instance from the request body as a parameter, and validates whether the user parameter is a valid user using the username and password in the user parameter instance. Here's a snippet of code example of what the request handler might look like.

```
src/java/webdev/services/UserService.java
```

```
@PostMapping("/api/login")
public User login(@RequestBody User user, HttpSession session) { ... }
```

The `login()` method uses the `findUserByUsernameAndPassword()` method in the user repository to find the user. If the user exists, the login should add the user to the HTTP session passed in as a parameter. Save the found user in a session variable called "user". This session variable can be used later by the user profile page to retrieve the information of the currently logged in user. The login function should return the user object which will be sent back to the client. The client login controller will use the user object to decide whether the login was successful or not and whether to navigate to the profile page.

Profile requirements (20pts)

Once a user registers or logs in they can view or edit their personal information from a profile webpage.

Profile front end requirements (10pts)

The following files implement the profile webpage.

File	Description	Points
`profile.template.client.html`	implements the user interface using HTML	3
`profile.style.client.css`	implements the styling of the webpage using CSS	1
`profile.controller.client.js`	implements the controller handling user events and rendering dynamic portions of the user admin page	4
`user.service.client.js`	add `updateProfile()` and `logout()` Web service client	2

Profile wireframes

The profile page shows a user's personal information once they've logged in or registered. The page is responsive having a desktop and mobile version as shown below.

Clicking on the Logout button logs the currently logged in user and navigates to the Login page. Clicking on the Update button saves the changes, stays in the same page, and a success alert appears at the top of the screen. The date of birth input field is of type date and renders as a date picker. The username field is readonly and can not be modified. The role field is a select input field with options Faculty, Student, and Admin. The role field would typically be read only

so that users can not change their own role. Role management would be an admin only use case, but let's allow users to change their role for now. Later assignments might change it to readonly as necessary.

Profile template (3pts)

Implement the profile template in a file called `profile.template.client.js` in a directory called `jquery/components/profile`. Use the wireframes, the sign in, and the sign up HTML code samples to get started.

Desktop profile wireframe

Mobile profile wireframe

Profile date of birth date input	Profile role options	Profile successfully saved alert message

Profile styling (1pt)

Implement styling specific to the profile template in a file called `profile.style.client.css` in a directory called `jquery/components/profile`. The template should render the title, labels, input fields, and buttons. Use the wireframe as a guidance. Feel free to improve on the layout and styling. Remember to make good use of white space (padding, margins), text wrapping, justification, and alignment throughout all your webpages.

Profile controller (4pts)

Implement the profile controller in a file called `profile.controller.client.js` in a directory called `jquery/components/profile`. Use the sign in, the sign up, and user admin controller code samples to get started. The buttons should invoke event handlers `logout()` and `updateProfile()` in the controller, which delegate the actions to corresponding functions in the user service.

Profile service client (2pts)

In the user service client implemented earlier, implement webservice client functions updateProfile() and logout() which use AJAX to send RESTful HTTP requests to server side webservices that implement the behaviors. The updateProfile() function should send a PUT request to /api/profile, passing a user instance with the new profile information embedded in the BODY of the HTTP request, formatted as a JSON object.

Profile middle tier requirements (10pts)

Profile webservice

In the UserService.java class, implement webservice endpoints updateProfile() and logout() mapped to /api/profile and /api/logout respectively. These endpoints will support the corresponding webservice client updateProfile() and logout() HTTP requests.

Implement updateProfile() webservice endpoint (6pts)

The updateProfile() webservice endpoint will respond to a PUT HTTP request mapped to /api/profile. The updateProfile() endpoint will check if a user is currently logged in, by retrieving the "user" attribute from the session. If the user attribute is set, then we are sure the request was made from a legitimate, logged in user, and we can use the User.java instance mapped to the attribute. Using the id property of the logged in user, we can then update the corresponding user properties in the database. If the user property is not set, then the request was not made from a logged in user, and the service should refuse to fulfill the request, and return null.

```
src/java/webdev/services/UserService.java
```

```
@PutMapping("/api/profile")
public User updateProfile(@RequestBody user, HttpSession session) { ... }
```

Implement logout() webservice endpoint (4pts)

The logout() webservice endpoint will respond to a POST HTTP request mapped to /api/logout. The endpoint will just invalidate the current session and return. Subsequent access to the profile, or other protected content, will not succeed since the session is no longer valid.

```
src/java/webdev/services/UserService.java
```

```
@PostMapping("/api/logout")
public User login(HttpSession session) { ... }
```

Database requirements

User database requirements

Verify the user table has a schema similar to the one below and is available in a file called user.sql in the webdev/models directory

src/java/webdev/models/user.sql

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(255) DEFAULT NULL,
  `last_name` varchar(255) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  `username` varchar(255) DEFAULT NULL,
  `role` varchar(255) DEFAULT NULL,
  `phone` varchar(255) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `date_of_birth` DATETIME DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=latin1;
```

Bonus

1. (3pts) Forgot password - Users can enter an email to which they are sent a link they can click on and reset their password
2. (1pts) Login validation - If login is unsuccessful, a message is displayed with the reason
3. (1pts) Registration validation - if registration is unsuccessful, a message is displayed with the reason
4. (1pt) Password validation - if passwords don't match when registering, a message is displayed with the reason

Source control requirements

Assignments and final project will be submitted through github source control. Use the school's github account. If you don't have one, create a free repository at github.com. Invite all instructors and TAs as collaborators to your github repository. They will need full access to clone your repository and help debug issues. There should be a commit for every single new file, and every new function or feature added to an already existing file. A point will be deducted for every missing commit with a maximum of 5 points.

Deliverables

As a deliverable, check in all your work into source control. Submit the URL of the repository in blackboard.

References

jQuery

[Spring Tool Suite™ Downloads]
[Sign up for free Heroku account]
[Deploying Spring Boot Applications to Heroku]
[Deploying Spring Boot Applications to Heroku (Slides)]
[Download MySQL Workbench]

Review

What do the following abbreviations and acronyms mean and what are they used for?
HTML, HTTP, CSS, JPA, DOM, AJAX, REST, JSON, UML, ORM, SQL, CRUD, URL, JPQL, CLI, STS

What are the following technologies used for?
jQuery, MySQL, Java Script, Java, Bootstrap, Spring Boot, Database

What do the following terms mean?
Server, Client, Browser, One to many, Many to many, Inheritance, Cardinality, Relationship, Primary key, Foreign key, Private, Public, Protected, Web service, Path Parameter, Query Parameter, HTTP BODY, Cookie, Session

TODO

This section is for internal use. Please ignore

- Add first name, last name to profile