

Santa Clara University Scholar Commons

Computer Engineering Senior Theses

Engineering Senior Theses

Spring 2019

SCU Courses

Conner Davis

Matthew Wong

Follow this and additional works at: https://scholarcommons.scu.edu/cseng_senior



Part of the [Computer Engineering Commons](#)

Recommended Citation

Davis, Conner and Wong, Matthew, "SCU Courses" (2019). *Computer Engineering Senior Theses*. 148.
https://scholarcommons.scu.edu/cseng_senior/148

This Thesis is brought to you for free and open access by the Engineering Senior Theses at Scholar Commons. It has been accepted for inclusion in Computer Engineering Senior Theses by an authorized administrator of Scholar Commons. For more information, please contact rscroggin@scu.edu.

SANTA CLARA UNIVERSITY

Department of Computer Engineering

I HEREBY RECOMMEND THAT THE THESIS PREPARED
UNDER MY SUPERVISION BY

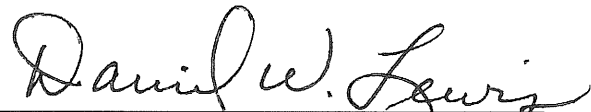
Conner Davis, Matthew Wong

ENTITLED

SCU COURSES

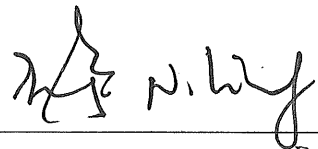
BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

**BACHELOR OF SCIENCE
IN
COMPUTER SCIENCE AND ENGINEERING**



Thesis Advisor

Date



Department Chair

Date

SCU COURSES

By

Conner Davis, Matthew Wong

SENIOR DESIGN PROJECT REPORT

Submitted to
the Department of Computer Science and Engineering

of

SANTA CLARA UNIVERSITY

in Partial Fulfillment of the Requirements
for the degree of
Bachelor of Science in Computer Science and Engineering

Santa Clara, California

Spring 2019

SCU Courses

Conner Davis, Matthew Wong

Department of Computer Engineering
Santa Clara University
June 13, 2019

ABSTRACT

Registering for classes is a nightmare that students at Santa Clara University undergo three or more times a year while juggling midterm exams. It's hard to find a schedule that works well for you, balancing the need to take classes that will satisfy degree progress with the need to work around obligations outside of class and avoid getting stuck in an 8am lecture. SCU Courses is a web app where students input their current degree progress and receive a list of possible schedules to take next quarter, collapsing the time-consuming process of carefully crafting a schedule into just one step: choose your favorite.

Table of Contents

1	Introduction	1
1.1	Problem Statement	1
1.2	Background	1
1.3	Objectives	2
2	Requirements	3
2.1	Functional	3
2.1.1	Critical	3
2.1.2	Recommended	3
2.2	Non-Functional	4
2.2.1	Critical	4
2.2.2	Recommended	4
2.3	Design Constraints	4
3	Use Cases	5
4	Activity Diagram	6
5	Conceptual Model	7
5.1	Landing page	8
5.2	Suggested schedules	9
5.3	Class search and calendar	10
6	Technologies Used	11
6.1	Node.js	11
6.2	Meteor	11
6.3	MongoDB	11
6.4	React	12
7	Architectural Diagram	13
8	Design Rationale	14
8.1	Degrees as Data	14
8.2	Omitting Ambiguous Course Requirements	15
9	Test Results	16
9.1	White box testing	16
9.1.1	Degree scraper	16
9.2	Black box testing	16
10	Risk Analysis	17

11 Societal Issues	18
11.1 Ethical	18
11.2 Social	18
11.3 Political	19
11.4 Economic	19
11.5 Health and Safety	19
11.6 Manufacturability	19
11.7 Sustainability	20
11.8 Environmental Impact	20
11.9 Usability	20
11.10 Lifelong Learning	20
11.11 Compassion	20
12 Development Timeline	21
13 User Manual	22
13.1 Installation Guide	22
13.1.1 Get the source code	22
13.1.2 Get Node.js	22
13.1.3 Get Meteor	23
13.1.4 Run SCU Courses	23
13.1.5 Note on Git branches	23
13.2 API Documentation	25
13.2.1 Web scraper	25
13.2.2 Views	25
13.2.3 Data flow	26
13.3 Maintenance Guide	27
13.4 Suggested Changes	28
13.4.1 Replace JavaScript by TypeScript	28
13.4.2 Cache data from CourseAvail API	28
13.4.3 Consider prerequisites in schedule suggestions	28
14 Conclusion	29
15 Appendices	30
15.1 Appendix I: References	30

List of Figures

3.1	Use Case Diagram	5
4.1	Activity Diagram	6
5.1	Landing page with student degree form	8
5.2	Choose schedules from list of suggestions	9
5.3	Build a custom schedule by searching for classes	10
7.1	Meteor retrieves degree requirements from the Bulletin, and makes calls to the CourseAvail API for class section data in the upcoming quarter. Ultimately, React is responsible for rendering the website the client actually sees.	13
12.1	Development Timeline	21

List of Tables

10.1 Risk Analysis table 17

Chapter 1

Introduction

1.1 Problem Statement

Registering for classes is stressful enough already, and made worse at SCU by the ineffective tools available to students. It feels impossible to find the "perfect" schedule because there are so many classes to take and students are often limited by complications like lab sections or commitments outside school. Furthermore, the online tools available to SCU students are limited in scope, only providing them the ability to search for classes individually, and not providing more relevant results based on the individual student's needs.

1.2 Background

The existing tool for assembling class schedules at Santa Clara University is CourseAvail¹. CourseAvail uses a public API connected to an SCU database containing all of the most up-to-date course information, including sections of classes that are currently available for next quarter, and information like how many seats they have. While invaluable for these reasons, CourseAvail does not perform any function beyond searching for classes and keeping track of one schedule at a time. Furthermore, it is agnostic to the student themselves, meaning the website cannot help guide students to specific classes that will help them finish their degree.

1.3 Objectives

We believe that the fundamental approach being used by CourseAvail is flawed because it is centered around the data about class sections. Instead, we want to create an application centered around the degree programs, which are ultimately what students are actually trying to accomplish. To do so, we need to find a reliable official reference for all of the degree programs at SCU, and develop an algorithm that can parse this information into meaningful data. Particularly, we need to know which classes (e.g. COEN 12 or ENGL 181) are required for each degree. There will need to be a system, too, for identifying ambiguous degree requirements, so these can be communicated to the user even though they may not be introduced into schedule suggestions themselves.

Next, we will create a web application that lets users fill out a simple form selecting the courses they have already taken in their current degree program. The form will send the results to a server that removes classes the student satisfied already from the pool of options. Our algorithm will then generate and, ideally, rank and sort by best all of the possible combinations of classes as individual schedules, and display them side-by-side so the user can quickly and easily choose their favorite. Ideally, the system will allow students to specify priorities they care about, like meeting times or distribution of workload across classes.

Chapter 2

Requirements

2.1 Functional

2.1.1 Critical

- Allow the user to input their degree progress (Classes Taken, Requirements Satisfied)
- Generate schedules based on the user's input

2.1.2 Recommended

- Allow the user to compare schedules side-by-side
- Allow users to search for classes open this quarter by all attributes
- Create custom schedules

2.2 Non-Functional

2.2.1 Critical

- Degree requirements to be obtained dynamically
- The schedule suggestions work when put into E-campus
- The program is easy to use

2.2.2 Recommended

- Sustainable over time with minimal effort from outside sources
- Mobile-friendly

2.3 Design Constraints

- Must be accessible via web browser
- No existing system describing degree requirements as data that is publicly available
- Cannot integrate existing student accounts, like Camino or E-campus

Chapter 3

Use Cases

There is only one type of user for this service: a student who wants to create a class schedule. They will have the option of providing their major and any classes they've already taken. If they provide this info, our system will be able to generate schedules automatically while also considering any other requirements specified, such as times they are busy with extracurricular activities. Regardless, users can create their own schedules by searching for classes they want to take, and adjust any schedules recommended to them by the system.

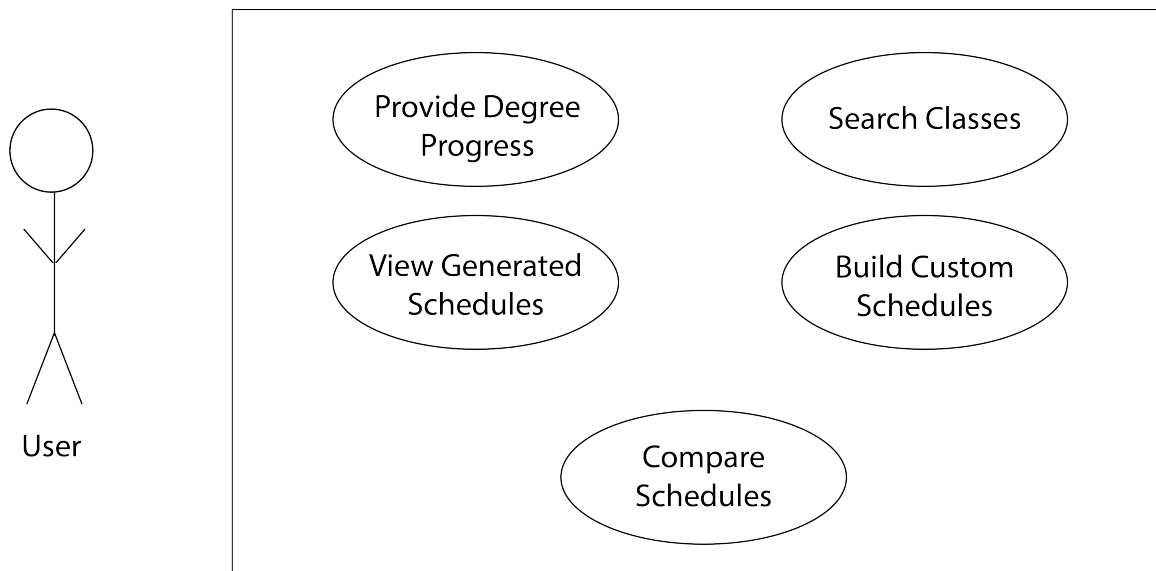


Figure 3.1: Use Case Diagram

Chapter 4

Activity Diagram

The activity diagram documents how users of our system might go about taking different actions.

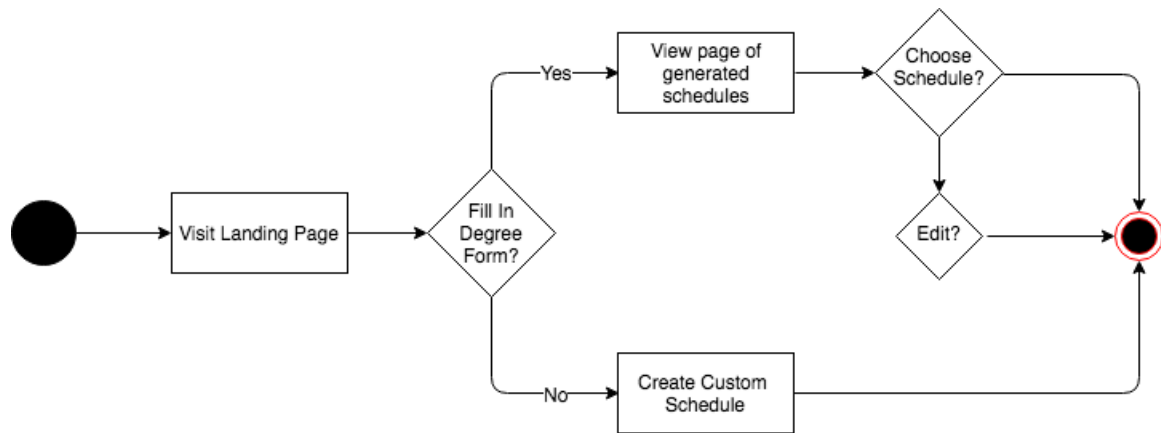


Figure 4.1: Activity Diagram

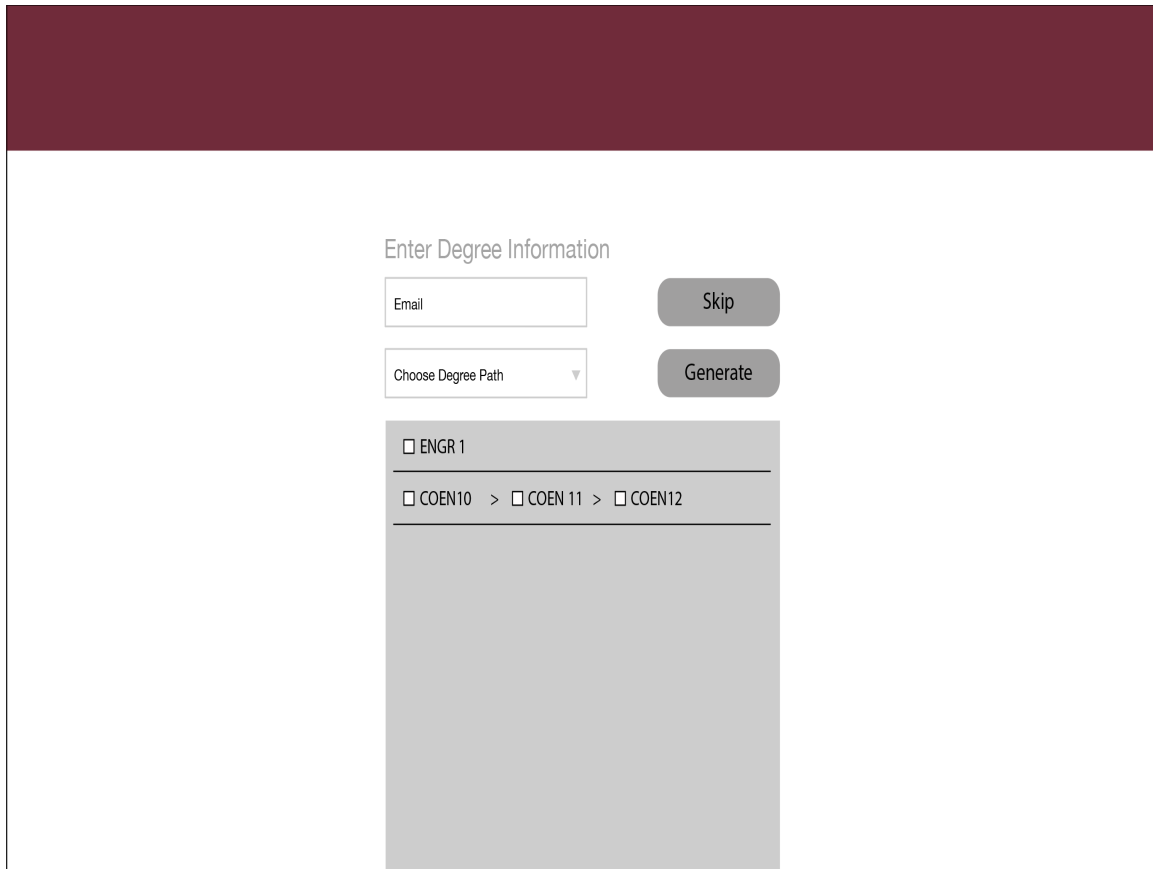
An activity diagram from the perspective of a student. Shows the two ways they can use the site, completing a degree form to generate class schedules or building their own schedule from scratch.

Chapter 5

Conceptual Model

The **landing page** (Section 5.1) greets users who will either choose to fill out the form or start creating their own schedules immediately. The form on the landing page provides the system with the data it needs to suggest schedules to the student. Once the form is complete, they will be brought to the **suggested schedules** (Section 5.2) page to compare options. If the user decides to not complete the form, or wishes to adjust a schedule that was suggested by the system, they can change it in the familiar **class search and calendar** (Section 5.3) view.

5.1 Landing page



The landing page features a dark maroon header bar at the top. Below the header, the main content area is white. Centered on the page is a form titled "Enter Degree Information". The form includes an "Email" input field and a "Skip" button. Below the email field is a "Choose Degree Path" dropdown menu and a "Generate" button. At the bottom of the form is a large gray rectangular area containing a list of degree paths: "ENGR 1", "COEN10 > COEN 11 > COEN12", and a large empty space below them.

Enter Degree Information

Email

Skip

Choose Degree Path

Generate

☐ ENGR 1

☐ COEN10 > ☐ COEN 11 > ☐ COEN12

Figure 5.1: Landing page with student degree form

5.2 Suggested schedules

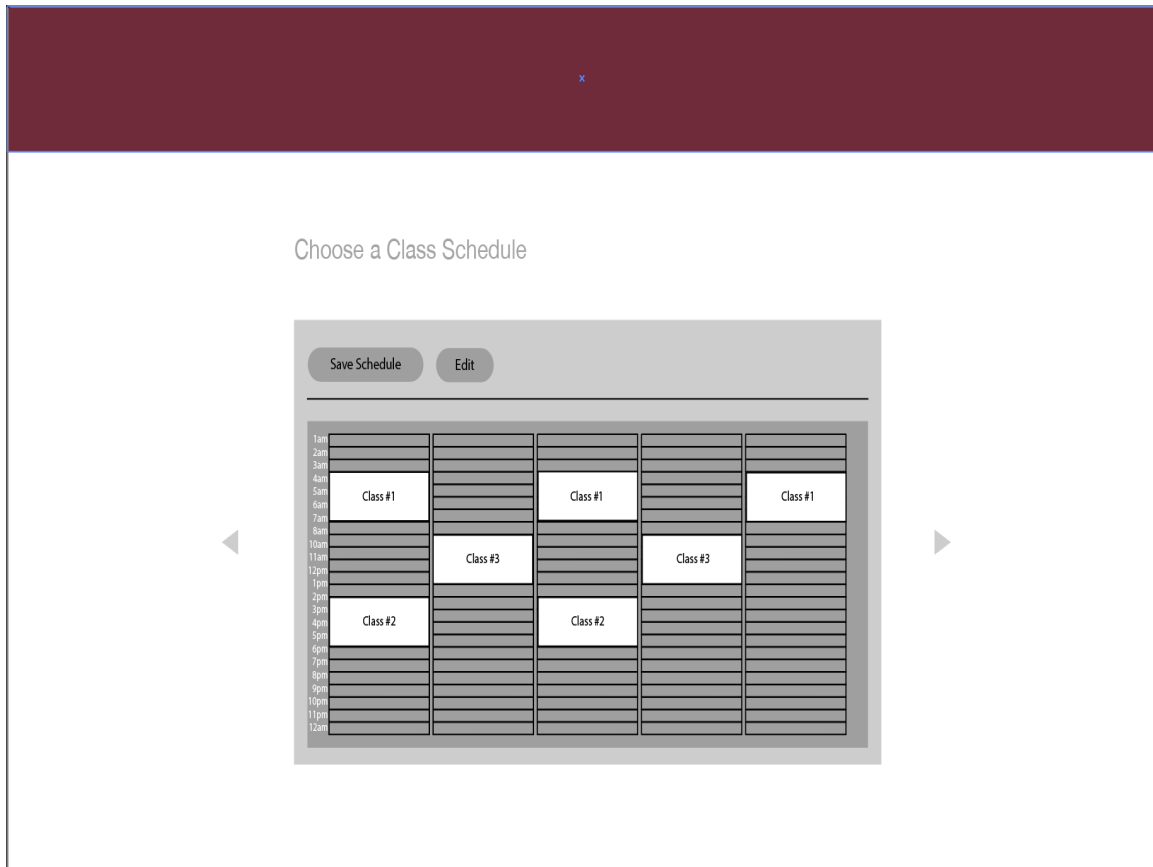


Figure 5.2: Choose schedules from list of suggestions

The diagram illustrates the process of generating a class schedule from a search query. It is divided into two main sections: a search interface on the left and a resulting schedule grid on the right.

Search Interface (Left):

- Search for Classes:** A search bar with a "Search" button and a "Generate" button.
- Course #1:** A list of options for the first course:
 - Option #1
 - Option #2
- Course #2:** A list of options for the second course:
 - Option #1
 - Option #2

Schedule Grid (Right):

The grid shows the resulting schedule for two plans, Plan A and Plan B, across 14 days (rows) and 5 classes (columns).

Plan A:

- Class #1: Days 1, 3, 5, 7, 9, 11, 13
- Class #2: Days 2, 4, 6, 8, 10, 12, 14
- Class #3: Days 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14

Plan B:

- Class #1: Days 1, 3, 5, 7, 9, 11, 13
- Class #2: Days 2, 4, 6, 8, 10, 12, 14
- Class #3: Days 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14

Figure 5.3: Build a custom schedule by searching for classes

Chapter 6

Technologies Used

6.1 Node.js

Node.js² is an ubiquitous JavaScript runtime capable of executing JS code outside of the web browser, which is where it is typically used. Of course, we're still writing an application for the browser, but Node.js lets us add a bunch of neat stuff to the pipeline before the HTML document arrives to users. Furthermore, it comes with a robust package manager called npm³ which has millions of actively maintained packages available free and open source.

Node.js is an asynchronous, event-driven engine meaning that when something happens, an object called an *event* is created, which contains a reference to the code that should run. Where we created the event, we also create a "callback function" that describes what to do after the event is finished. Then, we just have our event code call that function once it's done. Node.js has some API functions of its own, but for all intents and purposes, it's just the platform on which we write the web server.

6.2 Meteor

Meteor⁴ is a JavaScript web server framework built on Node.js. It is very similar to other web frameworks like Ruby on Rails⁵, in that it provides one central codebase to maintain a database, describe data models, and ultimately serve static HTML pages to clients.

6.3 MongoDB

MongoDB⁶ is the database management system, which provides the convenience of JSON objects to describe data. Mongo was a natural choice since it is extremely convenient to interact with in JavaScript, and our database is small and simple, not requiring any complicated query commands that may perform better or outright only work on a traditional SQL system.

6.4 React

React⁷ is a JavaScript library for building user interfaces. React works like the template engine in a traditional web server framework, introducing some subtle dynamic elements to server-side structure of HTML code. React goes a lot further however, by utilizing the classes and inheritance syntax introduced by ES6⁸ to provide an *object-oriented* system for views. It is extremely powerful and integrates smoothly with the Meteor server.

Chapter 7

Architectural Diagram

On the left-hand side, we have two remote documents that are used as inputs to generate the data used by the server. Moving to the right, the *Back-end* constitutes the web server, database, and the scraper that calls the *CourseAvail API* and parses the *Undergraduate Bulletin* webpages. The *CourseAvail API* provides up-to-date data about class sections available in the upcoming quarter. The *Bulletin* is the source we use to scrape degree requirements.

The way Meteor and React actually communicate with connecting clients is a little confusing. Meteor is responsible for network communication between server and clients, but React code is running on each client's browser and communicating with the Meteor server about how to serve data while they have the website open.

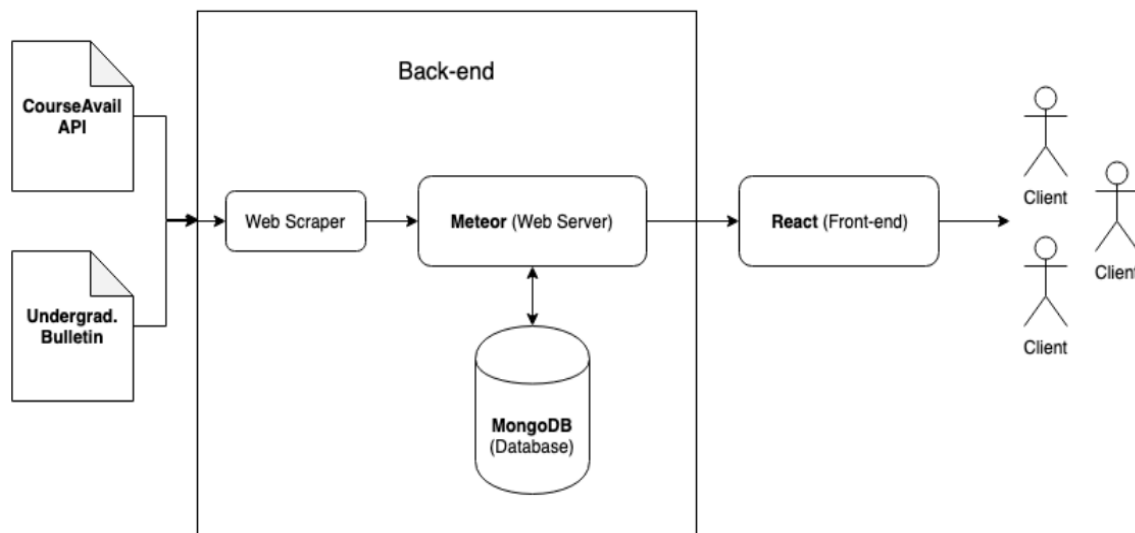


Figure 7.1: Meteor retrieves degree requirements from the Bulletin, and makes calls to the CourseAvail API for class section data in the upcoming quarter. Ultimately, React is responsible for rendering the website the client actually sees.

Chapter 8

Design Rationale

The goal of SCU Courses is to suggest relevant schedules to students based on their degree progress. First, that means we need a way to model degree programs at SCU with data. Furthermore, degree programs change over time, so we can't just hard code the requirements. Once we've overcome that obstacle, we need to provide the user with a way to select all the courses they have already taken. The final step is the algorithm that will use the courses they have not yet taken to generate a list of schedules applicable to them, ideally with some sorted order that is configurable by the user.

8.1 Degrees as Data

We originally planned to create a uniform format for the scu.edu website's pages for the various academic programs on campus, with the intent that it would then make parsing the content trivial without making it any more challenging for site administrators to maintain those pages. However, we quickly realized this was a poor approach since the time involved in contacting the website maintainers for each department was simply not worthwhile, and there was always the possibility that some department could be unwilling to work with our idea.

Instead, we decided to use the online Santa Clara University Undergraduate Bulletin¹⁰. The Bulletin is currently saved in Markdown format and then automatically converted into HTML. The HTML code for the Bulletin website is succinct and much easier to parse. Since it is the official Bulletin online source, it is always updated promptly.

8.2 Omitting Ambiguous Course Requirements

The course requirements listed on each degree program's page on the Bulletin do not follow a consistent format. Some lists delineate each course by a comma, some by semi colons, some by forward slashes or dashes. Often there is a mixture of several. Courses can also be expressed in a variety of different ways. Some are listed with just one department code, like "COEN 10, 11, 12, 20" where the code must be inferred for elements preceding the first.

To make matters worse, course requirements in general are not simple objects. Many requirements offer a wide variety of different types of classes one could take to satisfy the requirement. Some offer ranges of classes. Furthermore, more often than not, requirements (or parts of requirements) are just extremely wordy paragraphs with no decipherable "data" to interpret. Thus, these requirements, which may be crucial to the degree, are meaningless to our parser, and there's no real way to get around this.

These requirements are deemed by the parser to be "ambiguous". They are kept as their raw HTML, rather than being run through the algorithm that suggests schedules. This allows us to keep track of the requirement regardless of the fact that we do not yet know how to parse it.

Our main concern with this parser is that it will make erroneous suggestions. Therefore, we decided it best to implement only very basic parsing (i.e. on predictable and simple elements, namely lists of courses with minimal complexity). Theoretically, the project could be expanded forever to become more intelligent about the more cryptic degree requirements. It just needs to be tested quite rigorously on the relatively vast data set that is all of the degree programs across SCU.

Chapter 9

Test Results

Due to the difficulties we faced in designing the web scraper and subsequent parser, we did not get a chance to test to the fullest extent we would have liked, and did not complete all of the tests we originally set out to do.

9.1 White box testing

9.1.1 Degree scraper

To test the accuracy of the degree scraper, we added a Node.js package that creates a "deep diff" of two objects, indicating the specific keys that are different between them. When the scraper is finished, it compares the output it produced for each degree to the expected output. This theoretically would allow us to quantify the error of the results, but this is rather difficult to do in a meaningful way since all degrees have many ambiguous requirements that are straight up ignored by the parser anyway. Mostly, it serves as a way to continue unit testing the components of the scraper until it is finished with development.

9.2 Black box testing

We originally set out to conduct a visual test with some real students at SCU to see if they found the service useful, and to get an idea of specific issues they might have with the design and layout to try and identify shortcomings with the user interface. We did not have the time to conduct such a test, but doing so would be a high priority for us were we to continue development of the project and potentially seek an official release in the near future.

Chapter 10

Risk Analysis

In every project, there are situations that can arise that interfere with the projects completion. Below is a table analyzing the likely hood of some of these situations, and how severe their impact on the overall project would be.

Risk	Consequences	Probability	Severity	Impact	Mitigation Strategy 1	Mitigation Strategy 2
Time	Not finished by due date	.2	10	2	Time management strategies	Work divided among team to take full advantage of individual strengths
Work lost	Work delayed	.3	4	1.2	Use a GitHub Repository	
System is too complex, cannot be completed	Restart project, work severely delayed	.5	9	4.5	Research framework(s) chosen thoroughly before using	Test Framework early so that we can determine if it is suitable early on
Separate team member's code incompatible	Re-code sections, work delayed	.6	6	3.6	Communicate constantly even when working on different sections	Come up with team wide coding conventions

Table 10.1: Risk Analysis table

Chapter 11

Societal Issues

11.1 Ethical

Our main concern when designing this software was to make sure it is stated clearly that the algorithm is experimental and will not always generate accurate (i.e. applicable to the student) results. We do not want to create a situation where students are misled to believe they can sign up for a schedule they cannot. Of course, the ideal situation would be that there are never any errors. One way we address the issue is by our very limited scope to scanning for degree requirements. We consider a large number of all requirements to be ambiguous, and therefore do not attempt to parse them, for fear they will be interpreted as something they are not. This really limits the capacity for erroneous suggestions, but unfortunately the possibility is still present, particularly in the grammatical interpretations of lists of courses (e.g., mixed usage of commas, semi colons, dashes, and potentially other symbols to delineate entries in the list).

11.2 Social

Obviously the community of people impacted by our software is quite narrowly just the students at Santa Clara University. It's hard to know whether our software could have a lasting impact on students. One far-fetched concern might be that our software could make students lazy and create false expectations that signing up for classes should be automated entirely. In reality, of course, it is the student's responsibility (and to their benefit) to be aware of their degree requirements and best schedule plans. There is an argument to be made that perhaps students would be better off consulting their advisor who could potentially give them even more useful suggestions than our software could ever generate.

11.3 Political

Since the software only impacts SCU students and does not extend beyond the scope of slightly improving usability in the process of selecting class schedules for college, there is no reason to believe there will be ripples in the fabric of society. We cannot really see how this software could create any sort of political movement or any form of societal change, and that's just because it really isn't intended to do any of those things. Ultimately, it's just a convenient website for SCU students.

11.4 Economic

The only thing we need to pay for in order to fund this project is a server to host it. In today's age, most low-cost shared hosting web servers are powerful enough to handle a decent amount of traffic and we could always run diagnostic stress tests to assess whether more power or bandwidth would be necessary. The code is all written using entirely free and open source software, and runs on a free and open source platform. Furthermore, our database is small, does not contain any sensitive information, and is rarely written to, minimizing costs for storage which can sometimes be crucially expensive.

11.5 Health and Safety

Our project does not contain Health and Safety concerns as it is an online web page that generates non-physical schedules. The data we ask for would not be confidential or harmful information, i.e., degree progress.

11.6 Manufacturability

We feel that our project does not hold manufacturability concerns as it is not a product that will be mass produced. It also is not a standalone physical product that will cause safety concerns through handling or use.

11.7 Sustainability

The parser must be refined to be more reliable if it is to be trusted, and more robust if it is to be useful for all students. Ideally the best way to solve our sustainability concerns would be to collaborate with the developers responsible for maintaining the Bulletin to establish both a format for the degree program pages and a formal language for the contents of the degree requirements. This would make the parser much simpler and therefore far more effective.

One problem we cannot really solve is the possibility that the CourseAvail API is altered, particularly if CourseAvail itself changes in a significant way at some point in the future. Since this is our only vector to access up-to-date data on available class sections, any changes that disrupt the current implementation will be catastrophic.

11.8 Environmental Impact

Since the product exists solely as a website online there is no reason to suggest there will be an environmental impact.

11.9 Usability

Our main concern for usability is that students find it easier to find a schedule they would like to use than with the existing solution. With this goal accomplished, we hope that there's a possibility we can positively impact Santa Clara students in some small way.

11.10 Lifelong Learning

Through this project, we have been able to learn new and grow our current skills. Whether through tutorials or persistence, our project was able to show us the feeling of reward, failure, and getting back up to try again. Hopefully, this project will provide experience for future careers and situations we may find ourselves in.

11.11 Compassion

The compassion of this project is towards the students struggling with the options available to them for scheduling. Not only do we intend to help students understand what class they need to take, but also introduce them to paths they didn't even know existed before.

Chapter 12

Development Timeline

Below is our current timeline for the development of this project. First, we will continue working on our deliverables such as revising and completing our design report. Over winter break, we hope to begin the initial implementation, planning and setting up our project so we can begin working immediately during winter quarter. We aim to complete our initial systems implementation a few weeks before the design conference so we can test our project and begin to implement and test other Recommended Requirements that we may want to showcase. From there on and after the design conference, we will be continuing our final implementation and testing until the due date.

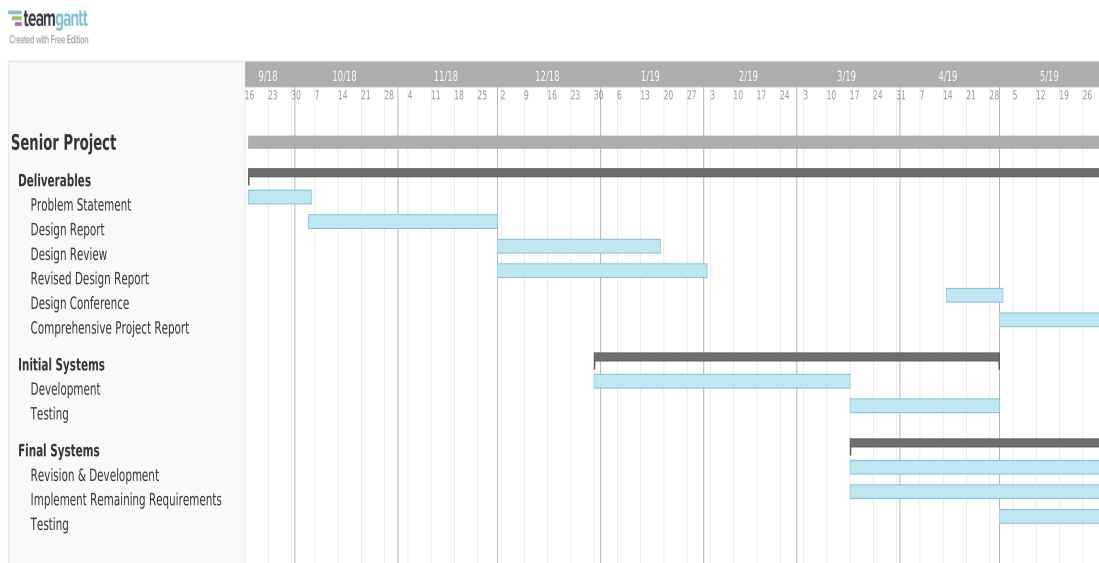


Figure 12.1: Development Timeline

Chapter 13

User Manual

13.1 Installation Guide

13.1.1 Get the source code

All code can be found publicly online at
<https://github.com/connergdavis/scu-courses>.

We strongly recommend you use Git⁸ to store a copy of the codebase on your local machine. Git is extremely easy to install on any operating system and largely useful for any developer.

To create your own copy of our repository, navigate to the folder you wish to store it in and type this sequence into the terminal or command prompt:

```
git clone https://github.com/connergdavis/scu-courses.git && cd scu-courses
```

13.1.2 Get Node.js

Now you need to get Node.js.

On Windows, you'll want to visit the download pages on their respective websites found in Appendix I. Node.js will become available in the command prompt automatically.

macOS users will find it convenient to download <https://brew.sh/Homebrew> as a package manager to then download and install Node.js. The benefit is that you can later use Homebrew to quickly update Node.js as it is changed over time. The Homebrew homepage contains a succinct guide on how to install it (it's one step). We won't need Homebrew for Meteor. To install Node.js, run:

```
brew install node
```

Linux users should have no trouble using their distribution's package manager to download Node.js. If it is somehow not provided, you can always install Homebrew on Linux or, in a worst-case scenario, compile

Node.js by source code, which can be downloaded from the website found in Appendix I.

Regardless of OS, verify Node.js by ensuring the following commands produce a version number as output:

```
node -v && npm -v
```

13.1.3 Get Meteor

Thankfully Meteor is a much simpler install process. Just visit the Meteor homepage found in Appendix I and follow the single command to install the software, which we will not repeat here should it change in the future.

Again, to verify a successful installation:

```
meteor --version
```

13.1.4 Run SCU Courses

And that's it! Now you just need to navigate back to the folder you originally cloned our GitHub repository to. And run:

```
meteor
```

... And Meteor will take care of all the rest. By default, the web server runs on the local machine (localhost) on port 3000.

13.1.5 Note on Git branches

One of the features of Git is the ability to create several different code bases that branch off of one another and exist simultaneously. The benefit of this is that it becomes a lot easier to work on one specific thing, such as adding one new feature, and isolate it from other code changes that might be going on in the background.

The source code we used to generate the demo of our project during the Senior Design Conference can be found at the branch `final-presentation-demo`. When the source code has been cloned with Git, switch to this branch is write:

```
git checkout final-presentation-demo git pull
```

To avoid accidental errors on this step, try to make sure you do not change ANY files up to this point. Once you've switched branches, you're good to go.

There are various other branches in our project. `master` is intended for the latest official release, which is considered relatively stable. At the time of writing this, we still do not consider the project to be past development stage. Thus, all of the latest code can be found in `development`. Finally, individual features that are actively being developed use the `feature/` prefix in their branch name.

13.2 API Documentation

The code base has a lot of moving parts, so it can be difficult to track what each and every little thing is actually doing. For starters, here's a rundown of the purpose of each folder in the root:

<code>.meteor/</code>	Safe to ignore - just Meteor configuration files
<code>client/</code>	One file that points to the React top-level component
<code>imports/api/</code>	Models of database elements, similar in concept to models in a traditional Model-View-Controller architecture
<code>imports/startup/client</code>	Code that should be run whenever a client connects to the server
<code>imports/startup/server</code>	Code that should be run when the server is first started
<code>imports/ui</code>	React components, which use React JSX format, and are similar in concept to views in MVC
<code>server/</code>	One file that points to scripts found in <code>imports/</code> we want to execute when the server starts

All that really matters is what's going on inside `imports/`. The files found at the root of the server are all configuration files that should not be altered unless you know what you're doing.

13.2.1 Web scraper

The web scraper is entirely found in `imports/startup/server/scrape.js`. The file has been documented heavily but is admittedly not the easiest to understand. Since there was no way to define a formal language for the degree requirements found on the Undergraduate Bulletin (as there is no consistent grammatical format), the parser is not as clean as we would like it to be. The scraper is called once on server startup and asynchronously parses degree requirements in the background.

The scraper can be augmented to visit new pages by editing the master list of pages found at `RemotePages`. To debug the results returned by the scraper, we strongly recommend you run the following function to display JSON output in readable format:

```
JSON.stringify(object_to_print, null, 2)
```

A better way to debug the scraper is to create a model of the expected output of the page, then use a tool to run a "deep diff" between the expected object and actual object. This will tell you exactly which elements do not match the expected output, making it easier to isolate what's not being output correctly, and fix your problem.

13.2.2 Views

To find React code, simply navigate to `imports/ui/`. The top-level component, `App.jsx`, is stored in the root here. Since our website is effectively a SPA, or Single Page Application, there is one "page" currently in view at any one time. Those pages are found in `imports/ui/pages/`. Conversely, `imports/ui/components` contains all React components that are elements of pages, but do not serve as pages themselves.

To better understand the way views are actually presented to the user, peek inside `imports/ui/App.jsx`. React implements its own "router" to switch between pages when a user clicks on a navigation link. In general, we highly recommend you spend a lot of time on React's official documentation if you are new to React, and consider making your own React application; there's a lot going on and the best way to really get it is through practice.

13.2.3 Data flow

When we scrape for degree requirements, those data are stored into MongoDB and hosted on the server. Furthermore, whenever we make an API call to CourseAvail to search for a class or class section, the call is made by the server and the response is packaged through to the client that asked for it.

But how, exactly? Data flow with Meteor is unique. Meteor uses a publish/subscribe system to control data flow. So on the pages (React components) where we need to call on some server data, we use `Meteor.subscribe('degrees')`; to register interest. On the other end, in the model for that data structure found in `imports/api/`, we declare `Meteor.publish('degrees')`;. In our case, we just publish every entry in the Degrees database, since we need all of them.

13.3 Maintenance Guide

Fundamentally, the entire project can safely be updated at any time with:

```
npm upgrade && npm update
```

Of course, the host machine should always have the latest versions of Node.js and Meteor installed as well, which is why a package manager is strongly encouraged for Node.js. Meteor can be updated at any time using:

```
meteor update
```

JavaScript and CSS linters have been included in the project to help ensure code formatting stays consistent and thus predictable. Warnings and errors about code style will be displayed obnoxiously on the server terminal. These rules may be altered to whatever you like, but the whole codebase is currently predicated on the rules we defined.

Since we use Webpack to assemble and compress the server and client codebases, it is likely that should an update break our implementation, it will cause disruptions that show up in Webpack first and foremost. Webpack can seem complex at first but is ultra-convenient and very much worth learning. Understanding how `webpack.config.js` works will prove invaluable should maintenance errors arise.

13.4 Suggested Changes

13.4.1 Replace JavaScript by TypeScript

We are currently in the process of re-developing the entire codebase using TypeScript⁹. One of the issues with this project is that there are lots of arbitrary data structures, namely, the degrees/degree requirements/courses/schedules of courses. While JavaScript's dynamic typing is convenient for writing scripts, it starts to lose its allure quickly when you realize you have no idea what the heck everything is supposed to actually be. TypeScript fixes that by introducing a full compile-time type checker that even allows you to define custom types.

That process is currently under development in the branch `feature/typescript`. It is almost complete but the web scraper and React form needs more work. However, the hypothesis that it would make the codebase cleaner and, particularly, the scraper easier to understand is proved quite well by this rewrite. We strongly believe this should be the top priority because of how much it will aid in development on the project in general over the long term.

13.4.2 Cache data from CourseAvail API

We really do not want to be calling the CourseAvail API for every single new query. There are tons of ways these calls can be minimized and replaced by cached results. For example, it would be safe to create a new data structure in Mongo called `Course`, and have each course contain a copy of all the attributes we need from the API response. Then we could refer to this and only make a new query for the latest data on that course if we need to know the number of seats available, which is the only field that will ever change.

13.4.3 Consider prerequisites in schedule suggestions

Currently, schedules can sometimes suggest classes where one is a prerequisite of the other. These are invalid and should be tossed out. This will be more challenging than perhaps it seems on the surface. The only way to know whether a course has prerequisites is to make an API call to CourseAvail for the class, and read its description field. At the very end, the same text is always written, "Prerequisites: X, X, .." This can easily be parsed to determine the relevant prerequisite courses.

However, careful consideration should be made here. We should really make sure we are caching results of this operation because prerequisites seldom if at all change. We're talking potentially years of unchanging values. These absolutely should not be calculated more than once, period, and can easily be stored on a semi-permanent basis inside Mongo.

Chapter 14

Conclusion

One thing we took away from our project was that it would have been nice to do a lot more preparation during the earlier months in the process. Had we spent more time learning how to develop on the Node.js platform with all the technologies we intended to use, and became more familiar with the development environment, it would have aided us substantially in the actual development of the application.

We also wish we could have developed several more features and, in retrospect, would have started development of the parser much earlier had we realized we would run out of time so quickly.

Our biggest priority is to make the parser more robust such that it can provide mostly accurate results for every single degree program. We do not like the fact that some students are currently left out with the parser as-is. Furthermore, University Core Curriculum is essentially completely ignored because of the nature of its requirements. We would really like to find a way to implement support for this crucial part of every student's degree.

Ultimately, our experience was a set of life lessons that will prove invaluable in our careers as software engineers. The reality is that no project ever goes "as planned". The larger the scope, the more challenging it becomes to accurately (or even remotely come close to) predict what will take the most time, or exactly how to do each and every step.

We are most proud of the fact that we were able to accept that certain pieces wouldn't make it through, and prioritized the parser, which mattered the most, getting it done on time and proving to ourselves that the project has potential to live on. The rest of the features will fall in line easily with the foundation set strong.

Chapter 15

Appendices

15.1 Appendix I: References

[1] *CourseAvail*. Santa Clara University, <https://www.scu.edu/apps/courseavail/?p=schedule>. Accessed 25 January 2019.

[2] *Node.js - a JavaScript runtime built on Chrome's V8 JavaScript engine*. Node.js Foundation, <https://nodejs.org/en/download/>. Accessed 25 January 2019.

[3] *npm — build amazing things*. npm, <https://www.npmjs.com/>. Accessed 25 January 2019.

[4] *Meteor — Build Apps with JavaScript*. Meteor Development Group, Inc., <https://www.meteor.com/install>. Accessed 25 January 2019.

[5] *Ruby on Rails — A web-application framework that includes everything*. Ruby on Rails, <https://rubyonrails.org/>. Accessed 25 January 2019.

[6] *MongoDB — Open Source Document Database*. MongoDB, Inc., <https://www.mongodb.com/>. Accessed 25 January 2019.

[7] *React - A JavaScript library for building user interfaces*. Facebook Inc., <https://reactjs.org/>. Accessed 25 January 2019.

[8] *Git*. Software Freedom Conservancy, <https://git-scm.com/>. Accessed 6 June 2019.

[9] *TypeScript - JavaScript that scales*. Microsoft, <https://www.typescriptlang.org/>. Accessed 6 June 2019.

[10] *Santa Clara University Undergraduate Bulletin - 2018-2019*. Santa Clara University, <https://www.scu.edu/bulletin/undergraduate/>. Accessed 6 June 2019.