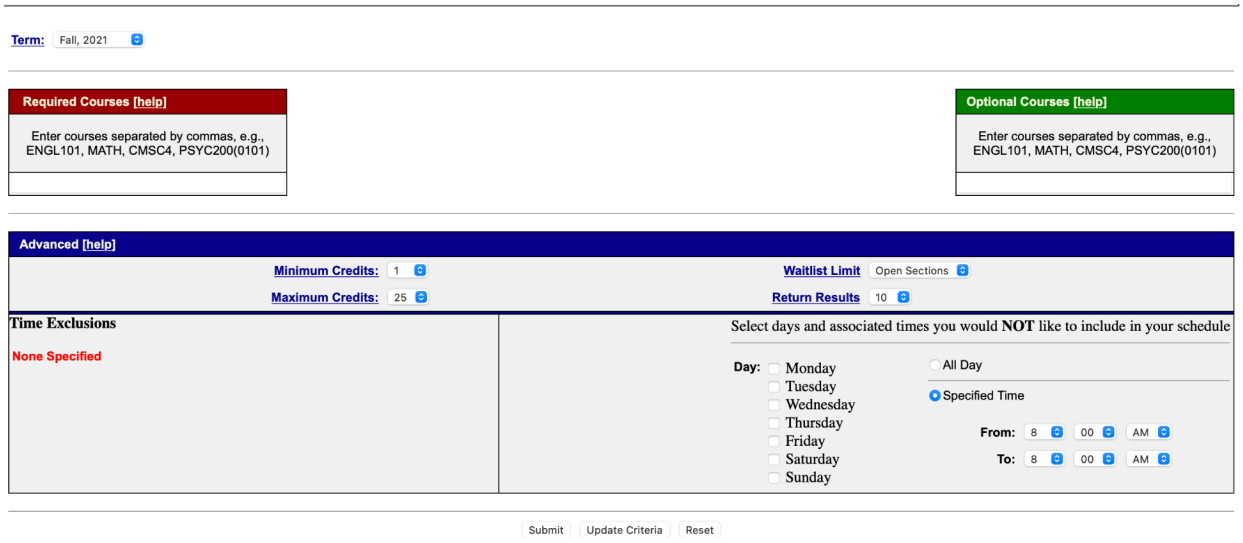
Jupiter: UMD Schedule Builder

by Surmud Jamil & Sinaan Younus

Project demo video link: Here

What is our application?

Our final project for CMSC436 is an iOS application called Jupiter. Jupiter is a schedule builder for UMD students that intelligently generates and ranks potential schedules based on professor ratings and average GPA, while also removing duplicates, and ensuring that there are no time conflicts. Our application was developed specifically for a mobile device because it allows Apple Calendar synchronization, which means that the Jupiter will automatically sync a potential schedule with the user's iPhone calendar. Once the user syncs a schedule, their calendar will have recurring meetings set up every single day for each course they selected at their proper time until the end of the semester.

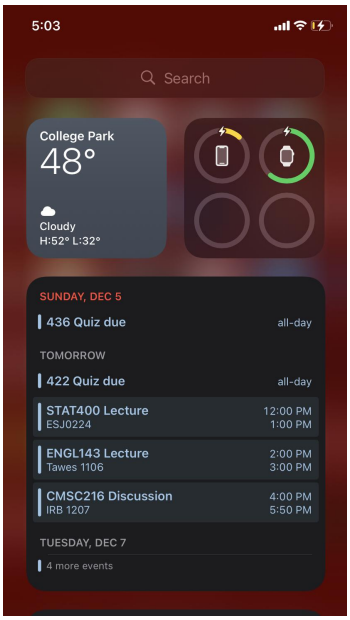


Why is it needed?

At the time of us beginning development on Jupiter, the only schedule builder that the university offered was called Venus, a web application that builds schedules for UMD students as well. However, as you can see from the image above, Venus was a pretty old application with not the best modern or attractive user interface. Although Venus does it's job of generating potential schedules for UMD students, it does not *rank* schedules based on how good they are for the student deciding on their courses, as Jupiter does.

Note: since we began development, the Venus schedule builder was updated recently with a more modern UI, but it still does not include any features that allow users to see the best schedules, as Jupiter does. Venus is also still a web application, which means that users do not have the option of synchronizing their schedule with their mobile calendar, (or any calendar for that matter) as our application does.

Why is mobile calendar synchronization important? Having one's schedule present on their mobile device is pretty convenient for an average student. As students are walking around campus with other devices stored away in a backpack, a phone is the most accessible device that people have to quickly view their schedule in order to see where they need to head after they attend a class. Apple's calendar widget, which shows a user's calendar events for the given day, also provides quick and easy access for users to view their schedule. This calendar synchronization is why Jupiter is very well suited in a mobile application.



What does our application do?
The overall layout of our app is broken into 4 different tabs, which we will describe below. Each tab serves a different purpose and the user has the option of switching between tabs, as they would in most apps.

About

Our About view displays general information about our application. The main purpose is to inform the user about what the app is and how it can be used.

Build
The build view displays a search bar where the user can type in the name of classes. This is where the user must enter in the class ID (eg. CMSC436) into the search bar. After the user enters in the name of the class, the user can click on the *"Add Course"* button which will add the course to the list of courses they are hoping to generate a schedule from (with animation). The name of the course will be displayed with an x to the right, to remove that course if desired (with animation).

After the user has entered the desired number of courses, they are able to generate a schedule by clicking the "*Build*" button. The build button doubles as a NavigationLink — it will generate all possible schedules for the current combination of classes, and also switches to a new view programmatically pushed onto the view stack.

This view (called BuildView) starts by listing the selected courses, so they remain visible to the user. It then displays the list of generated schedules in a seamless ScrollView. Each schedule displays the name of the course along with the selected time chosen for that schedule. For each schedule there is also a button that allows the user to view more information about the schedule, such as meeting times, professor ratings, and average GPA for that course. In

addition, all schedules have two more buttons - *add to calendar* and *save.* Add to calendar will allow the user to add a specific course schedule to their Apple calendar, while save will bookmark the schedule in a separate tab as described below. Once the user toggles either of these buttons, the buttons will change so that users now are able to remove the schedule from their calendar or unsave a specific schedule. The back button in this view allows us to restore the original Build page, with the input bar and course list.

Finally, the Build page also displays a reset button, which will remove all of the courses from the list, and remove all generated schedules — except for those bookmarked earlier.

View

The View tab gives details about a selected schedule. As detailed above, it includes information like class location/times, professors (and their average rating), average GPA, open seats, etc. If no schedules have been generated, or no schedule has been selected to view, it displays a message prompting the user to select a schedule (or generate some first).

Saved

This tab will display the various schedules that the user has bookmarked throughout the course of using the application. This allows the user to build multiple class schedules and have a place to look at the saved schedules that appeal most to them, as these schedules persist through pushes of the reset button.

How did we create the schedule builder?

In this section we will discuss more of the technical details regarding how we developed the schedule builder — the core of the app. Before any coding, we had to flush out its general design/architecture from a development perspective. By this, we mean the structure of classes, structs, and other pieces that would have to communicate with each other to function properly. To do so, we used principles of OOP and first broke down what a schedule really is — a list of sections of specific courses. We then broke down courses, which have a name, id, professor, (other instance members), and their own list of sections.

Our app's primary goal was, for a given list of courses: put together all possible schedules, and rank them according to their average GPA's and instructor ratings. To do this, our Schedule Builder class had to have a list of courses that the user could add to and remove from. It also needed a list of schedules to store the schedules after we generated them, in order to display them on the front end. Third, it needed a list of schedules that were bookmarked by the user, to save schedules between different resets.

The next task was to make the app functional — yes, we now had the template, but we needed a way to populate these different data structures. This brings in the two API's we used — UMD.io (which pulls from Testudo), and Planet Terp (a student-run website with its own course/professor data). Accessing data through these API's meant we needed a way to process HTTP requests. Specifically, we built 4 primary GET requests — 1) get_course, which uses UMD.io to get data about a specific course offering; 2) get_section, which uses the same API to

get data about a specific section of a specific course; 3) get_gpa, which uses Planet Terp's API to get a course's average GPA; and 4) get_prof, which uses the same API to get a professor's rating.
Swift, unlike other languages that are designed for data processing (e.g. Python), is a little more involved when it comes to requests. This complexity is largely two-fold — getting the data, and storing the data. Getting the data amounts to using shared data tasks with URLSession, simply inputting the necessary URL for the request. Storing the response required us to create several Codable structs that need their variables to match up one-to-one to the JSON format of the response. Our model.swift file covers the specific structs that we made (in order), with detailed comments, so for the sake of avoiding redundancy, we won't go through each one again here.

The complexity of requests didn't end there, however. There are two more issues to discuss — error cases, and (a)synchronicity. For error cases, we used "guard", "if let", and try/catch to make sure we caught all cases where the GET requests failed. The aforementioned 4 requests all worked together to build an individual course, so we had several decisions to make depending on which request failed. For example, if get_course failed, we could safely say that the course wasn't found. However, if only get_prof failed, that just meant that the given professor wasn't in Planet Terp's database. This left us at a conundrum — do we toss out schedules with missing professors (or average GPA's), or do we leave them in our rankings? If we rank them, how do we do so? To answer this question, and other similar ones, we thought about what we'd do in the general case of missing data. After back and forth discussion, we decided it'd be best to assign missing professors the median rating of all professors, and likewise for missing average course GPAs.

Moving on from missing data/error in the reponses, to asynchronicity: By default, the URLSession acts as an escaping closure, which means that it doesn't run until after its parent function terminates. This was a critical issue — like we mentioned earlier, we used a combination of 4 requests in our build_course method. We needed to access the data quickly, and efficiently. If requests operated asynchronously, we would have mangled data — courses without their proper members. We explored multiple solutions to this — for example, setting an instance variable for the current_course, and then using this to store all the different parts of the request responses, updating as needed. However, this didn't make sense from a design perspective, and with intertwined requests, quickly got clunky. The solution we found was in bypassing this async behavior. Using semaphores, we were able to process requests synchronously. This let us proceed as we originally hoped — we could use multiple GET requests in the build method, building our course piecemeal until it was complete, and ready to be added to the course list. An immediate downside of this is time — our app has to gather all a course's data before it adds to the list. However, for most courses this is a non-issue. It really only gets critical for courses (like ENGL101) with many sections (and as a result, many get_section requests).

The last few paragraphs covered the process of adding/building courses. But, as mentioned prior, this was but one aspect of our Schedule Builder. Once we had our list of courses set, the next task was to build all possible schedules — remember, schedules are lists of specific

sections of courses, not courses themselves. This meant that, to generate all possible schedules, we had to make all combinations of all sections. This operation can quickly become intractable given too high of a number of courses in the schedule. For this reason, and for the general belief that no student should take too many courses, we capped our course-list size at 5 courses. Moving forward, the next step was to make sure that a given schedule was viable (before adding it to our list of schedules). The main obstacle here was time conflicts — first we had to draw out exactly what a time conflict between two sections looked like, then we had to convert/parse our JSON section time data as Floats (in order to compare/judge for overlaps). Once we deemed a schedule viable, we had one more task before it could be added — scoring. Our scoring function gives equal weight to a section's average gpa and its professor's rating, and then weighs each section by the number of credits. For example, in a schedule with a class with 4 credits and another with 1 credit, we want to maximize the GPA/rating of the prior, as it has higher implications on overall GPA. After scoring, it assigns each schedule's rank using a probability density function, where higher scores are proportional to higher ranks, but do not guarantee exact order. This makes sense, as it's no exact guarantee that the highest scoring schedule will always be the best for any given student. This degree of randomness makes our app more genuine, and also more intelligent.

That, in essence, wraps up the main backend components. But Swift app development does not end with the backend, and neither will this section. We must move on to the front end, which has been briefly outlined above.Before we could do anything, we knew we had to pass in an instance of our Schedule Builder class in our controller file. After this, we had to identify which of our instance variables we needed to make @Published — the ones that needed to be observable (so the UI could update as their properties were changed). Now, we were finally ready to connect.

Building out the UI came down to using a lot of the tools we've learned in class (mainly through the projects) and combining SwiftUI best practices to build an app that matched (or exceeded) the look of our mockups. We set up the ContentView body with the TabView (About, Build, View, Saved), then built each tab. Each Tab (and subsequent Views) had an @EnvironmentObject reference to access the schedule builder passed in from the controller, as well as their own @State variables for properties that we needed to keep track of.

Since this section focuses on the schedule builder, we will only focus on the components of the UI that rely on it — but since it essentially is the app, we'll quickly cover most of the UI, without being too redundant. The Build tab communicates with our environment object, using its instance methods as button actions (ADD COURSE triggers add_course(), which triggers build_course, which triggers the GET requests; BUILD triggers build_schedules(), which triggers its helper methods). We rendered the published properties, like the course list and schedules list.

Most of the issues with the UI came down to small, but important details — spacing, alignment, etc. As experienced front-end developers (albeit JS, not Swift), we tried to carry over best-practices and other tricks we knew to solve many trivial issues. However, there were also a

few non-trivial issues. The first was the decision on where to have the VIEW DETAILS button lead to. Ideally, we wanted a navigation view that could easily be pushed/popped within the Build tab. The issue with this was that each VIEW DETAILS button was in a schedule, which was wrapped in a ScrollView. It wasn't possible to have a NavigationView within a ScrollView with the functionality we needed, so we pivoted. Instead of having our View tab display all the schedules as it did initially, we used NavigationView to display the list of schedules within the Build tab. This freed up a tab, and we could programmatically switch to View to see the details of just one schedule (instead of the list of all of them). Another bug we had was toggling the buttons "Save" and "Add to Calendar." Once a user clicks either button, it should switch to Unsave/Remove from Calendar, respectively. To manage this, we added this state to individual schedules in the backend — since our ScheduleView just uses a ForEach View to iterate through and display all the schedules. The issue was, when we toggled this text, it counted as a mutation, and because the save/calendar text was part of our struct's equals method (in order to implement Hashable), the Swift compiler sometimes errored when a user pressed either button (Save or Add to Calendar). The exact error was as follows:

> *Fatal error: Duplicate keys of type 'Schedule' were found in a Dictionary. This usually means either that the type violates Hashable's requirements, or that members of such a dictionary were mutated after insertion.*

As a solution, we added a unique id (UUID()) member to the struct, and made its equals method rely solely on this unique id. This change resulted in positive success, but has yet to completely eliminate the bug. We ask that you note this as a bug, and not a Fatal error as Swift does.

However, this gives rise for potential improvement. Along with fully eliminating this issue, we would like to further research how Swift handles HTTP requests. We want to combine the results of this research with some general algorithmic improvements of our schedule builder to improve the runtime — it's already relatively quick, but we want a lightning fast alternative to Venus. We also want to look into proportional/relative sizing in Swift, so our app can support more screen sizes (rather than just newer iPhones). With this, we hope to accomplish our mission of being the Venus alternative that UMD students need.

How did we add apple calendar synchronization?

In order to add apple calendar synchronization, we needed to use Apple's EventKit framework, which is for calendar event scheduling and removal. Since accessing a user's calendar is a privacy permission, we added the corresponding keys into the Info.plist file which allows us to request calendar permissions for our project.

Learning EventKit

The process we used for learning EventKit was a bit unorthodox. Instead of directly testing out EventKit within our app, we first created a test application in order to learn the basic underlying functionality behind the framework. Since EventKit is a much older framework, we were not able to find many modern tutorials on the internet for learning EventKit. Thus, we instead consulted Apple's Documentation at https://developer.apple.com/documentation/ which proved to be a much better resource for understanding EventKit, rather than pursuing online tutorials. In our

test application we learned the basic functionality of EventKit by trying out the basic functions, like saving an event to a calendar, and deleting an event. Once we understood that an event could be added programmatically, we knew that calendar synchronization was viable.

Challenges With EventKit

Some challenges we faced when implementing this part of our app included setting up recurring events on the user's calendar. Setting up recurring events was a challenge because of how we needed to parse the data we received from the get requests into a format that was consistent with EventKit's functions. For example, a course struct would contain the meeting times as a string containing the days of the week along with the times in AM or PM. The days of the week could be listed as "MWF", for example. Based on this, we needed to convert a string to represent a list of days along with parsing the time to be in a 24 hour format, with no AM or PM. The next problem needing to be solved with the recurring events was picking a valid start date for the recurring event. Currently, our application works with the Spring 2022 semester, so we needed to hardcode several constants representing the dates of the first week of the semester, along with the last day of classes. We foresee needing to change these constants every semester as the schedule builder would be operating on a new set of courses and dates. Despite these problems we faced, we were always able to come to a solution as our application now successfully adds recurring events to a user's calendar.

How did our ideas evolve from the start of the project?

In this section, we will discuss how our ideas have evolved since the start of our project. As a recap, our minimal and stretch goals from our project proposal are listed below. The goals that were met have been highlighted in green while the unmet goals have been highlighted in yellow.

Minimal Goals:

- Use machine learning to classify and rank classes based on their characteristics.
- Display the recommended classes based on their calculated ratings.
- Allow users to filter their selection criteria similar to Testudo. (eg. display results for a certain type of gen-ed)
- Use a modern and attractive library to display visualizations for the class data for any given class - unmet
- allow users to input their own schedule and display a visual of the user's class schedule - unmet
- generate possible schedules for a given user (new goal)
- Create a modern, attractive, and user-friendly interface that works well with IOS devices.

Our stretch goals are as follows:

- Allow for google/apple calendar synchronization (once the user enters in their schedule, allow it to sync with their calendar for the next semester)

- Automatically generate the optimal schedule for a student. This feature would create an entire schedule, and select the best classes, best professors, at the most optimal times and would allow for no time conflicts or walking conflicts.

It looks like we met our stretch goals but omitted two of our minimal goals. Why did this happen? As we developed and thought more about our application, we realized that some of our minimal goals don't quite make sense in terms of what a person would want from an application such as this. Our application is marketed as a schedule builder, so a user of our product would be focused solely on schedule-building and choosing classes, rather than other things like visualizations. We felt that our unmet goals highlighted in yellow diverged from the main purpose of the application, which is why we chose to not implement these features.

With our first unmet goal of creating visualizations for class data, we realized that a user who wants to build a schedule would not necessarily care about seeing visualizations of class data, rather they would just want to know the best average GPA for the class they know that they wanted to choose from, and have the best potential schedules built for them without needing to dig through the data themselves.

Additionally, our next unmet minimal goal of allowing the user to input their own schedule and display a visualization also deviated from the main purpose of our app. The main functionality we wanted to choose for our app was to build a schedule building application, so this goal of inputting a pre-built schedule does not make sense to us in hindsight, and looking at our final product, we don't really see how it would provide relevance to our project.

We do want to emphasize though, that the reason for not meeting these goals was intentional and solely because we felt they deviated from the purpose of our app, and not because of possible laziness. This is evident due to how we made sure to implement our relevant stretch goals and scratch the minimal goals that did not make sense.

How could our overall application be improved?

With that being said, we do have some other ideas for how we could further improve our app, and we want to talk about our ideas for improvement here. As mentioned before, one thing we would love to improve is the runtime of our schedule builder along with sizing our app to make it proportional across different devices and screen sizes. In terms of functionality, one feature that could be added are push notification alerts if a saved schedule's section is running low on seats. This would be useful for a student by alerting them to register for a course before it is filled up. In addition, we hope to add some persistence in our app by saving our saved schedules in core data. Currently, saved schedules last only for the lifetime of the app being opened/closed. By adding this feature, it will allow our save feature to become more useful. These are just a few of the ideas we have for our application, we ended up learning a lot from this project and we're definitely considering publishing the app, and adding the above features to it after this course.