Sonoma State University

CS470: Graduation Planner

Ivan Lim, Greg Hultberg, and David Sornberger

Supervised by:
Dr. Ali Kooshesh

December 11, 2017

Abstract

This project titled “Sonoma State Graduation Planner” is an iOS app targeted at students attending Sonoma State University. Sonoma State Graduation Planner will aid students planning their classes for their tenure at at SSU. The application allows students to map out their classes based around their major. The application also takes into account class prerequisites and will notify the user accordingly. Sonoma State Graduation Planner will not be a replacement for a student’s advisor but act as an aid for the student in conjunction with their advisor. The development process used Swift for the front end and a Node JS API to route data between the server and the client. The local phone database utilized the Core Data framework.

Features Included

- User authentication as well as automatic token login.
- Create potential schedules with multiple terms and classes
- Utilization of the Core Data framework to save, fetch, and delete schedules from the phone’s database.
- Notify users of unmet prerequisites when attempting to save a class. It will ask the user if they still want to add the class.
- Retrieve and display Sonoma State University’s catalog of courses and their relevant information from a remote API based on user-prescribed filters.
- Academic Report page contains a collapsible table view

Features We Wanted to Include

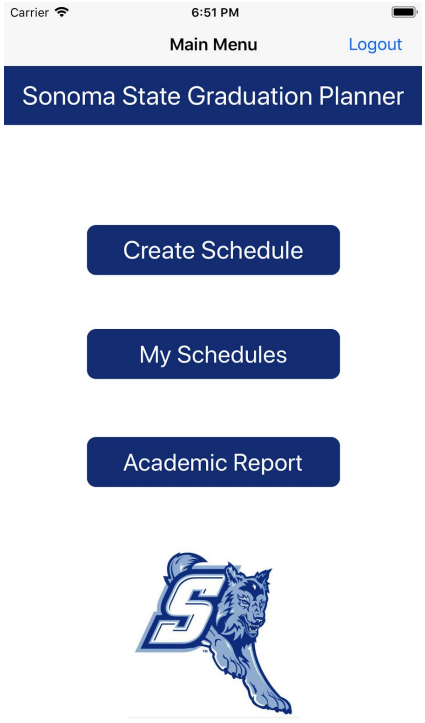
- Pull student data from the necessary API to check off requirements already fulfilled.
- Limit course offerings by spring/fall only types so that students can correctly plan on when they take certain courses. (The class data we had did not include which semester courses were usually offered)
- Display more specific general education requirements rather than just the basic GE pattern. (We did not have the necessary data for this)
- Achieving a consistent layout between all iPhones in the Apple ecosystem
- Editing a schedule after it has been saved to Core Data. Once saved, it can only be retrieved and its contents displayed.

Difficulties

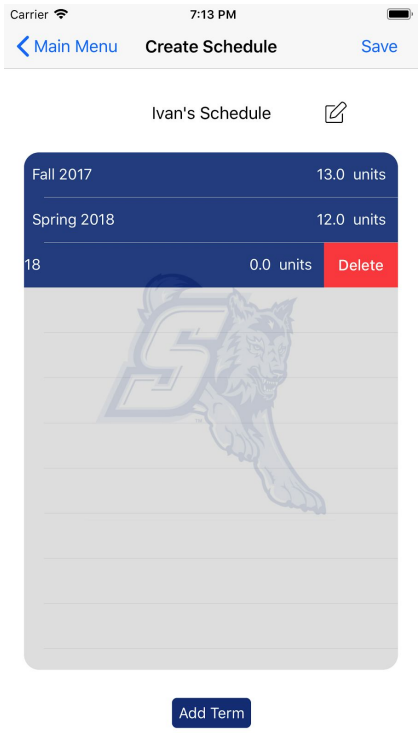
- Achieving a consistent layout between all iPhones turned out to be surprisingly difficult so we decided to work off of one iPhone and if we had time, integrate the rest. Please run this application using the iPhone 8 PLUS.

- Allowing the user to move cells around UITableViews by holding, dragging, and placing.
- Nesting multiple Core Data objects within each other. It was difficult to get started using Core Data but once the ball got rolling, it became much easier.
- Creating a consistent representation of the data as the grounds of the data we were working with changed.
- Making the data provided (JSON) consistent with what the parser (JSONDecoder) would be expecting for class objects and vice versa.

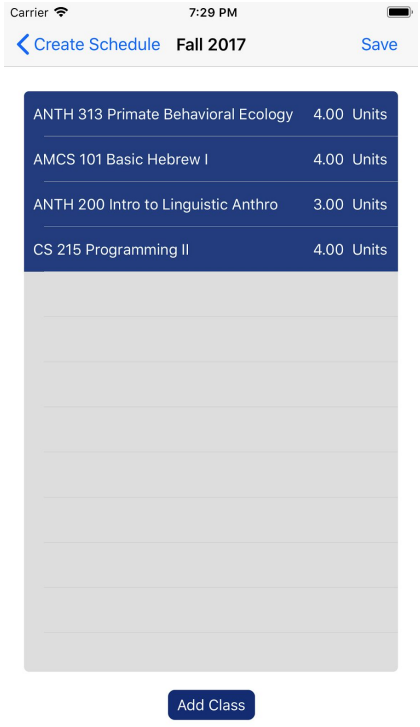
The login page is designed using UITextViews for the username and password inputs and a UIImageView for the Sonoma State logo. The background color conforms to Sonoma State University's official dark blue RGB value (20, 59, 135). When an incorrect input occurs, a UIAlertController will appear notifying the user of such event. When a valid username and password combination occurs, the authentication token is stored into Core Data and the user is taken to the main menu. When the login view controller initially loads, it will check Core Data for an existing token. If it exists, it will log the user in automatically.



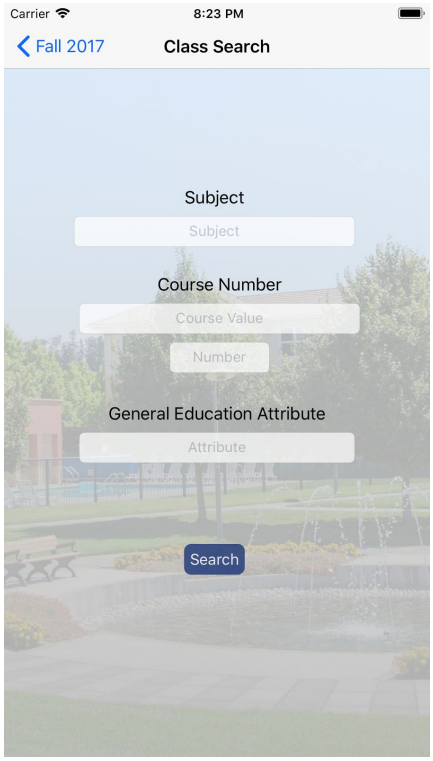
The main menu is the root of the navigation controller. A navigation controller was used in order to take advantage of its navigation stack. It allowed users to easily traverse through pages of the app without needing to traverse an equal amount backwards to get back to the root page. When the user taps the logout button, it will delete the token stored in Core Data disabling the user from automatically logging in in their next session.



The Create Schedule page contains a label denoting the schedule name which is editable by tapping the 'pen and paper' icon to its right. Tapping the icon brings up a UIAlertController with a text field allowing user input to change the schedule name. Tapping the 'Add Term' button brings up two UIAlertControllers, one after the other. The first asks the user to select whether the term is in the Fall or Spring. The second UIAlertController contains a UIPickerView allowing the user to select the year of the term. The new term is then inserted with a left slide animation into the UITableView. By dragging a cell right, the user has the option to delete it. The cell is then deleted with a fade animation.

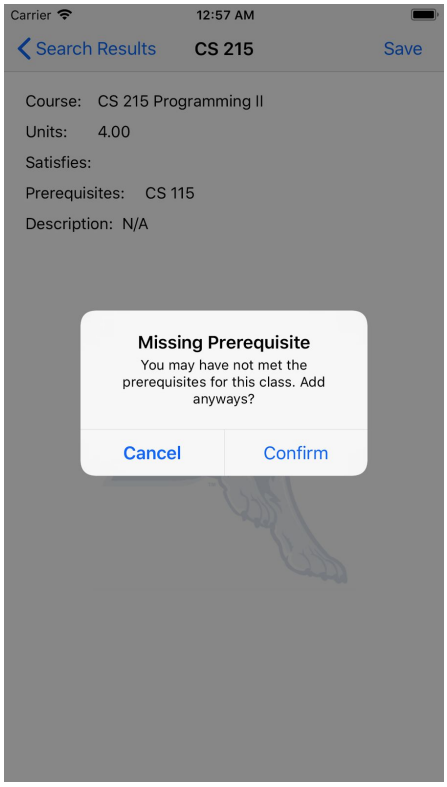


The Term Detail page has the title of the term the user selected in the previous page. It contains all the classes inside a specific term. The user can either decide to add more classes or save the term. The user can save a term and come back to it at a later time, assuming the entire schedule had not been saved. Users can swipe right on a cell to delete a class. The class is then deleted with a fade animation.

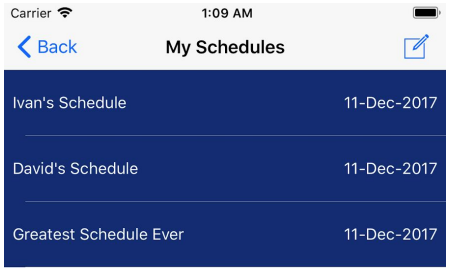


The Class Search page contains multiple UITextView objects that allow the user to filter out their class search by subject, course number, or GE attribute. Clicking on the UITextView objects will summon an overlying UIAlertController with a UIPickerView which allows users to scroll to select their desired filters. Clicking 'search' traverses through the course catalog and returns back an array of classes that match the user's given parameters.

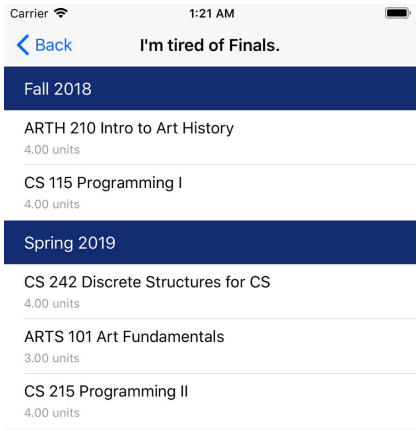
The Search Results page takes the previously generated array and uses it to populate the UITableView. Tapping on a class brings up more information about it.



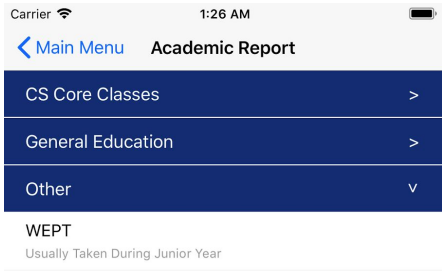
The Class Description page contain multiple UILabels each holding text about the course that was selected. A user can either decide to go back to the Search Results page or save the class to the the term. If the class has a prerequisite the student has not met either in their academic report or previously made terms, it will notify them via a UIAlertController.



The My Schedules page displays all schedules that have been created including the name and date it was created. The schedules are retrieved from Core Data upon loading the screen. Swiping right on a cell will allow the user to delete it from the UITableView and from Core Data. Clicking on a schedule will display more details about it.



The Detailed Schedule page appears when a schedule on the My Schedule page is tapped. This page splits up terms as their own separate sections of the UITableView and classes populate their respective terms. The title of the page is the name of the schedule that was tapped.



The Academic Report page displays the user's requirements they need to satisfy in order to graduate. This page features collapsible sections in a UITableView. Each section header can be tapped to be expanded or collapsed in order to make it easier and more appealing for user navigation.

Section I: Client Interface

Abstract: The client interface utilizes the storyboard and UI objects to display an interface to the user. It adheres to the Model-View-Controller architectural methodology. The information displayed on screen is mainly populated from either a remote API or retrieved from Core Data. The interface used a navigation controller to achieve a sense of organization and hierarchy to a user allowing quick and easy navigation between all the features.

Section 1.1: Ivan Lim -

Contribution Rating: 75% -

Section 1.1.1 - Summary. See views and descriptions in above pages for a more indepth summary of my Client Interface contribution. My work started with drawing basic wireframes (wireframe.cc) for the application to gain a top down glance at the hierarchy I would have to manage in Xcode. Once the wireframes were complete, I added a view controller in the storyboard for each page I would need to display and connected them using the schematics I created in the wireframes as well as connected the storyboard objects to their respective view controllers.. Once connected, I overrode the prepare function in the view controllers to pass around the proper data between view controllers such as a class or an array of terms. (Please see the above pictures of actual screenshots of the app and their descriptions).

Section 1.2: David Sornberger -

Contribution Rating: 15%

Section 1.2.1 - Summary. My work in the front end of the application was mainly to mainly into making sure that data was properly limited and fixing story board/segue issues.

Section 1.2.2 - Segues. Here I primarily focused on making sure that we had a clean navigation flow. The UINavigationController maintains a stack of viewcontrollers, which is helpful at times but other times doesn't make sense to the flow of the application, so from time to time it is necessary, on actions such as saving a schedule, to pop from the view controller stack so that the application's back button doesn't refer to something the user, most likely, does not want to go to.

Section. 1.3: Greg Hultberg -

Contribution Rating: 10%

Section 1.3.1 - Summary. My work started with inventing use cases for testing segue organization and how data is passed between views. I researched interface flow patterns and techniques to help get them implemented. Fixed compiler errors and warnings that came up during development.

Section II: Client Backend/Data

Abstract: Core Data was an essential part of our application as it allowed persistent storage of information after the user exited the app. Our application contained four Core Data entities: Schedule, Term, Class, and Token. We utilized our NetRequestHander to send our download and post requests to our API to obtain information to populate our objects and generate our authentication token.

Section 2.1: Ivan Lim -

Contribution Rating: 55%

Section 2.1.1 - Summary. My work started with creating the Core Data objects. I created three initial entities. A 'Schedule' which contained the entity 'Term' which contained the entity 'Class'. A schedule can hold multiple terms while a term could hold multiple classes. Based upon those parameters, I drew up the diagram in Xcode with the proper relationships between the entities. Initially, the objects had very little data in them but as the project continued, I added more properties I saw fit. Towards the end of the project lifecycle, I created one more entity: 'Token' which was used by David and I to automatically log a user in. The Token object would be removed from Core Data upon logging out.

Section 2.1.2 - Use in Code. I was in charge of storing, deleting, and retrieving objects not only from Core Data but between segues as well. Because of the relationship between a schedule, a term, and a class, I had to pass around an array of the terms between the view controllers while adding a class. I could not store a class into Core Data when a user selected it because the user could potentially not save their schedule. The same goes with saving a term. Using an array meant if the user decided the not save their schedule, the memory of the array would be automatically deallocated.

Section 2.2: David Sornberger -

Contribution Rating: 20%

Section 2.2.1 - Summary. A majority of my work went into connecting the data from the server to the application in an easy to access way. In conjunction with Greg I created the NetRequestHandler which was a download manager. I also worked on managing the classes which could contain the data (the models).

Section 2.2.2 - NetworkRequestHandler. This class was created with a (near) builder pattern in mind. This was so that requests could be in the form of x = NetRequestHandler(url).useParams().useToken(), which is much cleaner and an easy way to understand the request you're actually making. You could then make post_requests as well with callbacks in mind so that segues or anything else UI dependent could be performed after the data fetching occurred.

Section 2.2.3 - Decoding. The decoding in this needed to follow a pattern so that It could be recognized by the JSONDecoder in swift, so I also created model objects. These models all contain the different structures returned by the JSON in the web application.

Section 2.3: Greg Hultberg -

Contribution Rating: 25%

Section 2.3.1 - Summary. My work started with assisting David in designing the NetRequestHandler object. From there we designed parsing algorithms to return sets of classes based on the what the user is requesting and the state of the data at that time.. I tested our authentication handler using several SQL injection techniques. I tested our CoreData storage by creating mock data objects to verify our models are configured correctly.

Section 2.3.2 - Data Limitation. When prerequisites were considered for courses, there had to be some method to flag that a course was not able to be taken on the premise of missing pre requisites. The answer to this was finding if the prerequisite courses for a were a subset of the user's previously scheduled courses. This then could be used to limit the array, or access information about the course list in the class description view controller.

Section III: API and Server

Abstract: The server that this application currently exists on is blue, however, it can run on any linux server with port 8000 open. The application is a node.js application running in conjunction with a MySQL Database. The server has a single app.js file, managing the router, a DAL to handle DB requests, and a www which will handle the port management at startup.

Running Instructions: To run this application, first install the project. To do this, in your blue server specifically (as the db_connection refers to the localhost) run ./GradPlanApiInstaller.sh which will install the entire set of dependencies (it assumes node and npm are properly installed). After this, in the folder GradPlanSSUAPI run ./GradPlanApi.sh which will start the application on port 8000

Section 3.1: Ivan Lim - .

Contribution Rating: 0.5%

Section 3.1.1 - Summary: I sacrificed my blue account to be David's guinea pig test for his node installer.

Section 3.2: David Sornberger -

Contribution Rating: 99.5%

Section 3.2.1 - Summary. My work on the Node application started with setting up the routes. First I created files in the models folder to contain the various model routes that we would need for this application to function properly. I used the authentication_dal to handle students logging in to the application, the course_dal to handle courses and prerequisite requirements, and the requirements_dal to handle the degree graduation requirements for students. The database consisted of tables which were given to us by Dr. Kooshesh, these contain mock data, however they are accessed in a method which can be "hot swapped" for the actual MySQL tables at any point. The database is handled by a connection pool which manages a pool of ready to use database connections. The app file, essentially acting as the driver of the application, is a file which catches all the base routes.

Section 3.2.2 - The app file. The app file first was edited to restrict access to the application. The restrict method was used in conjunction with JSONWebToken (jwt) to securely control access to the API. Jwt is a token service which creates an encrypted token using the API's secret key, a specified encryption algorithm, and an object which can be encrypted. The object can then be decoded using the corresponding decryption algorithm. The restriction is as follows, if a user does not provide a token in accessing the API, tell the user no token has been provided and send an error. However, if the token is not properly decrypted by jwt then tell the user that the token is not valid, and send an error. Otherwise, allow the user to continue the rest of their request as normal. The app file contains reference to the various DAL files which manage the data connection services for the API. There is also a CronJob (a repeatedly occurring process on a time segment) which audits the pool to make sure there is a readily available database connection.

Section 3.2.3 - The Database files. In the database portion of the application there are two files, one is the connection_manager and the other is the db_connection. The db_connection file stores the database configuration of the application. This is not stored in the base project directory for simplicity of version control and removing the potential to add it to the publicly viewable github. The other file is the core of the database/node app interaction. It takes in the queries and sends them to an open connection in a pool of connections. This allows for a more persistent connection which will be more reliable and services a greater number of requests.

Section 3.2.4 - The DAL files. In the models folder there are files are pertaining to the data relations of the node application, these are the files that handle the sending of data (typically json) to the user.

The authentication_dal is the only non restricted route in the application, that is because users should be able to access the authentication_dal to log in without already having a token (say a first time application user). Users can either: Authenticate with their login, this process takes a post request to the server with the full name of the student and their student ID as parameters and then attempts to access an entry in the Database where there is a student with that ID and that full name. If there is the application creates a student json object using the data from the result and appends a token to that object. The token is a jwt encrypted token built off of the student result object from the database. The

database call is actually a stored procedure, which takes as input the fullname and student_id and compares it to the fname and lname of the student, concatenated with a space between and the student id. The other option for login is a token based login. With this method of logging in the user submits a token they already have and the server attempts to decrypt that token. If it can decrypt that token, it gives the user back the user object with an updated user token.

The course_dal serves as a data access procedure, the course dal has two separate sub-routes, namely prerequisites and catalog. The prerequisites route sends a list of every course with their corresponding pre requisites. The catalog on the other hand sends an entire list of courses, made unique by the individual course name (vs. the alternative, every section of every course). One of these is a plain database call to the connection manager, the other one however is an iterated json object.

Lastly the application has a requirements_dal, which renders the data as json objects for graduation requirements for majors. This route needs JSON objects of requirements, which it loads into the file and then sends as the payload for the access to /requirements/major.