Detailed Design Document

Rowan University

Dynamic Course Scheduling Application

Prepared By: David Carlin, Brian Mattei, Clifford Black, Cole Robertson, Nicholas Weintraut, Sean Dale

Class: Senior Project Fall 2016

Table of Contents

Introduction3

Module Design 5

Algorithm Design8

Website Design13

**1. Introduction**

This is the official design document, as per the requirements specified in the SRS, for the Dynamic Course Scheduling Application. The system will be developed using a combination of the MEAN stack (MongoDB, Express, Angular.js, and Node.js) along with Ionic Framework for iOS and standard Java programming. Our software will utilize five different modules. These modules are as follows:

* 1. Our Database module. This module will be hosted via mLab, and was designed with MongoDB. The courses stored within the database will have all of the relevant information contained within (course numbers, subject numbers, section numbers, meeting times, and meeting days). These courses will be stored as .json documents, and will be accessed via our server.
  2. The Parser module. This module will consist of Java code which will parse a given .csv file for the correct headers in which our Database and Server will function off of. The Parser module will be called and initiated via the command line and parse a given .csv file. The Parser then will upload the parsed .csv file to the Database.
  3. The Server module. This module will consist of Node.js code which is intertwined with our website code. The Server will initialize a link to the hosted Database module, and also perform any algorithms involved in finding appropriate course pairings for user-given input. The Server module also queries the Database module to find the appropriate course matches.
  4. The Website module. This module will consist of Angular.js and HTML code to design a functional website for the user. This website will allow users to input their specified times, days, and course. Once this is done, the user will submit their data, and the Angular.js code will take over in calling the Server module with the given user input. The Server module will return the appropriate data to the Website module, and the data will be displayed for the user on the website.
  5. The iOS App module. This module will function almost identically to the Website module, except with the added Ionic Framework in order to design an iOS application using the same methods that were used to design the Website module. This will allow seamless integration of the same code used to develop the Website module for the iOS App module, and also allow for the functionality to be identical to the Website module.

**2. Module Design**

**2.1 Database**

**Purpose:** The purpose of this module is to store all of our necessary data in such a way so that it is easily accessible by our Server module. This will allow seamless transition of data between the user and the backend of the application.

**Rationale:** MongoDB was used for our database design for seamless integration with other parts of the MEAN stack, and for the ability to store different types of data. We need a database so we have an easy way to access data that isn’t local, allowing for users to use the program even when there isn’t a local machine running it.

**Required Interface:** .csv file(s) parsed by the Parser module.

**2.2 Parser**

**Purpose:** The purpose of this module is to parse a given .csv file for the appropriate headers needed by the Database and Server modules. The given .csv file should be a list of all courses at a given college, and course meeting days, meeting times, course numbers, and course IDs should be held within the .csv file.

**Rationale**: Very quickly we realized how much useless data was contained within a .csv file generated by Section Tally. Much of this data did not need to be used within our algorithms and calculations when determining what course schedules students wanted. So, we decided we needed a module that would parse through the .csv files for only relevant data.

**Required Interface:** raw .csv file from a given university.

**2.3 Server**

**Purpose:** The purpose of this module is to handle all of the backend logistics and algorithms required to make the application function. This includes querying the database, calculating the schedules given the user’s input, and sending the queried data to the Website and iOS App modules.

**Rationale:** We need a server module to actually perform the calculations needed to make the app function. Node.js is able to query the Database module so that the given input from the user is used to find matching .json documents. We also needed a way to compile this data, and the Server module compiles the retrieved data in such a way that the Website and iOS App modules can utilize it and display it.

**Required Interface:** User input from the Website and iOS App modules

**2.4 Website**

**Purpose:** The purpose of this module is to handle all of the data, and display it in such a way that the user understands it via a nice looking UI. It also is the entirety of the front-end for the user, as it is the only thing the user will use and see (along with the iOS App module, so if they choose). The Website module also sends the user input to the Server module for querying so that the Website module receives the appropriate data back.

**Rationale:** The application needs a way to display the data, and for the user to actually *use* our application. The Website module (as well as the iOS App module) does this. It gives the user a UI to input data into, and a UI to display returned data from the Server module.

**Required Interface:** User input

**2.5 iOS App**

**Purpose:** The purpose of this module is the also handle all of the data, and display it via a UI. This module does the same thing as the Website module, albeit by similar (but different) means. This module will use all of the same techniques and functionality as the Website module, but with the addition of utilizing the Ionic Framework as a way of making an iOS application out of the existing Website module’s code.

**Rationale:** In order to give a cohesive and convenient user experience, we wanted to develop an iOS companion application in addition to the website. This will allow users to create their schedules on the fly, rather than having to have a computer with them at all times. Nearly everyone has a smartphone, so having a phone app gives users another benefit altogether. This module will be structured very similarly to the Website module, utilizing almost the same structure completely.

**Required Interface:** User input

**3. Algorithm Design**

For the algorithms needed to have our application run properly, one thing was certain: we needed to devise a way to accurately query our database based upon user input. To do this, the app works as follows:

**3.1** The first thing we need to do after connecting to the database is query the courses that we want on the days that we want them. In order to query the courses that we want we need to create a query with the following JSON format:

{‘$and’:  
 [

{‘$or’:  
 [  
 {‘$and’: [{‘Subj’: ####}, {‘Crse’: ####}]}, {‘$and’: [{‘Subj’: ####}, {‘Crse’: ####}]}, ...  
 ]  
},

{‘$or’: [{‘Meetings.Day’: {‘$nin’: ####}}, {‘Session’:‘Online’}]}

]

}

This will return the all of the sections in an array that are the courses being requested and fall on the correct days entered. If any course was entered incorrectly or did not fall on the days entered, the user will be notified.

**3.2** The next thing the algorithm does is it goes through each section of the array and goes through each meeting of that section and checks the time of the meeting against the time entered by the user for that specific day. If these times conflict that section is removed from the overall section array.

**3.3** The next thing the algorithm needs to do is build separate arrays for each course. Each array should contain only the sections for that specific course. To do this the algorithm will go through each section in the section array and compare the course number and the course subject and if they match it puts the section into its corresponding course array.

In order to explain how we created course pairs some groundwork needs to be laid first. Course array is an array that contains an array of sections for each course. It looks something like this:

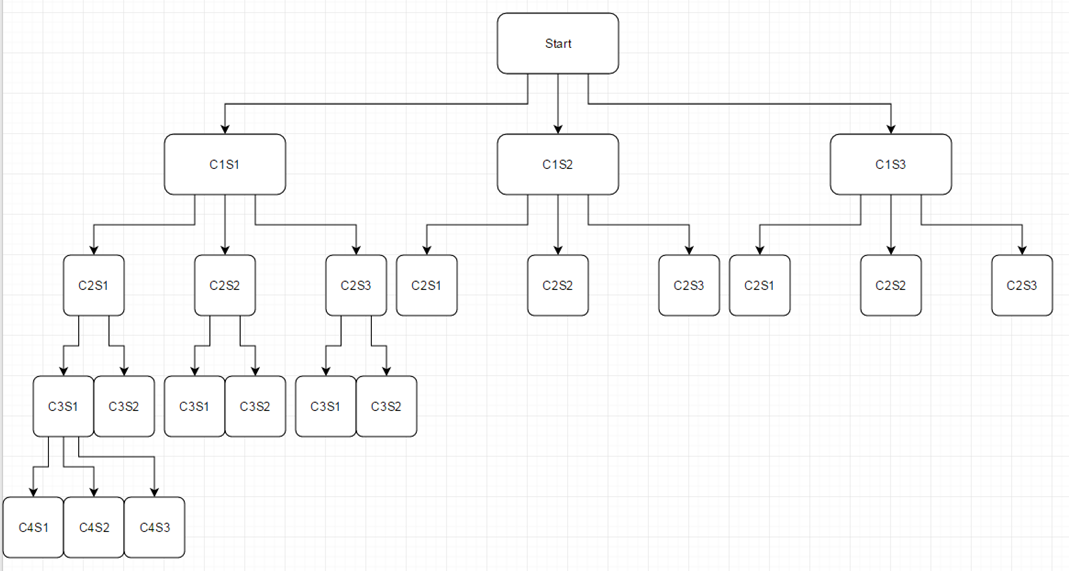
[[sections for course 1], [sections for course 2], [sections for course 3], …]

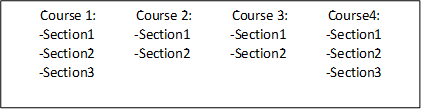
This structure brings with it a way to organize and call upon each section at will. We use a numbering system and create pairs based on the indexes of the courses and sections. For example, course 1 section 1 would be pair (1,1) and course 3 section 4 would be pair (3,4).

**3.4** In order to tackle the scheduling algorithm of this app we decided we would compare every section of a specific course to every other section of other courses. This creates pairs or pairs. For example when comparing course 1 section 1 to course 3 section 4 this generates the pair of pairs ((1,1),(3,4)).

We want to eliminate all good pairs of sections, I.E. sections that do NOT conflict based on time. In order to accomplish this we want to create ALL possible pairs first of sections and then remove all the good pairs based on no confliction. It is worth noting that online classes do NOT conflict with physical classes so they will be removed from the section pair list.

**3.5** We now have multiple pairs of sections that represent conflicting sections. In order to represent schedules we decided to use a tree structure where branching from the top level down to a leaf represents a schedule. We start off by building a tree which contains all possible connections, an example of which is shown below. The example has omitted repeats for the sake of clarity and neatness. In the tree each point C#S# represents Course # Section #.





**3.6** Diving into a node we can see how nodes are composed. Each node consists of its C#S# pair, a Boolean representing if this node conflicts with its parent, and finally a list of its children’s nodes. When we get down to a leaf, we exclude the list of children nodes.

We want to use the bad sections to remove them from the tree, leaving a tree the represents only good schedules. For each pair in section pairs we recursively branch through the tree. Each section pair represents an edge between nodes in a tree. Using the bad section pairs, we go through the tree recursively and once we find the represented edge of which there may be multiple within the tree structure, we change the child node of the pair’s Boolean variable to false. By changing this to false it represents that all possible pairs that are below the child in the structure.

**3.7** The way we determine what edge should be false is as follows: Each level in the tree represents one course, where each node is a section of that course. There are three different types of cases. The first one occurs when we are currently on the level of the tree that corresponds to the first section of the bad pair, and the second section corresponds to the level directly below the first, we set that the node corresponding to the second section to “false.”   
The second case also is on the level of the tree that corresponds to the first section of the bad pair, however, the second section of the bar pair is not a node directly below the level we are on currently. In this case, we have to recursively traverse through each child of the current node until we arrive at the level corresponding to the second section of the bad pair, and then set that second section to “false” once we arrive at it.   
The final case is if the level that we are currently on is higher than the level that we need to be on. In this case, we need to traverse through each node of our current level and recursively search for the first two possible inputs. This allows us to be on the level that we need to be on, and then the algorithm functions as if the first or second cases were being performed.

**3.8** Finally, the algorithm generates the schedules. To generate the schedules we recursively branch through the tree, from left-to-right, at each node we add that corresponding section to an array of sections that now represents the schedule. Once we arrive at a leaf, we add the entire schedule to an array of schedules. If we arrive at a false node, we skip over it, do not add it, and traverse back up the tree.

**4. Website Design**

**4.1** For our website, we have decided to incorporate textboxes for users to input their times. The format of the textboxes are such that the user will understand that they’re for times, as they’re set to display “--:--“ by default, indicating that times are requested. It also gives the option for users to input “AM” or “PM” to specify.

**4.2** In addition to this, there will be check boxes for the days Monday-Saturday in order for the user to check off which days he/she would like to be on campus for.   
**4.3** The website will also include text display fields for each day checked, showing the blocks of time the user has specified after a button labeled “submit” has been pressed. This field shows the user the blocks of time he/she wants, and also gives the option to remove a specific block of time via a “remove” button.

**4.4** Similarly, there will be two text fields for the user to enter their course subject (i.e. ‘CS,’ or ‘CHEM’) and their course number. Once these have been typed in, the “add” button can be pressed to add the course to be searched for. The courses will be added to a field similar to the day’s/times field, complete with a “remove” button for each as well.

**4.5** Finally, at the very bottom, there will be a “submit” button which will initialize the algorithm to search. Once the algorithm is done, the website will display the courses at the bottom of the page in little boxes displaying their Subject, Start Time, and End Time. These boxes are displayed for each day they are held (M-S), and displayed in time-order. They are displayed in each appropriate day column, and below all of the columns there is a “prev” and “next” button. These buttons allow the user to change through the potential schedule combinations.