USC Class Scheduler WebReg 2.0 - Final Documentation

Mo Jiang (mojiang@usc.edu), Breeze Pickford (bpickfor@usc.edu), Luke Albert (lpalbert@usc.edu), Jasmita Yechuri (yechuri@usc.edu), Advay Iyer (advayiye@usc.edu), Samuel Wu (samuelsw@usc.edu), Satwika Vemuri (vemurina@usc.edu), Edward Shao (shaoe@usc.edu)

University of Southern California
CSCI 201

TABLE OF CONTENTS

Project Proposal

A redesigned platform inspired by USC's Web Reg, featuring a more user-friendly and intuitive interface tailored to students' needs and demands. This platform will include additional features, such as an automatic “optimized” schedule maker. Students can input the courses they would like to take, rank their preferred class times (e.g., favoring the Friday lecture over the Wednesday lecture), and specify their ideal class time frames. The system will then generate a personalized, "optimal" schedule that aligns with their preferences and availability.

Weekly Meeting: Thursdays at 5:40 PM

Luke Albert, lpalbert@usc.edu
Mo Jiang, mojiang@usc.edu
Jasmita Yechuri, yechuri@usc.edu
Satwika Vemuri Naga Kaivalya, vemurina@usc.edu
Edward Shao, shaoe@usc.edu
Breeze Pickford, bpickfor@usc.edu
Samuel Wu, samuelsw@usc.edu
Advay Iyer, advayiye@usc.edu

High-Level Requirements

The platform will provide a user-friendly web interface for USC class registration and schedule management. Students can sign in using their email accounts to access the system. The intuitive design will ensure smooth navigation, enabling users to browse available classes effortlessly.

The platform will feature tools to create and manage custom class schedules for each semester, offering full flexibility for students to add or remove classes and allocate them to specific time slots. Additionally, the platform will include an intelligent scheduling algorithm that can automatically generate an “optimal” schedule based on the user’s selected classes and preferred timeframes. After generating a schedule, users can also make manual adjustments to tailor it further to their needs. Registered users will have the option to download their optimized schedules.

Technical Specification

I - Authentication System (12 hours)

- Implement Auth0 authentication system with Google SSO integration for ease of login
- Set up user login and registration procedures
- Implement user role management and permissions
- Set up secure token handling and verification
- Track user login status and change corresponding interface

II - Database Architecture (16 hours)

- Design and implement MySQL database schema with the following tables:
    - Classes (class_id, dept, class_code, start_time, end_time)
    - Users (id, email, user_name, password)
    - UserSchedules (schedule_id, user_id, class_id, semester)
- Implement database security measures, including:
    - Prepared statements to prevent SQL injection
    - Input validation and sanitization
    - Encrypted connections

III - Data Collection System (10 hours)

- Develop script in Java to interface with USC Schedule of Classes Web Services API
- Implement data parsing and validation
- Create an automated database population system
- Implement error handling and logging

IV - Web Interface (20 hours)

- Create a responsive front-end interface using HTML/CSS + Javascript
- Implement a class browsing interface with sorting and filtering capabilities
- Design and implement a schedule visualization system
- Create a manual schedule builder interface + Login/Register interface

V - Schedule Optimization System (24 hours)

- The Schedule Optimization System takes a list of course names (their department name and course ID (ex. CSCI201), finds all of the class sections of each of the courses, and recursively backtracks to find a schedule in which there is no overlap between any of the sections. If no such schedule is found, it will return an error.
- Implement the following constraints
    - Time conflicts

    - Course prerequisites
    - Time constraints
- Design a recursive backtracking algorithm to find a schedule of classes that does not overlap given a set of desired classes.
- Implement a dynamic programming algorithm or brute force search algorithm for schedule optimization
- Implement schedule comparison functionality

VI - Version Control, Testing, and Deployment (16 hours)

- Set up GitHub repository with proper structure
- Integrate other modules into one component
- Set up a testing framework and test the developed product

Detailed Design Document

Overview

This document describes the detailed design for a user-friendly web interface for USC class registration and schedule management. The application is designed to allow students to log in, enter the classes they want to take for the upcoming semester and view a schedule. The backend will be implemented using Java and a MySQL database, and the front end will be built using HTML/CSS + Javascript.

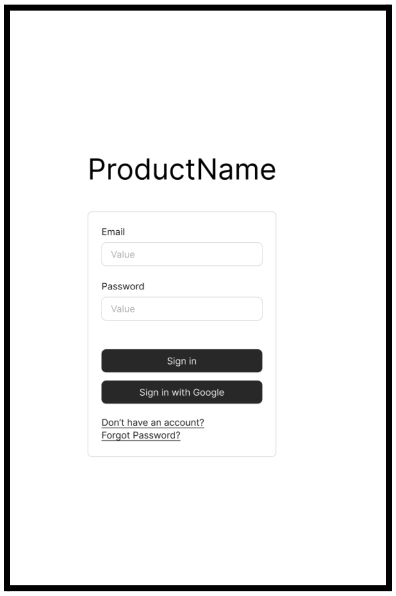
Authentication & Security

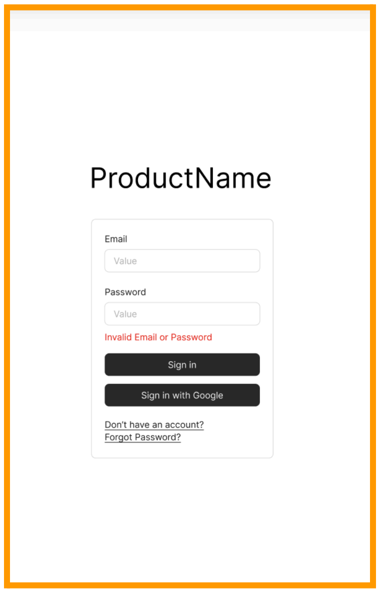
Auth0 Workflow:

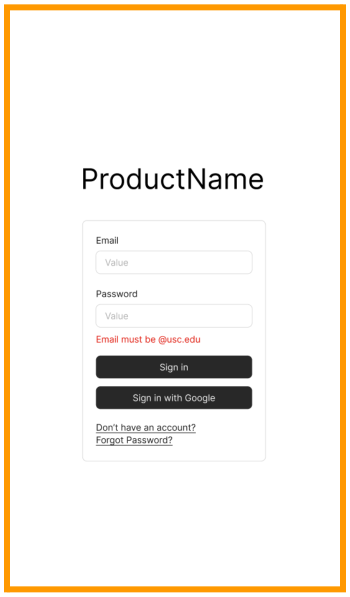
1. User visits the app and clicks "Login".
2. User either sign up with their own accounts and passwords, or the app redirects to login with Google (USC SSO) for authentication.
3. After successful login, the app redirects the user back to the app.

User vs. Guest: Only when we find that there is a logged in user present, we will show the signed-in user's name.

MOCKUP: **ERROR PAGE(S):**



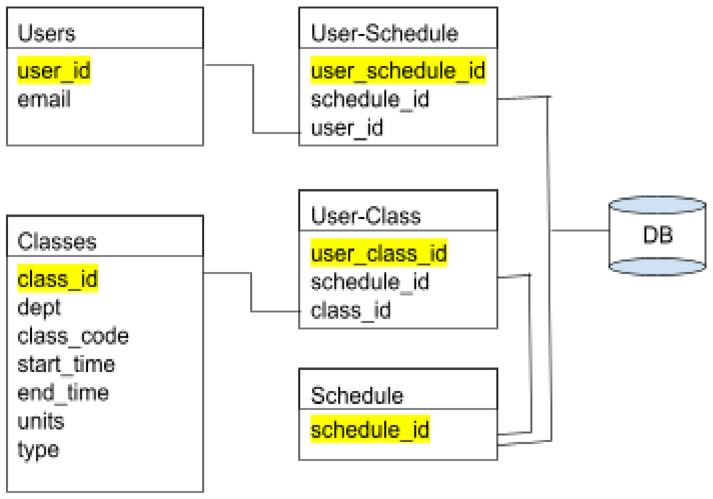




Database Architecture (MySQL)

- Tables
  - Users:
    - user_id (int primary key)
    - Username (VARCHAR(100))

        - email (VARCHAR(100))
        - Password (VARCHAR(100))
    - Classes:
        - class_id (primary key)
        - dept
        - class_code
        - start_time
        - end_time
        - units
        - Type (lecture, quiz, lab, discussion)
    - Schedule (to uniquely identify each schedule)
        - Schedule_id
    - User-Schedule (each user has many schedules)
        - user_schedule_id
        - User_id
        - Schedule_id
    - User-Class (to map users to classes in order to save and retrieve schedules)
        - user_class_id
        - class_id
        - Schedule_id



Data Collection System

The data collection system is an essential internal tool used to automate the process of updating our MySQL database with new class data every new semester. Without this tool, it would be difficult to develop a clean, working backend, where data is automatically formatted how we want it. This system is also essential to avoid making API calls every time a user queries class information, as all class information will already be stored in our MySQL database.

This program will be written using Python and will rely on the [USC Schedule of Classes API wrapper]() to collect class information. The functions used for this program will be:

- def get_all_departments(schedule: Schedule, semester_id: int)
    - Fetch all available department codes for a given semester.
    - Returns a List[String]
- *def fetch_usc_schedule(semester_id: int)
    - Fetch USC class information for all departments in a given semester.
    - Returns a List[String]
- get_semester_name(semester_id: int)
    - Convert semester ID to a human-readable name
    - Returns a String

**the public entry point function*

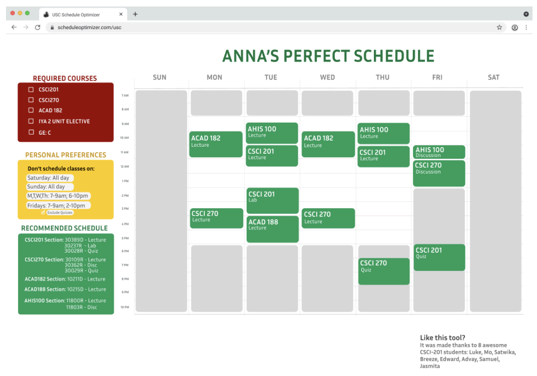
As specified in the [Database (MySQL) section](), all collection class information will be stored in an SQL database as follows:

- Class Table:
    - class_id (primary key)
    - dept
    - class_code
    - Start_time
    - end_time

Tentatively, this program will manually be run once class data for an upcoming semester has been released by USC, however, the automation of this process is a matter of future consideration.

Web Interface

MOCKUP:



It was made thanks to 8 awesome CSCI-201 students: Luke, Mo, Satwika, Breeze, Edward, Advay, Samuel, Jasmita

- The schedule should fill 4/5th of the width of the screen on a desktop view, aligned to the right of the screen. The first ⅕ of the screen (left) will display text information regarding 1) Required Course 2) Personal Preferences 3) Recommended Schedule Courses
- There will be a section for students to select the classes they must take (required courses). This should be displayed in a box colored USC cardinal red.
- There will be a section for students to add personal preferences for times (ex: don't schedule during X:XX). This should be displayed in a box colored USC gold. Once inputed, times to not schedule classes should be blocked out visually from their schedule in gray boxes.

- There will be a section for students to view the recommended schedule after optimization (based on preferences and required courses). Course sections and types should be listed in a box colored Google Green. They should also populate on the students schedule in boxes colored Google Green, with the width and position of the box equating to the duration and start/end time of the course. Green courses should always be displayed on top of any other element's layer (like a personal preference gray area).
- The schedule displayed in a modern manner, with each of the 7 days of the week as columns, each hour of the day, from 7am - 10 pm, being the rows.
- Font: Inria Sans Bold
- Note that for logged in users there will be a save schedule button after they input class details and our program displays a schedule for them
- Note that for logged in users, there will be a place to view saved schedules

Schedule Optimization System

To process the sections or classes that the user has indicated on the web page, an Asynchronous Javascript function, *optimizeSchedule*, is called. *optimizeSchedule* takes in an array of strings, containing the section IDs of the selected sections as a parameter. This function makes a POST request to the main servlet that calls the algorithm called *AlgorithmServlet*, and passes a JSON string of the array - using JSON.stringify() - as the body of the request.

AlgorithmServlet receives this JSON and decodes it into a string array of section IDs using the Gson().fromJson method from the *Gson* library. The servlet then loops through all section IDs and converts them into an ArrayList of *Section* objects.

- A *Section* is a Java class that represents a single section with details about the specific section including:
    - dates: List of integers (0 or 1) indicating which days the section is held (index 0 = Monday, 4 = Friday).
    - startTime: Start time as LocalTime.
    - endTime: End time as LocalTime.
    - id: Section ID
    - type: Type of section, including Lecture (Lecture-Lab is included as a lecture), Discussion, Quiz, or Lab
    - dClassCode: Which is the D Clearance code of the section
    - spacesAvailable: Which is the number of spaces available for the section

Retrieving such information is done through *JDBC.getCourse()* method which calls the database based on the section ID (methods outside the algorithm are called to retrieve and store information about the classes in the database).

After creating all *Section* objects, a Map of key-value pairs (Section Name (as a String), ArrayList<Section>) is created to categorize and partition the sections based on if they share the same class. For every unique class name, there is an ArrayList of sections that holds the Sections that are affiliated with the name.

An ArrayList of type *Course*, called *wantedClasses*, is created which loops through the Map and inserts each unique class and their corresponding class as a *Course* object.

- A *Course* is a Java class that represents a course with its associated sections categorized into subcourses (lectures, discussions, quizzes, labs).
    - Structure
        - name: Name of the course (ex. CSCI201)
        - title: Tile of the course (ex. Principles of Software Development)
        - sections: An ArrayList of all sections of the class that was selected by the user as an ArrayList<*Section*>
        - lectures: An ArrayList of all lecture sections of the class that was selected by the user as an ArrayList<*Section*>
        - discussion: An ArrayList of all discussion sections of the class that was selected by the user as an ArrayList<*Section*>
        - labs: An ArrayList of all lab sections of the class that was selected by the user as an ArrayList<*Section*>
        - quizzes: An ArrayList of all quiz sections of the class that was selected by the user as an ArrayList<*Section*>
        - numLectures: Number of lectures stored
        - numDiscussions: Number of discussions stored
        - numLabs: Number of labs stored
        - numQuizzes: Number of quizzes stored

wantedClasses is then passed into an *Algorithm* object as a parameter, and the algorithm object calls the method *compute(0).*

This *Algorithm* class has the following:

Methods (Java):
- **Properties:**
    - An ArrayList<*Section*> *res* which stores the final result of the backtracking method
    - An ArrayList<*Section*> *currentSections* which stores the current/intermediary classes during the backtracking class
    - An ArrayList<*SubCourse*> *subCourses* which stores the subcourses as a list
        - A *SubCourse* is a Java class that stores the specific sections of the class in an ArrayList (the lecture sections of CSCI 201, the quiz sections of CSCI 201, the lecture sections of CSCI 270, etc.)
        - This ensures that for each class, if the class has a discussion section, a discussion section must be chosen, and if a quiz section exists for that class, a quiz section must be chosen for that class, etc.
    - An Integer *minSecionsRequired* which stores the minimum number of sections required for this schedule to be valid.

- **Constructor:**
    - Initializes an wantedClasses.
    - Extracts SubCourse objects from Course into a subCourses list.
    - Computes the minimum sections required (minSectionsRequired) by looping over each *Course* object from the wantedClasses array and increments based on if the Course has a lecture, discussion, quiz, etc.
- **Compute(int currIndex)**
    - A recursive backtracking method that builds a valid schedule.
    - Base case: If all SubCourse objects are processed, save the current schedule.
    - Recursive case: Iterate through sections of the current SubCourse, add valid sections, and recurse.
- **hasOverlap(Section potentialSection, ArrayList<Section> currSections)**
    - Determines if a section overlaps with any currently selected sections.
    - Checks date and time conflicts:
        - Date overlap: Both sections occur on the same day.
        - Time overlap: One section starts before the other ends.

Implementation:

- **Constructor**
    - Purpose: Initialize the Algorithm object with the list of desired classes and populate subCourses.
    - Key Steps:
        - Iterate through wantedClasses.
        - Add non-empty SubCourse objects (lectures, discussions, quizzes, labs) to subCourses.
        - Increment minSectionsRequired for each valid SubCourse.
- **Backtracking (compute())**
    - Recursive function to construct a valid schedule.
    - Key Steps:
        - Base Case: If currIndex == subCourses.size(), save currSections to res.
        - Recursive Case:
            - Iterate through sections of the current SubCourse.
            - Check for conflicts using **hasOverlap()**.
                - Iterate through currSections.
                - Check for overlap in dates.
                - If dates overlap, check for time conflicts:
                    - A section's start time is earlier than another section's end time.
                    - A section's end time is later than another section's start time.
            - Add the section to currSections if no conflict exists.
            - Recurse to the next SubCourse.
            - Backtrack by removing the last added section.

Output:
After the *compute()* method is called, the servlet retrieves the result as an ArrayList<*Section*> object, called *res*, which contains a schedule with no overlapping sections. If *res* is a valid section, then it will only have a size of *minSecionsRequired* of the algorithm instance, otherwise, *res* is an invalid section (not all sections that were required were inserted into the section).

- If *res* is a valid schedule, meaning a valid schedule was found, the servlet returns a 200 status indicating a successful response along with the completed schedule back to the Javascript function *optimizeSchedule* as a JSON object for frontend processing and displaying on the front-end.
- If *res* is a valid schedule, meaning a valid schedule cannot be found, the servlet returns a 400 status response indicating that a valid schedule cannot be found.

Testing Plan

Authentication System

Test Case 1 - Black Box Testing - Login - Successful Authentication

Input:

- User clicks on "Login" and logs in with a valid email and password

Expected Output:

- User is redirected back to the main application screen
- The user’s past class schedules are fetched from the backend and are accessible on the main application screen

Test Case 2 - Black Box Testing - Login - Invalid Email

Input:

- User clicks on “Login” with an invalid email address

Expected Output:

- User fails to login, remains on the login page with a red error message under email address reading “Invalid email address”

Test Case 3 - Black Box Testing - Login - Invalid Password

Input:

- User clicks on “Login” with an invalid password

Expected Output:

- User fails to login, remains on the login page with a red error message under password reading “Invalid Password”

Test Case 4 - Stress Testing - High Login Frequency

Input

- Simulate 1000 users attempting to log in simultaneously using the Google SSO integration and website logisns. Users should be a mix of those signing up with their own accounts and those using the USC SSO.

Expected Output:

- The authentication system should handle all login requests smoothly without crashing. All users should be authenticated successfully within an acceptable time frame (e.g., under 5 seconds per user).

Database Architecture

Test Case 1 - Black Box Testing - Database Schema - Create Class Entry

Input

- Create an entry in the `Classes` table with valid data: `class_id = 1`, `dept = 'CS'`, `class_code = 'CS101'`, `start_time = '09:00'`, `end_time = '10:30'`, `units = 3`, `Type = 'lecture'`.

Expected Output:

- Entry is successfully created in the `Classes` table. The data is retrievable with a query, and all the fields match the input data.

Test Case 2 - Black Box Testing - Database Schema - Create User Entry

Input

- Create an entry in the `Users` table with valid data: `user_id = 1`, `email = 'student@usc.edu'`, `name = 'John Doe'`, `created_at = '2023-10-05 10:20:30'`.

Expected Output:

- Entry is successfully created in the `Users` table. The data is retrievable with a query, and all the fields correspond to the input data.

Test Case 3 - Black Box Testing - Input Validation - SQL Injection Attempt

Input

- From the Java SQL side, attempt to create a user with `email` set to `'" OR '1'='1"; -- ``.

Expected Output:

- The system prevents the SQL injection, creates no entry, and raises an error message indicating invalid input.

Test Case 4 - Black Box Testing - Input Validation - Invalid Email Format

Input

- Create a user entry with an invalid email, `user@xyz.`

Expected Output:

- User entry creation is denied, and an error message is displayed: "Invalid Email Address".

Test Case 5 - Black Box Testing - User-Schedule Relation - Add Schedule

Input

- Create an entry in the `UserSchedules` table with valid data: `schedule_id = 1`, `user_id = 1`, `class_id = 1`, `semester = 'Fall 2023'`.

Expected Output:

- Entry is successfully created in the `UserSchedules` table. The data can be retrieved, and it matches the input details.

Test Case 6 - Black Box Testing - Duplicate Schedule Entry

Input

- Attempt to create a duplicate entry in the `UserSchedules` table with the same `schedule_id = 1`, `user_id = 1`, `class_id = 1`.

Expected Output:

- System prevents the creation of a duplicate entry, returning an error indicating a primary key or unique constraint violation.

<u>Data Collection System</u>

Test Case 1 - Blackbox Testing - Data Validation

Input:

- User enters invalid class data (invalid class timings/invalid class code, etc.)

Expected Output:

- System identifies invalid data by cross-checking with database and reports back to the user on web interface
- Database is not updated with incorrect field information

Test Case 2 - Black Box Testing - Data Parsing and Validation

Input:

- Feed the retrieved JSON data into the parsing function of the Python script.

Expected Output:

- The parsed data is structured correctly according to the database schema. Invalid or incomplete data is flagged by validation checks and not processed.

Test Case 3 - Black Box Testing - Automated Database Population

Input:

- Initiate the database population process with valid parsed data.

Expected Output:

- The data is inserted into the database successfully. The database content matches the structured data from the parsing phase.

Test Case 4 - Stress Testing - Rapid API Requests

Input:

- Initiate multiple API requests in rapid succession to simulate high usage scenarios.

Expected Output:

- The script manages the rapid-fire requests without overwhelming the system or exceeding API request limits. Error handling mechanisms properly log and manage rate limit errors, if encountered.

Test Case 5 - Black Box Testing - Error Handling for API Failures

Input:

- Simulate a network failure while calling the API through the Python script (e.g. disconnect the Internet).

Expected Output:

- An error is logged with an appropriate message indicating network failure. The script handles the failure gracefully without crashing.

Test Case 6 - Black Box Testing - Missing Fields in Data

- **Input**: Feed JSON data missing critical fields (e.g., class code or start time).
- **Expected Output**: System flags the incomplete data, returns a warning or error, and skips inserting it into the database.

<u>Web Interface</u>

Test case 1 - Black Box Testing - Responsive Design

Input:

- User selects desired classes and requests to generate a schedule

Expected Output:

- Page displays a desired schedule or alerts the user to the fact that no valid schedule can be designed
- Schedule is saved to user's account if logged in

Test case 2 - Black Box Testing - Sorting Functionality

Input:

- User selects sorting features (Class Category, AND/OR, etc.)

Expected Output:

- Available classes are sorted based on boolean operators and desired class categories

Test Case 3 - Stress Testing - Large Schedule

Input:

- A user selects a large number of classes (e.g., 20+ classes) and requests to generate all possible schedules.
- The user applies multiple filters and sorting options.

Expected Output:

- The web interface handles the complex request without crashing or becoming unresponsive.
- Schedule generation completes within a reasonable time frame (e.g., under 30 seconds).
- The interface provides progress feedback and remains interactive.
- Generated schedules are accurate, and the user can view and save them without issues.

Test Case 4 - Stress Testing - Heavy Data Rendering

Input:

- User requests to view all available classes across multiple departments and semesters simultaneously.
- The page needs to display thousands of class entries with full details.

Expected Output:

- The web interface loads and renders the large dataset efficiently and quick
- Scrolling through the data remains smooth and responsive.
- No crashes, freezes, or script errors occur in the browser.
- The user can interact with the data (e.g., select classes, view details) without performance issues.

Schedule Optimization System

Test case 1 - Black Box - Optimization w/ no prerequisites, no conflicts

Input:

- Classes[] contains classes without prerequisites or time conflicts.
- ClassesSelected[] initialized to false.
- User has preferences set matching some classes.

Expected Output:

- bestScore is calculated correctly using the user's preferences
- The optimal schedule with no conflicts with the maximum score based on the user's preferences is selected

Test case 2 - Black Box - Optimization w/ some prerequisites

Input:

- Classes[] contains some classes with prerequisites
- Prerequisites map is properly initialized with classes and the prerequisites
- User has some preferences set for classes with prerequisites

Expected Output:

- bestScore is calculated correctly using only the classes that the user can take given prerequisites and the user's preferences.
- Classes requiring prerequisites are not selected unless the user has taken the prerequisite classes

- The optimal schedule with only classes that the user is able to take and the maximum score based on the user's preferences is selected

Test case 3 - Black Box - Optimization w/ time conflicts

Input:

- Classes[] contains some classes with time conflicts
- User has some preferences set for classes with time conflicts

Expected Output:

- bestScore is calculated correctly using only the classes that fit user preferences and do not cause time conflicts
- The optimal schedule with no conflicts and the maximum score based on the user's preferences is selected

Test case 4 - Stress Testing - Optimization w/ all classes

Input:

- Classes[] contains all classes
- Prerequisites map is properly initialized with classes and the prerequisites
- User has preferences set to all classes

Expected Output:

- bestScore is calculated correctly using classes that fit user preferences, do not cause time conflicts, and are able to be taken by the user
- The optimal schedule with no conflicts, only the classes that the user is able to take, and the maximum score based on the user's preferences is selected

Test case 5 - Black Box Testing - Optimization w/ no preferred classes selected

Input:

- Classes[] only contains classes that do not fit the user's preferences
- User's preferences are not set, or are set to classes that are not available

Expected Output:

- bestScore should be the minimum value of a schedule (0).
- No schedule is selected, and the user is sent “No optimized schedule could be found.”

Test case 6 - Stress Testing - Optimization w/ no classes

Input:

- Classes[] contains no classes

Expected Output:

- bestScore should be the minimum value of a schedule (0).
- No schedule is selected, and the user is sent “No optimized schedule could be found.”

Deployment Documentation

Step 1: Push Application to a Live Server

- Transfer the backend code files to the Tomcat application web server.
- Upload frontend code files to the web server.

Step 2: Deploy Code

- Start or restart the application server to deploy backend code services, and verify that all endpoints are functioning correctly.
- Ensure calls to the USC Schedule API are consistent and not interrupted.
- Deploy frontend code and ensure frontend site design meets the standards outlined in the Detailed Design Document, and the user is able to access and navigate through the site smoothly.

Step 3: Server Configuration Testing

- Ensure server compatibility.
- Ensure that the server is properly configured to support and run the backend code, frontend code, and scripts.
- Ensure that the server is accessible via the domain.

Step 4: Database Data Migration

- Export existing data from the current database into SQL files.
- Import the SQL files into the live servers MySQL database, ensuring all tables and data are correctly filled.

Step 5: Verify Data

- Check that all Users, Classes, User-Schedule, User-Class, and Schedule data in the database are present and not corrupted.
- To do this, query the record count, check for null values, and compare each original table to the new db tables (query to find records which exist in one and not the other).

Step 6: Regression Testing

- Make sure that while getting this running in a new environment that nothing was ‘broken’.
- To do so, run all tests outlined in the Testing Deliverable document to ensure proper behavior of all major functionalities.
- Monitor server performance to make sure it runs smoothly without slow downs.

Step 7: Notify User

- Notify users by email about the new deployment.
- Provide them instructions on how to access the new system, such as the url.
- Update user documentation - user guides and FAQs are important.
- Provide a comprehensive video presentation detailing functionality and usage of the application.