Sycamore Scheduler

Team Members

Adina Kruijssen, kruijsse@usc.edu

Guillermo M. Leal Gamboa, lealgamb@usc.edu

James Meyer, jamesmey@usc.edu

Tran Ngo, tranngo@usc.edu

Sajeev Saluja, sajeevsa@usc.edu

Table of Contents

Proposal

Overview

We want to help USC students build their course schedules. To this end, the user can create a profile with their major/minor, and classes they have already taken, and the website will display the remaining courses needed to complete their degree program(s). Ideally we want to have the website automatically generate the user's schedule for a given semester. The user could also give constraints like desired professors, and matching classes with a friend. The schedule can be exported to Google Calendar. The data for classes and degree programs are to be scraped from the USC courses website, which is public facing.

GUI

We plan on using front-end languages (HTML and CSS) to develop and design our GUI. We could use frameworks, such as Bootstrap, to facilitate this. Using JavaScript and CSS can also enable us to make animations, such as a loading animation when users are loading their previously saved schedule(s).

Limited Functionality

One of the ways we can limit functionality for guests is by not allowing the user to save their schedules. Authenticated users can save the schedules they create onto our database so that they can pull it up the next time they visit the web app.

Multithreading

Multiple users can edit a course schedule simultaneously if in collaborative mode.

Networking

Users will be able to view events on their Google Calendar when they are creating their schedules. This will allow them to plan accordingly.

Users will also be able to edit schedules in groups of two or more (e.g. students who want to take classes together) and see the live changes on their own computers. This is similar to editing a Google Doc, except that users will be working on the schedule. This is essentially a cooperative/team mode that will be a feature of our project for registered users.

High-Level Requirements

We want to create a web application that simplifies course planning for USC students.

When entering the website, students will have the option to continue as a guest, sign into an existing profile, or create a new profile. If the user creates a new profile, they must enter their degree program(s), current year, and classes taken. Guests will not be able to input any of this information, and can simply build a course plan using the courses available on the site, and export it for offline access only.

At this point the user will be directed to the main page of the website, which shows a list of courses for that user's degree program. The user can click on any course and view more information about it. If they'd like to take that course, they can add it to their course plan. When the user is finished shopping for courses, they can choose to view their schedule, which will direct them to a new page.

When editing their schedules, users should be able to complete basic actions. These actions include adding courses, viewing courses, requesting to see and add other users' course schedules, adding users to cooperate with, blocking users from viewing or adding their schedule, and other similar actions that a user may require.

Registered users should also be able to cooperate with other users. Multiple users can edit a course schedule simultaneously if they share a major and/or minor. For example, the two users can select professors and courses they wish to take together. Based on these criterias set by the user, the web app should generate a new set of courses from which the users can choose. This should update on both of the user's screens in real-time.

When users are editing their course schedule, they should have access to additional resources to guide them in the schedule-creating process, like an official course plan containing the requirements for the user's academic program. An example of this would be having the Viterbi handbook for engineering majors. Access to these resources should still be within the web interface like a small minimizable window at the bottom right of the screen.

If a user saves their schedule, the schedule will be saved in our database. Users can then access the database by logging into their account and finding their previously saved schedule on the web interface.

When generating the schedule for the user, our web app will use data previously garnered from the USC Courses website to determine which courses are mandatory (e.g. prerequisites, corequisites, etc.) for their respective academic program. Our web app will also have access to the courses that are available/offered for the upcoming terms, which will assist the user in determining which classes they might want to take in the future.

Overall, this web interface should make course scheduling for students simpler and provide them with a visual of what their schedule will look like for the upcoming academic

terms. The user should be able to export this schedule to have offline access to it and add the schedule to their Google Calendar.

Technical Specifications

Web interface

- 24 hours
- Start with a login page with three buttons: Login, Sign up, Enter as Guest
    - If the user clicks Sign Up, they will be redirected to a Sign Up page. The page will have a form asking for name, email, current major(s), and current minor(s). When they select their major(s)/minor(s), more fields will display asking which courses from those degree program(s) they have taken.
        - After clicking the submit page (and the information has been successfully submitted to our database), the user will be redirected to a page that asks for the courses that he or she has taken.
    - If the user clicks Login, the user will be prompted a form that asks for username and password. If the user clicks on submit, the user will either be redirected to their homepage (their credentials were valid) or the user will stay on the same page and be prompted to re-enter their username and password (the user entered incorrect credentials).
    - If the user clicks Guest, they will be prompted for their major and then taken to the normal homepage, where they can look at the classes needed and build a schedule, but will not be able to save classes taken or the schedule they build to an account. However, the save button will be omitted from the page.
- Main page: The main page will have a toolbar at the top with a “Classes” tab, “Schedule” tab, and “Profile tab”. By default, the Classes tab will display.
- Classes: This page will show a list of courses for the user’s degree program(s). Next to the list, a tree of classes for their program(s) will be displayed. The tree div will be tabbed, so if a user has multiple programs, there will be a different tab for each program.

MySQL Database

- 2 Hours
- There will be nine tables in the database: **User**, **Class**, **UserClasses**, **DegreeProgram**, **DegreeClasses**, **Department**, **Prerequisite**, **Corequisite**, and **Instructor**.
- The **User** table will be used for authentication and keeping track of what educational track they are on (namely, their major(s) and minor(s)). It will contain a user ID, the

name of the user, a hashed password, the ID of their major degree(s), the ID of their minor degree(s), and their email.

- The **Class** table will be used to keep a list of classes that USC has to offer for the upcoming semester. The columns it contains are the class ID, the department, the class number, section, session, type, start time, end time, days, number of registered, maximum number of registered, instructor, location, syllabus (if present), other information (if present)
- The **UserClasses** table will be used to keep a track of any given user's classes. It will contain a user's class ID that is unique to that relationship, as well as a user ID, a class ID
- The **DegreeProgram** table allows us to keep track of the list of all the degrees possible at USC, so the only columns it contains are the degree ID and the name of the degree. This will include minors.
- The **DegreeClasses** table allows us to correlate any given class with a degree program. There may be more than one degree program per class. The columns it contains are a degree's class ID, the degree ID, and the class ID.
- The **Department** table contains a list of all the departments USC has (for example CSCI or BISC or PHIL). For that reason, the only fields it contains are the department ID and the name of the department.
- The **Prerequisite** table contains a list of prerequisites for various classes, so the fields are the prerequisite ID, the class ID, and the ID of prerequisite for the class given by the class ID.
- The **Prerequisite** table contains a list of corequisites for various classes, so the fields are the corequisite ID, the class ID, and the ID of corequisite for the class given by the class ID.
- The **Instructor** table contains a list of instructors that teach at USC. The columns it contains are the instructor ID and the link to their RateMyProfessor profile if it exists.

MySQL Backend Driver

- 4 Hours
- Create a JDBCDriver class
- This is similar to the lab solution
- Class should be static and support:
    - Connecting
    - Closing
    - Adding users
    - Updating users
    - Updating user schedules
    - Any add, delete, and update operations deemed necessary along the way

Web Scraping

- 8 Hours
- Various USC websites will be scraped and their information imported into the database.
- From USC's catalogue website, catalogue.usc.edu/, the following information can be provided:
    - Degree programs and the classes required for completion of the degree.
- From USC's classes website, classes.usc.edu/, the following information can be provided:
    - Class information: name, section, session, type, start time, end time, days, number of registered, maximum number of registered, instructor, location, syllabus (if present), other information (if present), prerequisites (if present), corequisites (if present).

Backend Servlets

- 6 Hours
- Create a Controller package. Have a servlet for each page.
    - GET should redirect to its corresponding JSP
    - POST should be used to submit data to the server
- RegisterServlet:
    - Verify user input is valid.
        - No empty fields, minimum length met, special characters, etc.
        - No duplicate usernames and emails
    - If data is invalid:
        - Send error message
        - Do not redirect to <Home>
    - If all data is valid:
        - Create a new User
            - Add new User to the database
            - Add user to session
            - Redirect to Classes page
- LoginServlet:
    - Verify user input is valid.
        - No empty fields, minimum length met, special characters, etc.
        - Authenticate username and password with database entries
    - If user is authenticated:
        - Add user to session
        - Redirect to Classes page

- LogoutServlet:
    - If user has unsaved changes, give option to save
        - If yes, update the database
- ClassesServlet:
    - GET should get a list of all the classes for the user's program and redirect to the classes JSP
- ScheduleServlet:
    - GET should get the user's saved schedule from the database and redirect to the schedule JSP
    - POST should update the user's schedule and the changes should be reflected in the database
        - This can include adding or removing a class
- ProfileServlet:
    - GET should get the user's first name, last name, schedule (if public) and redirect to the profile JSP
    - POST should update an personal changes the user has made and the changes should be reflected in the database
        - This can include changing the user's email, program, etc.
- On login page, we need to be able to pass in user-inputted username and password to a servlet in order to verify credentials.

Multithreading

- 8 Hours

*The multi-threading requirement is now going to be fulfilled by adding the feature of users being able to follow another user's schedule and being able to view and edit it together.

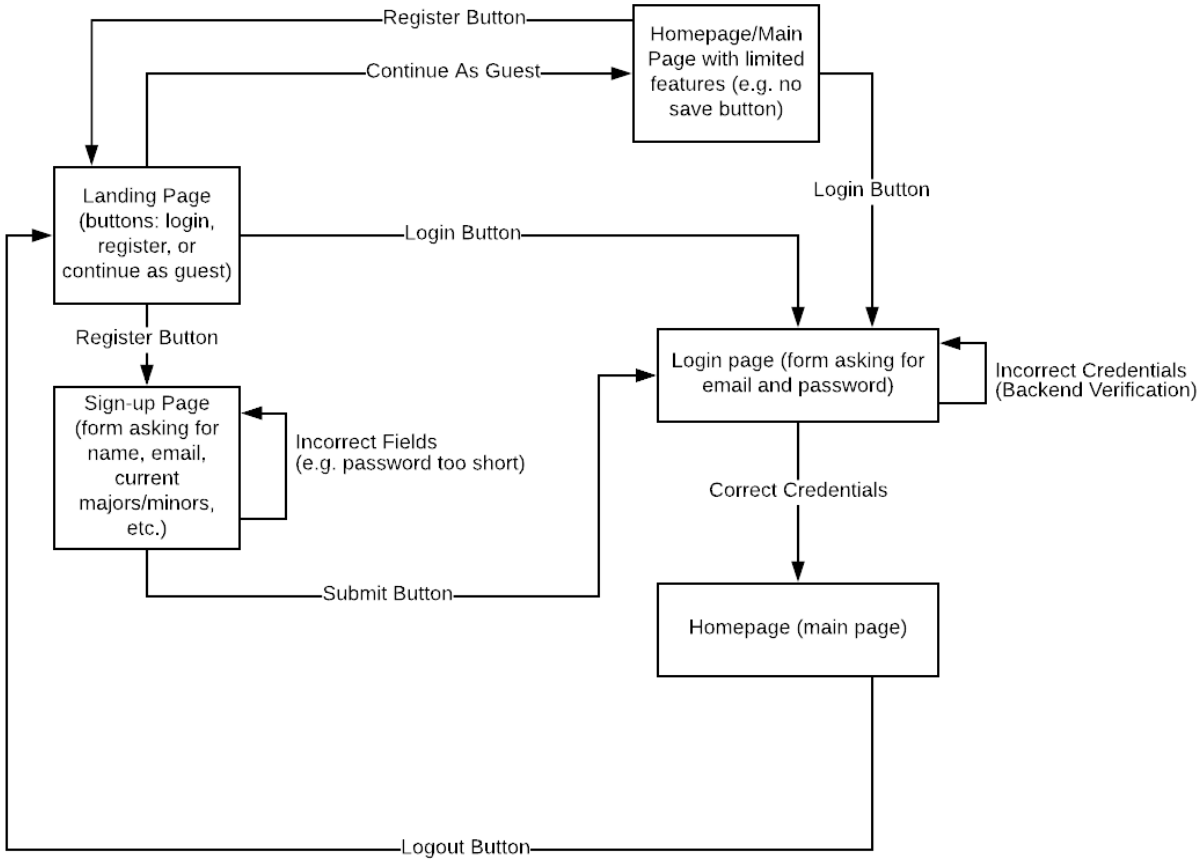
Networking

- 2 Hours
- Websockets are used for two-way communication between the client and the server
- Course plan updates are pushed through the websocket from client to server when the user is interacting with the website
- The server can also push course plan updates through the websocket to the client when it receives updates from other clients.
- Clients that are viewing other people's course plans will get live updates via the websocket communication

Detailed Design

All of the specifications in the Technical Specifications document are designed out.

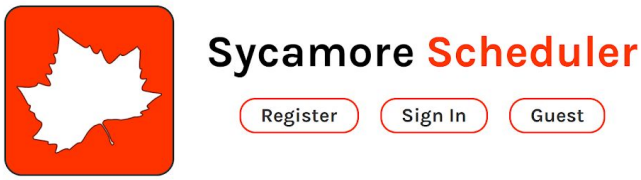
Site Map



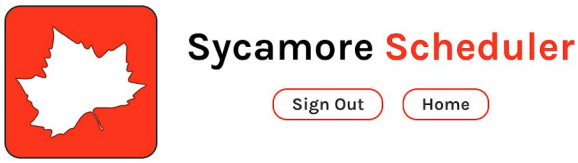
Note: After reaching the homepage/main page, buttons will no longer redirect the user to another page (i.e. the homepage will be one dynamic webpage). The only page redirection from the homepage is the logout button which will redirect the user back to the registration/login page.

GUI Mockup

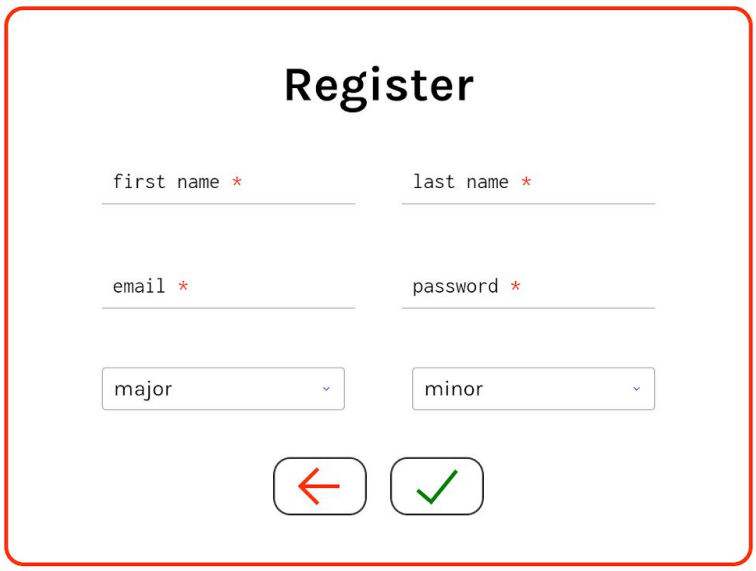
(Refer to site map above for details on how to get from one page to another):



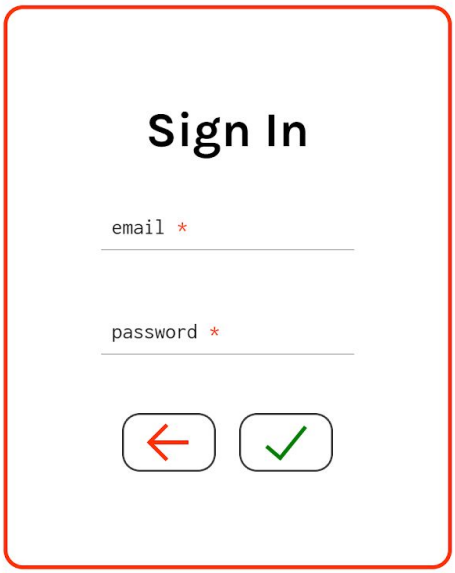
Landing Page



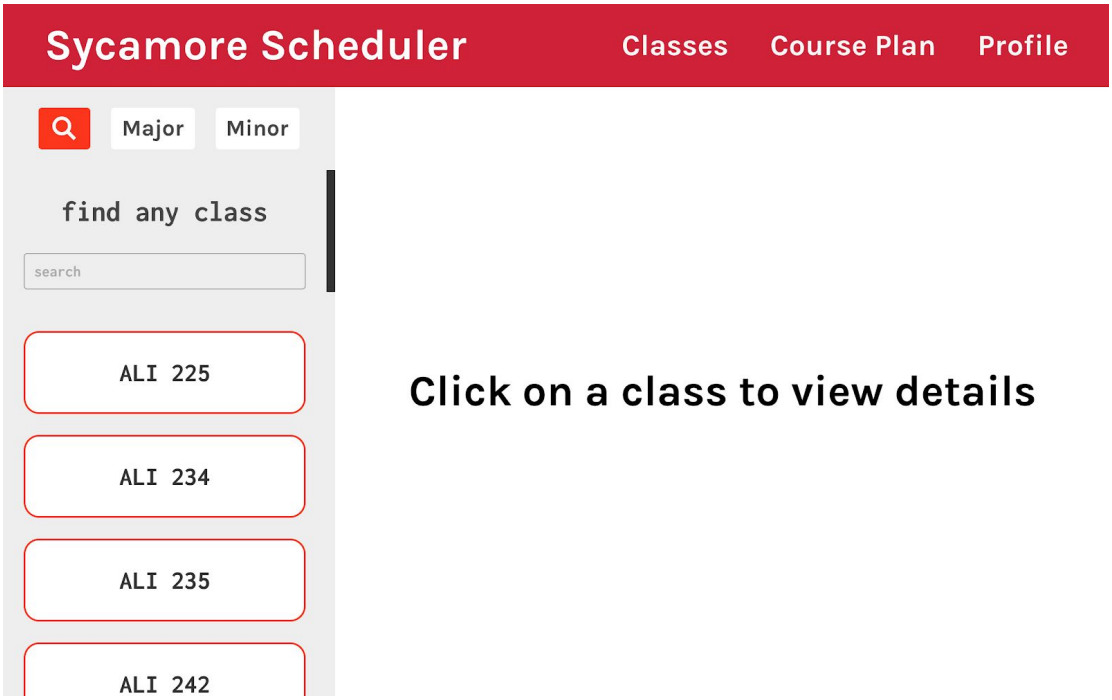
Landing Page (signed in)



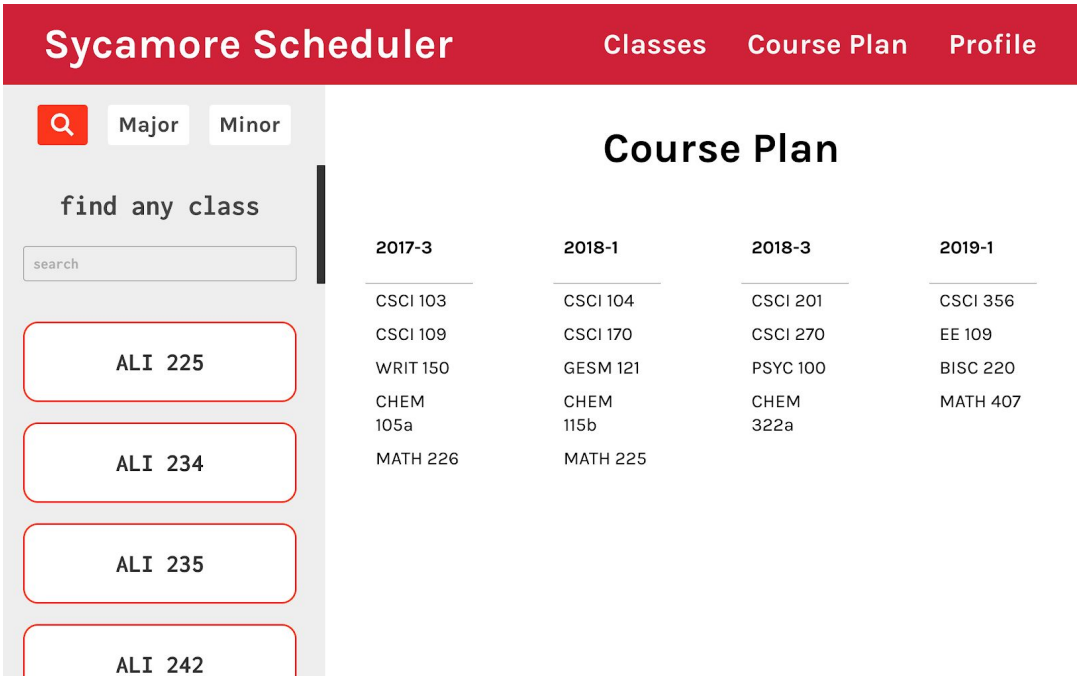
Register Page



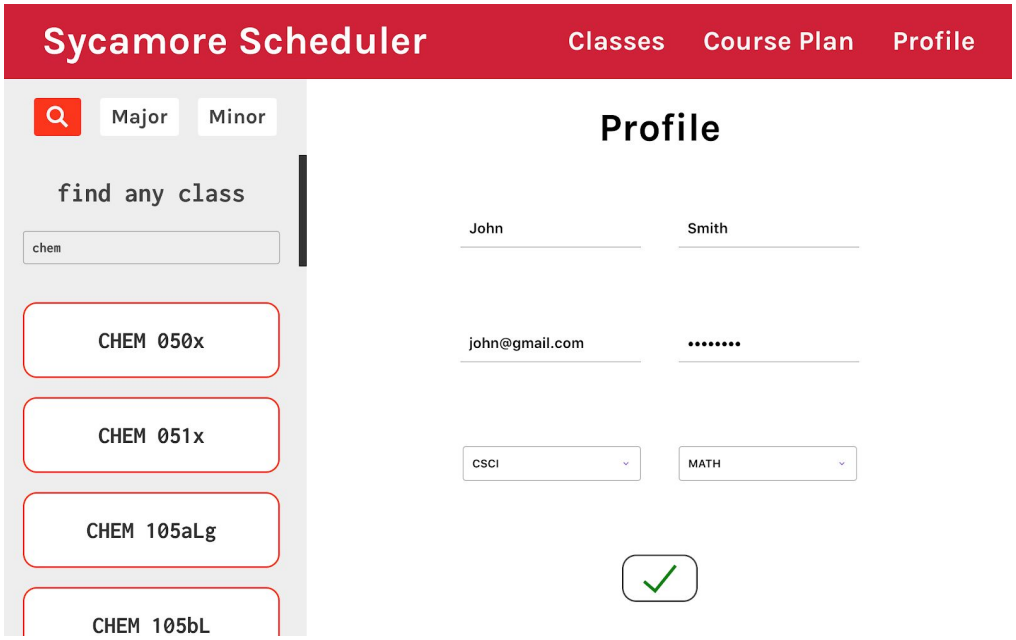
Sign In Page



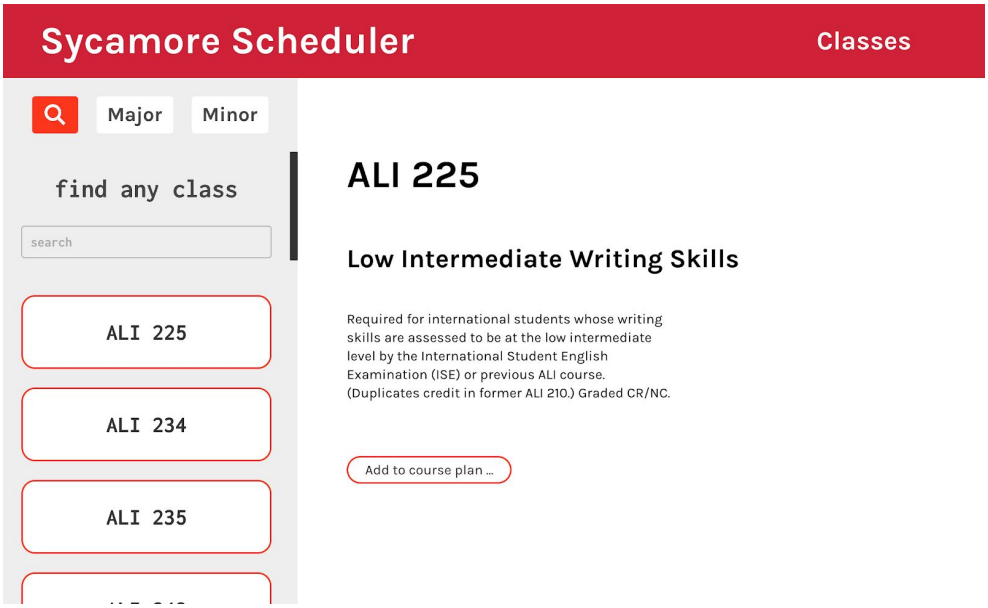
Home Page (Classes tab) for Signed In Users



Home page (Schedule tab) for signed in users

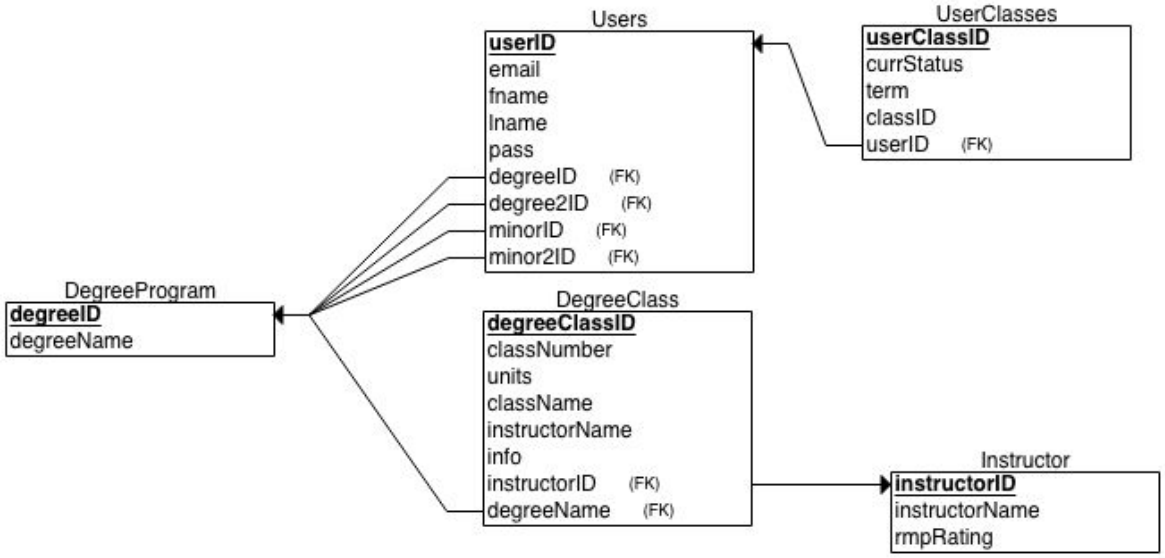


Home page (Profile tab) for signed in users



Home page (Classes tab) **for guest users only**

Database Schema



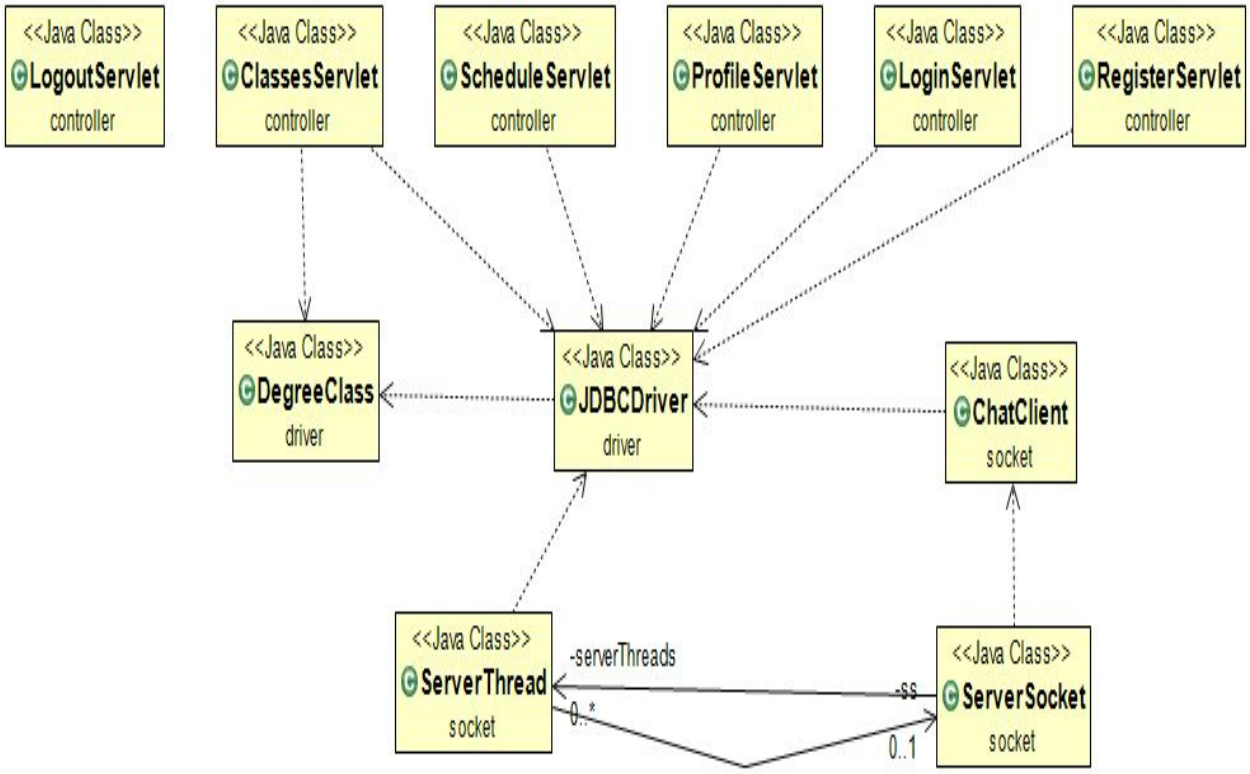
*Add degreeName to UserClasses table. This will simplify the JDBCDriver.getSchedule() method so only classes for a specific degree program are returned.
**Add following table to keep track of which users can access another user's schedule. There will be a a user id as the primary key, and then a column for another user, and a column that is 1 if the other user can access the first user's schedule, 0 if they cannot (this is similar to Assignment 3)

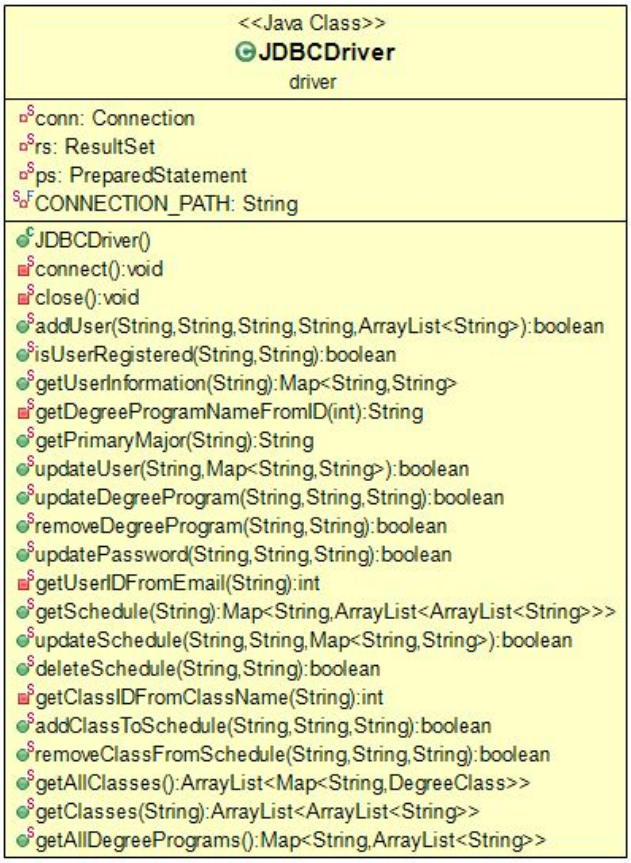
Hardware and Software Requirements

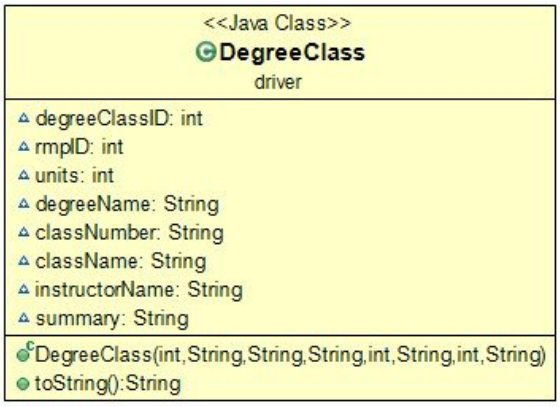
- Hardware: Nothing Specific
- Tomcat Server
- Java and Java Servlets
- GSON
- MySQL JDBC Driver 8
- Python (Scraping)
- JavaScript
    - If you want to build the static HTML from React.js source, you need npm
- Eclipse
- Objectaid
- Maven

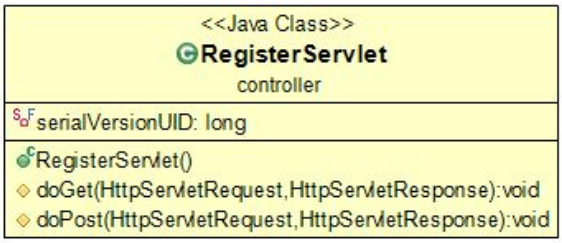
Class Diagram and/or Inheritance Hierarchy

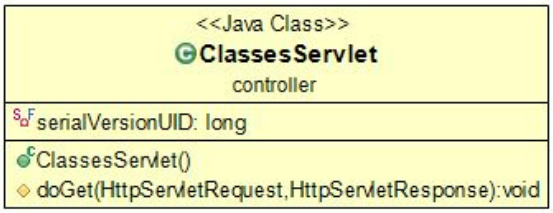
- Package: driver
    - JDBCDriver
    - DegreeClass
- Package: controller
    - RegisterServlet
    - LoginServlet
    - LogoutServlet
    - ClassesServlet
    - ScheduleServlet
    - ProfileServlet
- Package: socket
    - ChatClient
    - ServerSocket
    - ServerThread

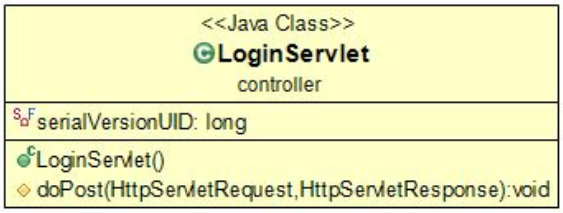


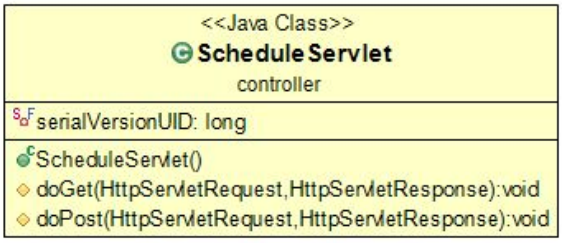

<<Java Class>>
JDBCDriver
driver
conn: Connection
rs: ResultSet
ps: PreparedStatement
CONNECTION_PATH: String
JDBCDriver()
connect():void
close():void
addUser(String,String,String,String,ArrayList<String>):boolean
isUserRegistered(String,String):boolean
getUserInformation(String):Map<String,String>
getDegreeProgramNameFromID(int):String
getPrimaryMajor(String):String
updateUser(String,Map<String,String>):boolean
updateDegreeProgram(String,String,String):boolean
removeDegreeProgram(String,String):boolean
updatePassword(String,String,String):boolean
getUserIDFromEmail(String):int
getSchedule(String):Map<String,ArrayList<ArrayList<String>>>
updateSchedule(String,String,Map<String,String>):boolean
deleteSchedule(String,String):boolean
getClassIDFromClassName(String):int
addClassToSchedule(String,String,String):boolean
removeClassFromSchedule(String,String,String):boolean
getAllClasses():ArrayList<Map<String,DegreeClass>>
getClasses(String):ArrayList<ArrayList<String>>
getAllDegreePrograms():Map<String,ArrayList<String>>

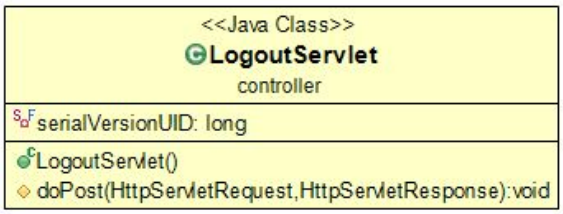

<<Java Class>>
DegreeClass
driver
degreeClassID: int
rmpID: int
units: int
degreeName: String
classNumber: String
className: String
instructorName: String
summary: String
DegreeClass(int,String,String,String,int,String,int,String)
toString():String

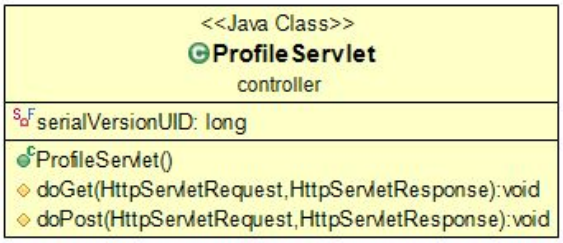

<<Java Class>>
RegisterServlet
controller
serialVersionUID: long
RegisterServlet()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void

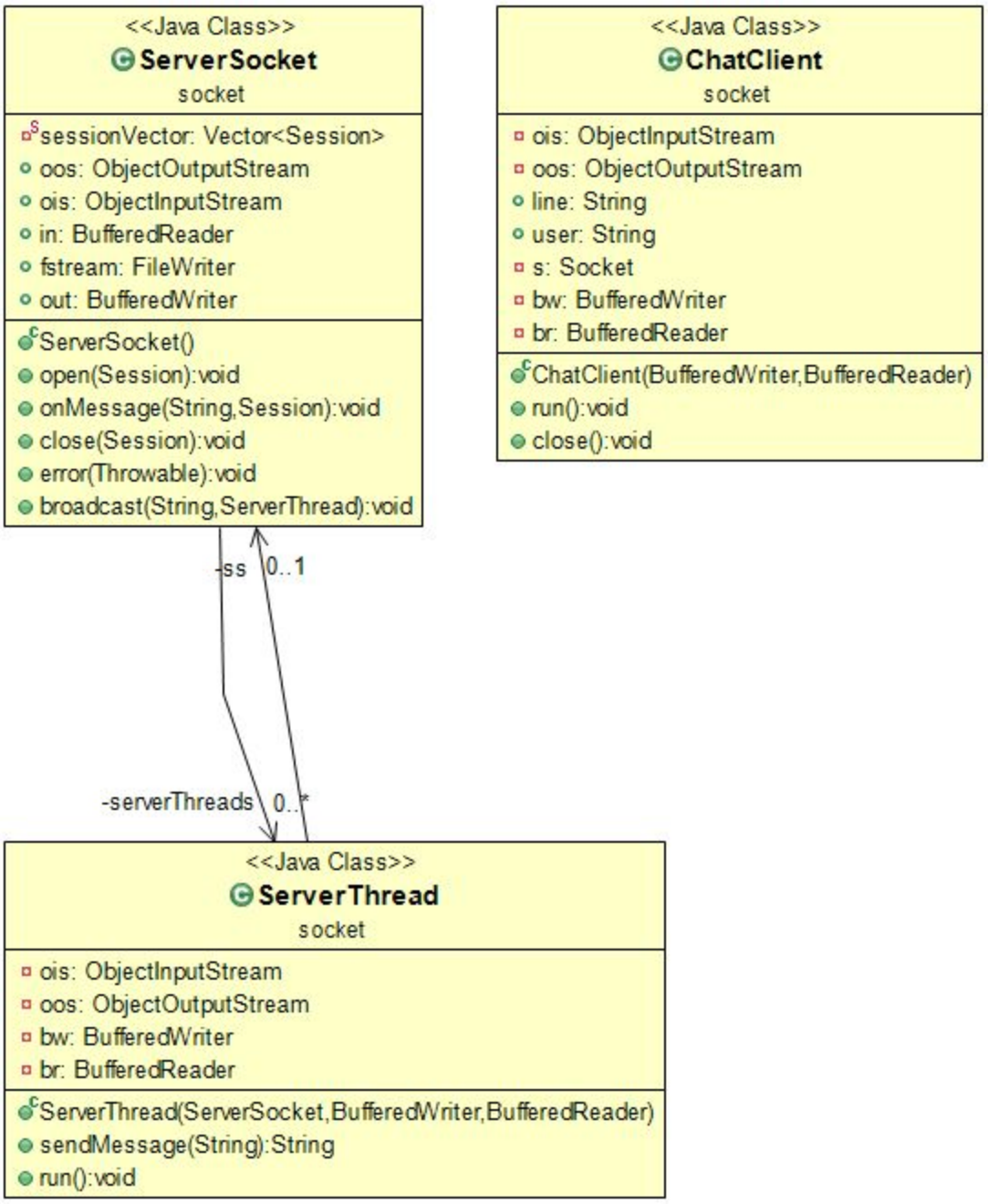

<<Java Class>>
ClassesServlet
controller
serialVersionUID: long
ClassesServlet()
doGet(HttpServletRequest,HttpServletResponse):void


<<Java Class>>
LoginServlet
controller
serialVersionUID: long
LoginServlet()
doPost(HttpServletRequest,HttpServletResponse):void


<<Java Class>>
ScheduleServlet
controller
serialVersionUID: long
ScheduleServlet()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void


<<Java Class>>
LogoutServlet
controller
serialVersionUID: long
LogoutServlet()
doPost(HttpServletRequest,HttpServletResponse):void


<<Java Class>>
ProfileServlet
controller
serialVersionUID: long
ProfileServlet()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void


<<Java Class>>
ServerSocket
socket
sessionVector: Vector<Session>
oos: ObjectOutputStream
ois: ObjectInputStream
in: BufferedReader
fstream: FileWriter
out: BufferedWriter
ServerSocket()
open(Session):void
onMessage(String,Session):void
close(Session):void
error(Throwable):void
broadcast(String,ServerThread):void
<<Java Class>>
ChatClient
socket
ois: ObjectInputStream
oos: ObjectOutputStream
line: String
user: String
s: Socket
bw: BufferedWriter
br: BufferedReader
ChatClient(BufferedWriter,BufferedReader)
run():void
close():void
-ss
0..1
-serverThreads
0..*
<<Java Class>>
ServerThread
socket
ois: ObjectInputStream
oos: ObjectOutputStream
bw: BufferedWriter
br: BufferedReader
ServerThread(ServerSocket,BufferedWriter,BufferedReader)
sendMessage(String):String
run():void

Key Algorithms

- The JDBCDriver class will be responsible for accessing a MySQL database. This class will handle a lot of the operations that the program requires
    - Because of this, most classes except for the LogoutServlet class will require access to this class
- The RegisterServlet class will access the Users table in a MySQL database using the JDBCDriver class, which uses the jdbc:mysqldriver
    - The RegisterServlet.doGet() method is now implemented to retrieve all possible degree programs when a user is trying to register
    - The boolean addUser() method will take five parameters:
        - four strings representing the email, first name, last name, and password
        - one ArrayList<String> representing the academic programs the user is enrolled in
        - returns true if a user is successfully added to the database and false otherwise.
- The LoginServlet class will access the Users table in a MySQL database using the JDBCDriver class, which uses the jdbc:mysqldriver
    - The isUserRegistered() method will take two strings as parameters, representing the email and password
        - It will return a boolean value specifying whether the user was authenticated or not against the Users table
- The LogoutServlet class will
    - Sign the user out and redirect to the home page
- The ClassesServlet class will access the Class table in a MySQL database using the JDBCDriver class, which uses the jdbc:mysqldriver
    - The getClasses() method will take one string as a parameter, representing the department name
        - It will return an ArrayList<ArrayList<String>> representing the information for the department's classes
- The ScheduleServlet class will access the UserClasses table in a MySQL database using the JDBCDriver class, which uses the jdbc:mysqldriver
    - The getSchedule() method will take two strings as parameters, representing the email and degree program name
        - It will return an ArrayList<ArrayList<String>> representing the user's schedule
    - The ScheduleServlet will also handle updates to the student's schedule.

        - The front-end will pass an action (a string variable) either being “add” or “remove”
            - If the action is add, we call on the JDBCDriver’s addClassToSchedule() method which takes three strings as parameters, representing the email, class name, and degree program name
            - If the action is remove, we call on the JDBCDriver’s removeClassFromSchedule() method which takes three strings as parameter, representing the email, class name, and degree program name
- The ProfileServlet class will access the Users and UserClasses table in a MySQL database using the JDBCDriver class, which uses the jdbc:mysqldriver
    - The getSchedule() method will take two strings as parameters, representing the email and degree program name
        - It will return an ArrayList<ArrayList<String>> representing the user’s schedule
    - The boolean updateUser() method will take two parameters
        - One string representing the email
        - One Map<String, String> representing the key-value pairs of the information to update
        - Returns true if a user’s information is successfully updated on the database and false otherwise.
- In general, all servlets now follow these guidelines:
    - Set the HttpServletResponse status.
    - Get the PrintWriter and write the success/error message or send the data from the database.
        - This information is sent as a JSON.

Testing

White Box Testing

- Test the login functionality (invalid credentials). Specify a username that exists and a password that doesn’t exist and vice versa (the user should stay at the login page with a “invalid login” message on their screen).

```
// User.login(username, password) attempts to log a user in with the given
    // credentials
// expect false, because wrong passwords are rejected
boolean loginBadPassword() {
```

```
        return User.login("existing_user", "wrong_password");
    }
    // expect false, because wrong passwords are rejected
    boolean loginBadUsername() {
        return User.login("nonexistant_user", "any_password");
    }
```

- Test the login functionality (valid credentials). Specify a username and password that exists in the database (the user should be redirected to the main homepage).

```
    // User.login(username, password) attempts to log a user in with the given
        // credentials
    // expect true, because the username and password belong to an existing user
    boolean loginGoodCredentials() {
        return User.login("existing_user", "correct_password");
    }
```

- Test the logout button: on the main homepage, clicking on the logout button should redirect the user to the landing page. The user should not be logged into their accounts when the "guest" or "login" button are pressed.

```
    // User.isLoggedIn() checks if the user session is currently logged in
    // expect true, because the username and password belong to an existing user
    boolean logoutTest() {
        User.logout();
        return User.isLoggedIn();
    }
```

- Test the registration page (invalid fields). On the registration form, entering a short password (e.g. less than 6) or not having at least one special character (e.g. @!",) should bring the user to the same registration page with an error message to re-enter their password. This should also apply for the username if the user does not meet the username requirements (e.g. too short).

```
    // User.register(username, password, email) attempts to register a user with the
        // given information passed into the function
    // expect false, because password should be no shorter than 6 characters
    boolean registerShortPassword() {
        return User.register("username", "pass!", "valid_email@usc.edu");
    }
    // expect false, because password should contain at least one symbol
    boolean registerNoSymbolsPassword() {
        return User.register("username", "password", "valid_email@usc.edu");
    }
```

- Test the registration page (invalid email). If the user does not have exactly one "@" sign and a period in the email field, then the user should stay on the same registration page with an "invalid email address" message.

```
// expect false, because email should contain exactly one "@" sign
boolean registerNoAtInEmail() {
        return User.register("username", "password!", "valid_emailusc.edu");
}
// expect false, because email should contain exactly one "@" sign
boolean registerMoreAtsInEmail() {
        return User.register("username", "password!", "valid@email@usc.edu");
}
```

- Test changes in profile picture. When the user updates their profile picture, ensure that the new change is reflected on all pages. The given URL must also be valid.

```
// User.updatePicture(urlLink) attempts to update the profile picture with the
        // picture in the URL link
// User.getPicture() return the URL string of the current URL picture
// expect false, because picture must be updated from a valid URL
boolean pictureNonexistantUpdate() {
        return User.updatePicture("http://notavalidsite!!!.com/nopicture.jpg")
}
// expect true, because picture change should be reflected on all pages
boolean pictureNonexistantUpdate() {
        String newPicture = "http://www.shawnee.edu/_resources/images/profile-
                placeholder.png";
        User.updatePicture(newPicture);
        return User.getPicture().equals(newPicture);
}
```

- Test changes in password. The user should be able to log out and log back in with their new password.

```
// User.changePassword(password) attempts to update the current password with
        // the password passed in
// User.getPassword() return the password of the user
// expect false, because password should be greater than or equal to 6 characters
boolean changeShortPassword() {
        return User.updatePassword("pass@");
}
// expect false, because password should contain more than zero symbols
boolean changeNoSymbolsPassword() {
        return User.updatePassword("mypassword");
```

```
}
// expect true while logged in, because password change should work
boolean changeGoodPassword() {
        String newPassword = "password@";
        return User.updatePassword(newPassword);
        return User.getPassword().equals(newPassword);
}
```

Black Box Testing

- Test the landing page: clicking on the "sign up" button should redirect the user to a page with a username, password, email address, and name form (user should also be allowed to select their majors and minors). Clicking on the "login" button should redirect the user to a page with a username and password form.
- Test the "continue as guest" feature: at the landing page, clicking on the "guest" button should redirect the user to the main homepage for guests (e.g. no save button). Ensure that this guest homepage is not the same homepage for users with an account.
- Test the guest user functionality by making sure they can't save their schedules. Test that guest users can view classes, see schedule, and export schedule
- Test the export feature. The user should be able to download a version of their course plan onto their desktop by clicking on the "export" or "download" button.
- Test changes in major/minor. Start with a user in the CSCI major, and ensure the "courses" tab displays the correct CSCI catalogue. Click the profile tab, and change the major to BUAD. Then click on the "courses" tab. The courses should now be BUAD classes.
- Test the courses tab. Click 'Courses' on the main navbar. You should see the courses for your specific major, or a search tab if you did not input any major or you are a guest.
- Test the profile tab. Click 'Profile' on the main navbar. You should see the profile information including name, email, username, major/minor/degree program, and courses taken.
    - Test the functionality. You should be able to change any of these fields and it should be persistent upon refresh of the page.
- Test the course plan tab. Click 'Course Plan' on the main navbar. You should see a scrollable div of tables that each represent a term of the student's academic career until the expected graduation date (default = 4 years from start term [start term default = this term]). The tables should be filled in with any courses that are in the 'Added' status in the courses page.
    - Subtest: test 'Added' status. Add a course in the courses tab. Switch to the course plan tab, and that course should now appear. Now click the 'x' button next to the

course. Verify that the course no longer appears in the course plan table AND that it is no longer of ‘Added’ status in the courses tab.

Unit Testing

Package: driver

- Insert SQL code in the email variable of the JDBCDriver.addUser() method. Verify that the SQL code specified is not executed against the database. Repeat with the other parameters which are first name, last name, and password.

```
void emailSQLI() {
    JDBCDriver.addUser(“test; DROP DATABASE IF EXISTS SCHEDULER --”, “FirstName”, “LastName”, “password!”, new ArrayList<String>(“CSCI”));
}
// Then manually testing to check if any damage was caused

void firstNameSQLI() {
    JDBCDriver.addUser(“email”, “test; DROP DATABASE IF EXISTS SCHEDULER --”, “LastName”, “password!”, new ArrayList<String>(“CSCI”));
}
// Then manually testing to check if any damage was caused

void lastNameSQLI() {
    JDBCDriver.addUser(“email”, “FirstName”, “test; DROP DATABASE IF EXISTS SCHEDULER --”, “password!”, new ArrayList<String>(“CSCI”));
}
// Then manually testing to check if any damage was caused

void passwordSQLI() {
    JDBCDriver.addUser(“email”, “FirstName”, “LastName”, “test; DROP DATABASE IF EXISTS SCHEDULER --”, new ArrayList<String>(“CSCI”));
}
// Then manually testing to check if any damage was caused
```

- Test the JDBCDriver.addUser() method with an invalid email, first name, last name, password, and list of academic programs. Verify that the user is not added to the database and that it returns false.

```
void createUserMalformedEmail() {
```

```
        // email must have at least an @ sign
        JDBCDriver.addUser("email", "FirstName", "LastName", "password!",
    new ArrayList<String>("CSCI"));
    }
    void createUserMalformedFirstName() {
        JDBCDriver.addUser("email@gmail.com", "Really Long Name For
    Testing An Overflow Of The FirstName Column", "LastName", "password!",
    new ArrayList<String>("CSCI"));
    }
    void createUserMalformedLastName() {
        JDBCDriver.addUser("email@gmail.com", "First Name", "Really Long
    Name For Testing An Overflow Of The LastName Column", "password!", new
    ArrayList<String>("CSCI"));
    }
    void createUserMalformedProgramList() {
        JDBCDriver.addUser("email@gmail.com", "First Name", "Last Name",
"password!",
    "Not an ArrayList<String>");
    }
```

- Test the JDBCDriver.addUser() method with an existing user's email. Verify that the user is not added to the database again and that it returns false.

```
    // expect false, because user should not be created again
    boolean createUserExistingEmail() {
        return JDBCDriver.addUser("existing_email@gmail.com", "First Name",
    "Last Name", "password!", new ArrayList<String>("CSCI"));
    }
```

- Test the JDBCDriver.addUser() method with a valid email, first name, last name, password, and list of academic programs. Verify that the user is added to the database by getting a list of users with size + 1 and that it returns true.

```
    // expect true, because user should be created
    boolean createUserAllValid() {
        return JDBCDriver.addUser("email@gmail.com", "First Name",
    "Last Name", "password!", new ArrayList<String>("CSCI"));
    }
```

- Insert SQL code in the email variable of the JDBCDriver.isUserRegistered() method. Verify that the SQL code specified is not executed against the database.
    - Attempt the exact above code (JDBCDriver.addUser() SQLI checks) for JDBCDriver.isUserRegistered() using email and password for parameters.

- Test the JDBCDriver.isUserRegistered() method with an invalid email, then password. Verify that it returns false for each.
    - Attempt the exact above code (JDBCDriver.addUser() invalid parameter checks) for JDBCDriver.isUserRegistered() using email and password for parameters.
- Test the JDBCDriver.isUserRegistered() method with a valid email and password. Verify that it returns true.
    - Attempt the exact above code (JDBCDriver.addUser() valid check) for JDBCDriver.isUserRegistered() using email and password for parameters.
- Test the JDBCDriver.isUserRegistered() method with a valid email in lowercase, uppercase, and mixed case. Verify that it returns true.
    - Attempt the exact above code (JDBCDriver.addUser() valid check) for JDBCDriver.isUserRegistered() using email and password for parameters in lowercase.
- Insert SQL code in the email variable of the JDBCDriver.updateUser() method. Verify that the SQL code specified is not executed against the database. Repeat with the other parameter which is a map of the key-value pairs of the information to update.
    - Attempt the exact above code (JDBCDriver.addUser() SQLI checks) for JDBCDriver.updateUser() using email and map for parameters.
-
- Insert SQL code in the email variable of the JDBCDriver.getSchedule() method. Verify that the SQL code specified is not executed against the database.
    - Attempt the exact above code (JDBCDriver.addUser() SQLI checks) for JDBCDriver.getSchedule() using email and degree program for parameters.
- Test the JDBCDriver.getSchedule() method with an invalid email and degree program name. Verify that it returns null.
    - Attempt the exact above code (JDBCDriver.addUser() invalid checks) for JDBCDriver.getSchedule() using email and degree program for parameters.
- Test the JDBCDriver.getSchedule() method with a valid email and degree program name. Verify that it returns an Map<String, ArrayList<ArrayList<String>>> that represents the user's schedule information.
    - Attempt the exact above code (JDBCDriver.addUser() valid check) for JDBCDriver.getSchedule() using email and degree program for parameters.
- Insert SQL code in the email variable of the JDBCDriver.updateSchedule() method. Verify that the SQL code specified is not executed against the database. Repeat with the other parameters which are the degree program name and a map of the key-value pairs of the information to update.
    - Attempt the exact above code (JDBCDriver.addUser() SQLI checks) for JDBCDriver.updateSchedule() using email, degree program, and map of changes for parameters.

- Test the JDBCDriver.updateSchedule() method with an invalid email, degree program name, and map of changes to be made. Verify that it returns false.
    - Attempt the exact above code (JDBCDriver.addUser() invalid checks) for JDBCDriver.updateSchedule() using email, degree program, and map of changes for parameters.
- Test the JDBCDriver.updateSchedule() method with a valid email, degree program name, and map of changes to be made. Verify that the new information overrides the old information in the database by comparing two lists (old and new) of the information and it returns true.
    - Attempt the exact above code (JDBCDriver.addUser() valid check) for JDBCDriver.updateSchedule() using email, degree program, and map of changes for parameters.
- Insert SQL code in the email variable of the JDBCDriver.deleteSchedule() method. Verify that the SQL code specified is not executed against the database. Repeat with the other parameter which is the degree program name.
    - Attempt the exact above code (JDBCDriver.addUser() SQLI checks) for JDBCDriver.deleteSchedule() using email and degree program for parameters.
- Test the JDBCDriver.deleteSchedule() method with an invalid email and degree program name. Verify that the database entries are not deleted and that it returns false.
    - Attempt the exact above code (JDBCDriver.addUser() invalid checks) for JDBCDriver.deleteSchedule() using email and degree program for parameters.
- Test the JDBCDriver.deleteSchedule() method with a valid email and degree program name. Verify that the database entries are deleted and that it returns true.
    - Attempt the exact above code (JDBCDriver.addUser() valid check) for JDBCDriver.deleteSchedule() using email and degree program for parameters.
- Insert SQL code in the degree program name variable of the JDBCDriver.getClasses() method. Verify that the SQL code specified is not executed against the database.
    - Attempt the exact above code (JDBCDriver.addUser() SQLI checks) for JDBCDriver.getClasses() using the degree program for the parameter.
- Test the JDBCDriver.getClasses() method with an invalid degree program name. Verify that it returns null.
    - Attempt the exact above code (JDBCDriver.addUser() invalid checks) for JDBCDriver.getClasses() using the degree program for the parameter.
- Test the JDBCDriver.getClasses() method with a valid degree program name. Verify that it returns an ArrayList<ArrayList<String>> that represents the classes for the given degree program name.
    - Attempt the exact above code (JDBCDriver.addUser() valid check) for JDBCDriver.getClasses() using the degree program for the parameter.

Package: controller

- Test the RegisterServlet.doGet() method. Verify that it sends a success message to the front-end with the degree program names as a JSON.

```
var test = function ( ) {
        $.get({
                url: '/RegisterServlet',
                data: {
                        // empty
                },
                success: (data) => {
                        alert(JSON.stringify(data)); // check this
                },
                error: (err) => {
                        alert(err);
                }
        });
}
```

- Test the RegisterServlet.doPost() method with an empty/null email, first name, last name, and password. Verify that it sends an error message to the front-end and it doesn't call the JDBCDriver.addUser() method.

```
var test = function ( ) {
        $.post({
                url: '/RegisterServlet',
                data: {
                        // empty
                },
                success: (message) => {
                        alert(message);
                },
                error: (err) => {
                        alert(err); // this should alert
                }
        });
}
```

- Test the RegisterServlet.doPost() method with an existing user's email. Verify that it calls the JDBCDriver.addUser() method which will return false and that it sends an error message to the front-end.

```
var test = function ( ) {
        $.post({
                url: '/RegisterServlet',
```

```
                data: {
                        email: ‘existinguser@gmail.com’
                },
                success: (message) => {
                        alert(message);
                },
                error: (err) => {
                        alert(err); // this should alert
                }
        });
}
```

- Test the RegisterServlet.doPost() method with a valid email, first name, last name, and password. Verify that it calls the JDBCDriver.addUser() method which will return true and that it sends a success message to the front-end.

```
var test = function ( ) {
        $.post({
                url: ‘/RegisterServlet’,
                data: {
                        email: ‘validemail@gmail.com’,
                        FirstName: ‘validfirstname’,
                        LastName: ‘validlastname’
                },
                success: (message) => {
                        alert(message);
                },
                error: (err) => {
                        alert(err); // this should alert
                }
        });
}
```

- Test the LoginServlet.doPost() method with an empty/null email and password. Verify that it calls the JDBCDriver.isUserRegistered() method with will return false and that it sends an error message to the front-end.

```
var test = function ( ) {
        $.post({
                url: ‘/LoginServlet’,
                data: {
                        // empty
```

```
                },
                success: (message) => {
                        alert(message);
                },
                error: (err) => {
                        alert(err); // this should alert
                }
        });
}
```

- Test LoginServlet.doPost() method with a valid email and password. Verify that it calls the JDBCDriver.isUserRegistered() method which will return true and that it sends a success message to the front-end.

```
var test = function ( ) {
        $.post({
                url: '/LoginServlet',
                data: {
                        email: 'validemail@gmail.com'
                },
                success: (message) => {
                        alert(message); // this should alert
                },
                error: (err) => {
                        alert(err);
                }
        });
}
```

- Test the LogoutServlet.doPost() method. Verify that it sends a success message to the front-end after setting all session attributes to null.

```
var test = function ( ) {
        $.post({
                url: '/LogoutServlet',
                data: {
                        // nothing needed
                },
                success: (message) => {
                        alert(message); // this should alert
                },
                error: (err) => {
                        alert(err);
```

```
            }
        });
    }
```

- Test the ClassesServlet.doGet() method() with an empty/null degree program name. Verify that the JDBCDriver.getClasses() method returns null and that it sends an error message to the front-end.

```
var test = function ( ) {
        $.get({
            url: '/ClassesServlet',
            data: {
                // empty
            },
            success: (message) => {
                alert(message);
            },
            error: (err) => {
                alert(err); // this should alert
            }
        });
}
```

- Test the ClassesServlet.doGet() method with a valid degree program name. Verify that the JDBCDriver.getClasses() method returns an ArrayList<ArrayList<String>> representing the degree program name's class information and that it sends a success message to the front-end with the corresponding JSON response.

```
var test = function ( ) {
        $.get({
            url: '/ClassesServlet',
            data: {
                degree: 'CSCI'
            },
            success: (data) => {
                alert(JSON.stringify(data)); // check this
            },
            error: (err) => {
                alert(err);
            }
        });
}
```

- Test the ScheduleServlet.doGet() method with an empty/null email and degree program name. Verify that it calls the JDBCDriver.getSchedule() method which will return null and that it sends an error message to the front-end.

```
var test = function ( ) {
        $.get({
                url: '/ScheduleServlet',
                data: {
                        // empty
                },
                success: (message) => {
                        alert(message);
                },
                error: (err) => {
                        alert(err); // this should alert
                }
        });
}
```

- Test the ScheduleServlet.doGet() method with a valid email and degree program name. Verify that it calls the JDBCDriver.getSchedule() method which will return an ArrayList<ArrayList<String>> that represents the user's schedule information and that it sends a success message to the front-end with the corresponding JSON response.

```
var test = function ( ) {
        $.get({
                url: '/ScheduleServlet',
                data: {
                        degree: 'CSCI',
                        email: 'validemail@gmail.com'
                },
                success: (data) => {
                        alert(JSON.stringify(data)); // check this
                },
                error: (err) => {
                        alert(err);
                }
        });
}
```

- Test the ScheduleServlet.doPost() method with an empty/null email, degree program name, and map of changes to be made. Verify that it calls the

JDBCDriver.updateSchedule() method which will return false and that it sends an error message to the front-end.

```
var test = function ( ) {
        $.post({
                url: '/ScheduleServlet',
                data: {
                        // empty
                },
                success: (message) => {
                        alert(message);
                },
                error: (err) => {
                        alert(err); // this should alert
                }
        });
}
```

- Test the ScheduleServlet.doPost() method with a valid email, degree program name, and map of changes to be made. Verify that it calls the JDBCDriver.updateSchedule() method which will return true and that it sends a success message to the front-end with the corresponding JSON response.

```
var test = function ( ) {
        $.post({
                url: '/ScheduleServlet',
                data: {
                        email: 'valid@email.com',
                        degree: 'CSCI',
                        changes: changesMap
                },
                success: (message) => {
                        alert(message); // this should alert
                },
                error: (err) => {
                        alert(err);
                }
        });
}
```

- Test the ProfileServlet.doGet() method. Verify that it sends a success message to the front-end and a JSON response with the user's information.

```
var test = function ( ) {
```

```
        $.get({
                url: '/ProfileServlet',
                data: {
                        // empty
                },
                success: (data) => {
                        alert(JSON.stringify(data)); // check this
                },
                error: (err) => {
                        alert(err);
                }
        });
}
```

- Test the ProfileServlet.doPost() method with an empty/null email and map of the changes. Verify that it calls the JDBCDriver.updateUser() method which will return false and that it sends an error message to the front-end.

```
var test = function ( ) {
        $.post({
                url: '/ProfileServlet',
                data: {
                        // empty
                },
                success: (message) => {
                        alert(message);
                },
                error: (err) => {
                        alert(err); // this should alert
                }
        });
}
```

Stress Testing

- Out of the scope for this project (e.g. we will not be having 10,000 users)

Regression Testing

- Will update if necessary as project develops

Deployment

1. Download the project folder as a .zip and expand it.
2. Navigate to the sycamore-maven directory where pom.xml is located
    a. Run “mvn package” in the console
    b. This will generate a target directory with a SycamoreScheduler.war file and deployable directory (called SycamoreScheduler)
3. Run the fullDB.sql script found in the database directory script in MySQL
    a. Ensure that the CONNECTION_PATH variable in src/driver/JDBCDriver.java is correct (Line #21 at the top of the JDBCDriver class)
    b. Start the MySQL server

There are two ways to proceed.

1. Using Visual Studio Code
    a. Open the sycamore-scheduler directory in vscode. Make sure to have the “Tomcat for Java” and “Maven for Java” extensions installed.
    b. The Maven for Java extension should automatically recognize the pom.xml file located in sycamore-maven. Right click on this project and run clean, followed by package.
    c. Right click on the SycamoreScheduler folder that is created in the sycamore-maven/target/ folder. Run this folder on the Tomcat Server of your choice.
    d. Navigate to localhost:8080/SycamoreScheduler

2. Using Eclipse

    a. In Eclipse, import the SycamoreScheduler.war file found in /sycamore-maven/target. When importing, import the following web libraries: gson-2.8.5.jar, mysql-connector-java-8.0.13.jar, protobuf-java-3.6.1.jar.

    b. Right click on the SycamoreScheduler project and run the application on the Apache Tomcat Server version 9.0.

    c. Navigate to http://localhost:8080/SycamoreScheduler/