

Degree Auditing System

By

Alexis Santiago

Hugo Santamaria

Mario Velasquez

Victor Martinez

Final Report for CSCI 4390

The Computer Science Department

University of Texas Rio Grande Valley

Spring 2020

We hereby certify that this senior project report satisfies the final report requirements of CSCI 4390.

Name

Date

Faculty Advisor

Abstract

This project aims to provide computer science students a simpler alternative to degree works, with the added functionality of storing their advisement sessions on their account. As for administrators, they will have an online record of advisement results, being able to see the outlook of which courses students plan to take, and plan their staffing accordingly.

Contents

Abstract	1
Introduction	3
Goals	4
Implementation	4
GUI	4
Flowchart/Functions/Layout	4
Database Model	6
Testing	4
Results	4
Bugs	4
Overall Outcome	4
Conclusion	4
References	4

Introduction

The motivation behind this project is to provide a solution to the advising problem for computer science students. As the number of students grows so does the need to organize the information related to each student. Our project aims to provide a solution to this problem by providing students and advisors with a system that collects and provides the advisor with this information. The Goal of this project was to first provide a system to manage advising sessions and release the students' hold.

The tracking and updating of a students degree progression is another feature that we wanted to implement. Basically, an alternative to Degree Works. We want students to be able to update which courses they've taken, and subsequently, seeing the progression in their degree plan. This will require Administrators to create and manage courses(and their categories), degree plans, pre-requisites, and so on. Our database is designed with all of this in mind.

The project was broken down into two main components, the api and the frontend. This was done so that the project was not limited to one website or one mobile app. The implementation with an api allowed for any developer working on this project to use the framework they are most comfortable with and allowed for variety for the end user. This way users can install the app if they want more features than just a browser but are not limited to a mobile app if they don't want to install one.

Goals

The end goal of the project is to

- Provide fluid academic advising
- Keep track of degree updates and changes
- More efficient class planning
- Degree audits
- Address on time graduating deficiencies
- Be fully aware of specific major requirements in the general core

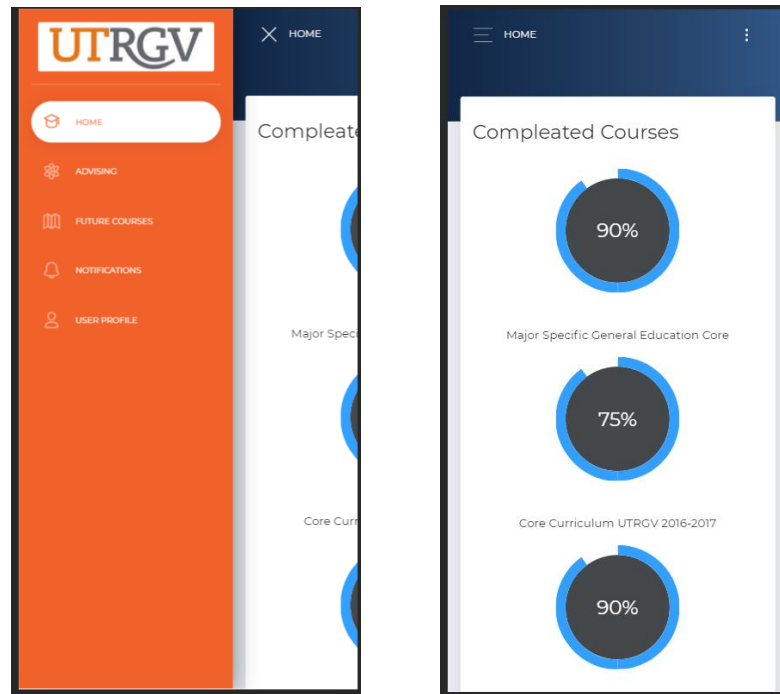
Implementation

GUI

- Desktop (nodejs)

The desktop gui was built using node js running on an express server. This allowed the use of tools like handlebars js for templating, cookie-parser for managing cookies, and axios for Jason web token authentication. For the Home page and the rest of the pages I used a bootstrap theme by Creative Tim called Now-ui. This theme allowed for quick development by providing us with bootstrap elements that we can then customize to our needs. The main reason for picking this theme was because of its adaptability of different screen sizes.

The main purpose of this application is to be used during a group advising session. This forced us to take into consideration what devices users will have available. We came to the realization that some students may not have a laptop with them and therefore it is a requirement that the website will also be mobile responsive.



Handlebars allowed for the use of layouts. Layouts are used to organize the directories and remove redundant code. Instead of having the same html template duplicated for all pages I created a main.html file with the sidebar and header. All other html files are injected into that file unless they are explicitly told not to such as the login and register pages.


When the user goes to our website, they will be directed to the login page. If they do not have an account, they must click on the register page to create an account. By default, all users are students because admins must be added manually. In the register page a user is asked for an email(username), password, first name, and last name. when this data is submitted the form calls a post function in the server.js file. This function takes the parameters through the use of body parser and using axios sends that data to the api. The api then returns that Jason web token which is stored in a cookie. This token will be used to authenticate the user and must be passed to each api call via the header.

```

//get login information from api
//`http://localhost:4567/api/login?username=${username}&password=${password}`
app.post('/login', (req, res) => {
  let username = req.body.username.trim();
  let password = req.body.password.trim();
  axios.get(`http://localhost:4567/api/login?username=${username}&password=${password}`)
    .then(function (response) {
      console.log(response);
      //if succsesful
      const token = response['data'].token;
      console.log("login token: "+token);
      //store token in cookie
      res.cookie("token", token);
      res.redirect("/home");
    })
    .catch(function (error) {
      console.log(error);
      //if unsuccessful
      res.redirect("/login");
    });
});

//Log User outs
app.get('/logout', (req, res) => {
  //delet user token
  res.cookie("token", {expires: Date.now()});
  res.redirect("/login");
});


```



UTRGV Degree Audit

LOGIN

[Forgot Password](#)
[Create An Account](#)



UTRGV Degree Audit

REGISTER

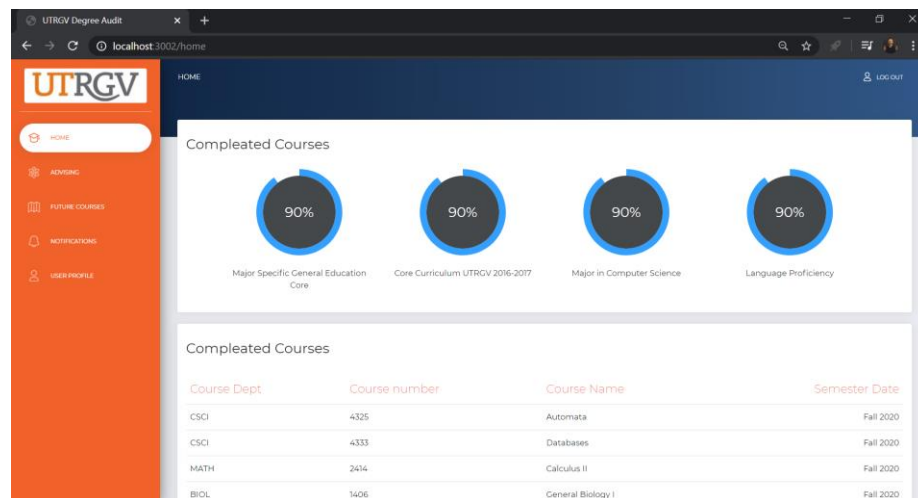
[Forgot Password](#)
[login](#)

```

//get home page
app.get('/home', (req, res) => {
  //get all the courses using token
  axios.get(`http://localhost:4567/all/Courses`, {
    headers:{
      'Authorization': `bearer ${req.cookies["token"]}`
    }
  }).then(function (response) {
    // handle success
    // console.log("courses: "+JSON.stringify(response["data"]));
    let courses = response["data"];
    console.log("courses variable: "+courses);
    //render page with courses
    res.render("home", {active:
      { home: true },
      page: "Home",
      courses: courses
    });
  })
  .catch(function (error) {
    // return error to notify user
    console.log(error);
  })
});

```

When the token is received the user is redirected to the login page where their credentials can be used to login. We once again use axios to send the credentials to the api to gather the information to be displayed on the home page. This information is the courses the user has completed along with the course department, course number, course name, semester date, and gpa. The information is used to calculate the degree requirement progress of the user. The information is sent to the html file as Json data where handlebars is used to iterate and inject the classes into the view.



During group advising students will be directed to the advising tab. It is here where they can submit the necessary information to release their hold. The requirements for this page is the date of the advising session, the name of the advisor, the list of classes they plan on taking, and the authentication code to prove that they were present.

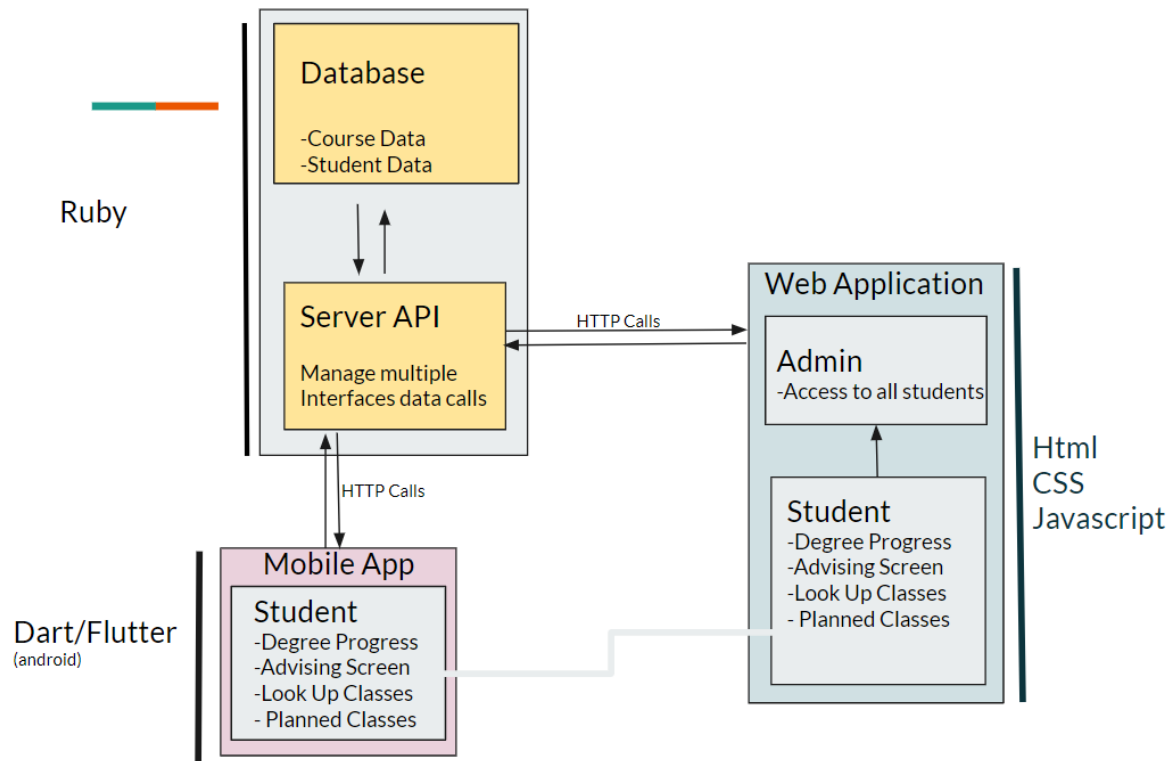
To add courses the student plans on taking they must go to the future courses section and select the course by section and use the dropdown menu to select the semester. When the user has selected the courses they want for that category they hit the submit button and the form sends the classes to the node server which calls the api and adds them to the planned future courses table with the student id. When the student goes back to the advising section the courses will now appear there with the option to remove them.

The user profile page is used to view the students information. This page is also used to display the students information for the advisor to see. The information in this page is the student name, email, gpa, classification, graduation date, catalog year, advanced hours, advanced cs hours, and total credits applied.

- Mobile (Flutter)

The mobile application was specifically designed for android , however porting to an IOS mobile device is very foreseeable as the flutter framework offers compatibility for ios devices. It allowed for deployment in any android phone as well as being a free framework with great

support and documentation from google. The main goal was to obtain and write information onto the database via a server API. This was done to have freedom for multiple front ends as shown.



Flutter allowed the freedom of custom UI design. The UI design was to mainly resemble the color scheme of UTRGV's official website by having a few main colors saved in their standard hex format. This would allow for a similar looking experience and straying away from the official google's material design, to give it a more authentic appearance.

The app uses many different routes or screens to take the user to different pages. After the first login page a token ,as aforementioned by the desktop GUI, was used to access the users

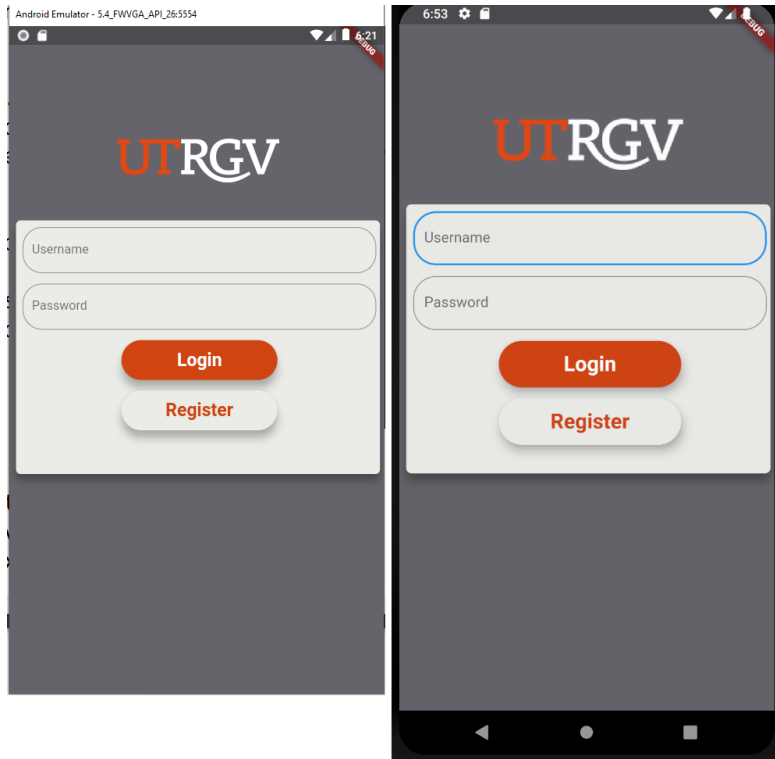
information. This form of authentication is required for every route and even containers within the screen each need a token.

The flutter basic routes can be seen here.



Flutter can work on almost all android devices, but android devices have a multitude of resolutions that is a hardware constraint. For this case the UI was designed to fit any screen, and keeping the aesthetic consistently on portrait devices. A query for the device's resolution is executed at the start of every screen. A grid is created to wrap on the devices screen and every device will have to go through the grid creation. Every element in the screen from the font to the size of each container to the screen is proportional to said grid. This was tested with two drastic resolutions. In the following figure we have a **480 x 854** resolution on the left

compared with a **1440 x 2880** resolution on the right. As shown both have a dynamically proportioned UI.



The mobile app has specific ways to provide courses and information, of course with the inclusion of the token. Our database holds a user table and a multitude of course tables. Multiple API provided by the Ruby Server can serve to obtain said information from the data. This data is then given to any front end choice. In the case of the mobile app, a class object is ready to be created for both a student and a course. Because a student can be a single object only one object is needed. But a list of courses is needed, so a List of objects is created within flutter ready to be read by the widgets. It could be said that mobile app has its own backend where it had to create objects, and hold functions that would talk to the more graphical intense side of the application.

```
Future<List<Course>> _getCourses() async {
  String value = await storage.read(key: "token");

  var response = await http.get(
    "$address/all/Courses",
    headers: {HttpHeaders.authorizationHeader: "Bearer $value"},
  );

  var data = json.decode(response.body);

  List<Course> courses = [];

  for (var i in data) {
    Course course = Course(i["CourseID"], i["CourseDept"], i["CourseNum"],
      i["Name"], i["Institution"]);
    courses.add(course);
  }
  print("This is the number of courses => ${courses.length}");
  return courses;
}
```

The code above is a Flutter Function which calls in for a response within the yellow box.

```
# Read all courses from AllCourses Table
get '/all/Courses' do
  api_authenticate!
  halt 200, AllCourses.all.to_json
end

# > models > AllCourses.rb
1 require 'data_mapper'
2
3 class AllCourses
4   include DataMapper::Resource
5
6   property :CourseID, Serial
7   property :CourseDept, String
8   property :CourseNum, Integer
9   property :Name, String
10  property :Institution, String
11 end
12
13
```

This is the Ruby Code which reads the database entries, in green, and assembles it into a list of json ready to be returned back into the flutter function.

```

class Course {
  final int courseID;
  final String courseDept;
  final int courseNum;
  final String name;
  final String institution;
  final String grade;
  final String semester;
  //final bool taken;

  Course(this.courseID, this.courseDept, this.courseNum, this.name,
    this.institution, this.grade, this.semester);
}

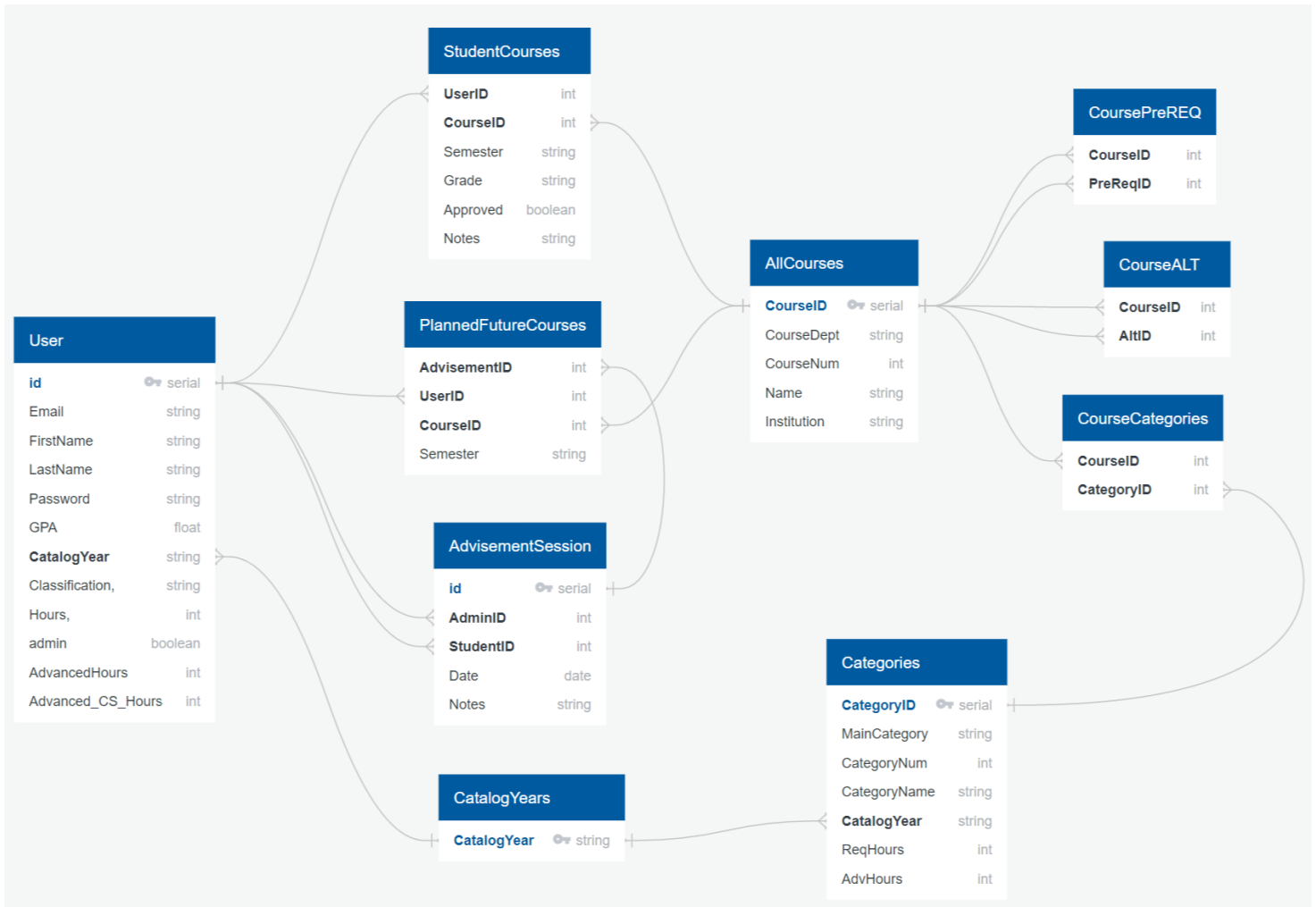
```

This is a flutter object that is being assembled by the first flutter function and being returned as a list of objects, ready to be used in the flutter app by widgets. This method of calling to the server and obtaining is common and repeated many times for objects such as planned courses , student information.

The operating Systems used to develop the app were both Linux and Windows. Linux was used to both develop the app and run the server. Windows was later then used to continue development of the app. Linux was used in the beginning for early compatibility with our server api as most gems and plugins are most efficient when running in a linux environment. Once the windows subsystem for linux was installed and our server was optimized for more deployable options a switch to windows was made due to comfort in working in a windows environment. Things to note is there is much less overhead in resources when working within linux so development was smooth early on and would have probably stayed running efficiently. Once the app became more efficient itself it did not require all that extra overhead and was able to run in a windows environment.

It was extremely important for the user information to be secure and be stored correctly, a package in flutter would be used to store tokens onto the physical device. This way the token is not floating around in the interwebs with the risk of being obtained or hacked. It is only called when needed and kept to a minimum at exposing it within urls. The token was stored on the device as it is the most secure place for this application. It is using Android's keystore system which is a container within the device that is extremely difficult to extract from. It also provides extra security features and is used for cryptographic operations when storing , in case the user has malware within their device. All in attempts to keep the users information private and information free from being tampered with , which was of the highest concern.

Database Model



Our database model was designed for students and administrators, and having them be able to manage advising and degree auditing. With advising and degree auditing in mind, we needed a way to store courses, course categories, course prerequisites, degree plans by year, and a way to track advising.

User - Our User table holds both students, and administrators, with the admin attribute used to differentiate them. We chose one table for both types of users to make signing in and registering simpler, only querying one table. This table holds some useful student information as well, GPA, CatalogYear (degree plan they're on), Classification (Freshman, Sophomore, etc), Hours, AdvancedHours, Advanced_CS_Hours.

AllCourses - The most obvious table that's needed is one to hold all our courses, AllCourses. AllCourses has course names separated into sections, CourseDept

(CSCI, MATH, etc), CourseNum (1380, 3300, etc), Name (Computer Science 1, Software Engineering 1, etc), and Institution (if we want to hold courses from schools with the most abundant transfer students).

We have two tables forming relationships between User and AllCourses, StudentCourses, and PlannedFutureCourses. PlannedFutureCourses relies on the AdvisementSession table, needing a record to be created there first.

StudentCourses - StudentCourses holds courses a student has taken, along with their grade, and semester taken. The table stores a students UserID (student ID number) and CourseID (from AllCourses table) to form the relationship between User and AllCourses. The last two fields in StudentCourses, Approved, and Notes, are in case a student tries to add a course they've taken, but there is some sort of restrictions that wouldn't let it get approved automatically (Think skipping a prerequisite), therefore the notes are there so a student could provide their reasoning, and an administrator could approve/deny the course based on those notes.

PlannedFutureCourses - This table holds courses a student plans to take, and on which semester. Like in StudentCourses, it uses a students UserID, and a courses CourseID, from the User and AllCourses tables, respectively. The AdvisementID attribute is from the AdvisementSession table, basically, a student needs to be present with an admin (1 to 1, or group session) to make entries to the PlannedFutureCourses table.

AdvisementSession - This table forms a relationship with two users, a student, and an admin, along with the PlannedFutureCourses table. This table was created with the requirement of a student being advised by a staff member in mind, therefore allowing entries to the PlannedFutureCourses table once an advisement session is created. The table also holds the date, and any notes the student or admin may wish to enter.

With degree auditing, we realized we needed to be able to manage courses, and several aspects about them. The most obvious requirement was how were we going to keep track of a users degree progress, courses needed a way to be organized into their respective categories, along with prerequisites. CourseCategories is a table that forms a relationship with AllCourses and Categories. There's also two simpler tables, CoursePreREQ, and CourseALT.

Categories - The Categories table is used to store degree plans, and all the categories in them. Using the Bachelor of Science in Computer Science 2019-2020 degree plan, the MainCategory attribute holds the main category, such as:

“A - General Education Core”, and “B - Major Requirements”. CategoryNum is for the category numbers, such as the “020” in “020 - Mathematics - 3 hours”. CategoryName would be the “Mathematics” in the aforementioned example. CatalogYear would hold the “2019-2020” catalog year, from the CatalogYears table. With ReqHours and AdvHours holding the required hours, and required advanced hours for a category, respectively.

CourseCategories - This forms a relationship between AllCourses and Categories, holding the courseID from AllCourses, and the CategoryID from Categories. We’ll be able to tell whether a course belongs under the Computer Science Core, or under Mathematics, for example, being able to accurately credit a students degree progress.

CoursePreREQ - Simple table that Makes AllCourses have a relationship with itself. CourseID is a course with a prerequisite, PreReqID being said prerequisite.

CourseALT - Another simple table created in case we want to add acceptable course substitutions. This table would hold a course, and it’s substitution in CourseID, and AltID, respectively.

During the process of designing our front ends, we noticed that it would be inefficient to query the Categories table for all the CatalogYear entries, and removing duplicates. An admin looking up older degree plans by year is one example of this query being needed. To simplify this query, we created the CatalogYears table.

CatalogYears - This table, as the name implies, stores catalog years. Tables, User and Categories, use this table for their CatalogYear attributes. As already mentioned, this table was created to simplify queries requesting all catalog years.

Possible Future Tables

As this degree auditing system is still an unfinished endeavor, with front end requirements changing periodically, there’s a possibility of the database model undergoing some adjustments. One potential future addition is a Semesters table.

Semesters - This table would hold semesters, Fall “year”, Spring “year”, Summer I “year”, and so on, with “year” being replaced with the actual year. I see this as a potential add on to ease the front end from creating their own values when submitting entries to the tables that have Semester as an attribute. They would instead just get actual, available semesters from the Semesters table.

Back End

When starting this project, we discussed the best method of implementing our back end and database. The majority of the members had experience working with Ruby, Sinatra, JSON, and DataMapper. Getting our API up and running was much easier than starting from scratch because of our shared experience using these resources.

- **Ruby** - The programming language used to write the back end was done on Ruby. “Ruby is a dynamic, open source programming language with a focus on simplicity and productivity.”(“Ruby”). Since Sinatra and DataMapper are Ruby Gems, it made sense to use this language, as we’d be able to create everything we needed with it.
- **JSON** - “JSON (JavaScript Object Notation) is a lightweight data-interchange format.”(“Introducing JSON”). We used this format because of how universal it is. Our two front ends, and back end are able to communicate and exchange data that’s easily parsable with JSON.
- **Sinatra** - “Sinatra is a Domain Specific Language for quickly creating web applications in Ruby with minimal effort”.(“SINATRA”). It allows creating of routes, HTTP methods paired with url-matching patterns. This is the main workhorse of our API, taking and handling all the HTTP requests from our two back ends. Responses are done with halt code, and requested data is returned in JSON format.
- **DataMapper** - “Datamapper is an object-relational mapper that offers a unified interface for managing databases like MySQL, PostgreSQL, and SQLite”. For our API, we used PostgreSQL as the main database. Since DataMapper has many pre-defined methods to manipulate the tables of the database and support for PostgreSQL, this made it easier for us to do what we needed to do.
- **JSON Tokens** - User security is implemented using JSON based access tokens. “JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.”(Jones, Bradley, & Sakimura). These JSON tokens are also used to verify what type of user is logged in. This is useful because manipulation of some tables should only be allowed by administrators, such as creating prerequisites, or requesting information pertaining to students.

- Back End HTTP Routes

The structure of the actual code that handles HTTP calls is set up with a CRUD design, meaning Create, Read, Update, and Delete. The Create portion creates records on the database, this is where students add courses they’ve taken, plan to take, etc. Also,

admins create degree plans, add courses to the database, assign prerequisites, etc. These HTTP calls require a bearer token to be passed, the aforementioned JSON based web tokens. Required parameters are also checked, and they are then used to manipulate the database depending on the called function. An example, to create a course a student has taken, the HTTP call needs to pass the bearer token, and parameters: UserID, CourseID, Semester, Grade, and optionally; Approved, and Notes. The HTTP function will check if there's a logged in user via the provided bearer JSON web token, returning an error halt code if not. Afterwards, proper passed parameters are verified, UserID and CourseID are used to check if they're valid entries in their respective tables (User and AllCourses). Other parameters, such as Semester and Grade, are checked to make sure that they're not empty strings. Error, or approval halt codes with messages are returned to the front end depending on the param check results. If every check passes, there's a halt 200 code with a successful message returned.

The Read portion of CRUD contains HTTP calls that return requested data. The Update portion takes in parameters, and edits existing records, whereas the Delete portion deletes records based on the provided parameters. The same principle applies in that a bearer JSON token is verified, then parameters are checked for validity. Halt codes are used to provide success or failure messages. On the Read portion, the requested data is returned in JSON format.

Testing

Mobile testing was done on Android 7 (Nougat), Android 8 (Oreo), Android 9 (Pie), and Android 10 (Q). The developer preview of Android 11 (R) was not compatible and held to many issues with testing. However this could be because Android 11 has not been officially released to any android device and is purely for testing and development. IOS Devices could not be tested due to not owning a Mac and not owning any apple developer licenses.

Initial back end testing was done using Postman. Postman allows making HTTP calls on local or remote servers. Using Postman, we were able to test out all our API methods, making sure our database was being manipulated to our expectations and requirements. Postman's easy way of making API calls prevented writing unnecessary code on the front end, because it allowed us to spot bugs before wasting that time on Flutter or Node.js.

Results

Bugs

A mobile obstacle was attempting to request information from the server during the development stage. The local server could not be accessed by flutter with the most popular http requesting methodology, it would result in many socket exceptions and delaying of basic testing. A solution would be to host it publicly with a public address as that has been shown to work with the http calls. Upon further inspection in the mobile process the http calls could not access Class A or Class B addresses. Such as localhost and 127.0.0.1 which all connect to the hosts server. Upon this realization binding the server to a Class C address of 192.168.xxx.xxx immediately resolved this issue, allowing for the http requests from the device to the server to be in effect.

Overall Outcome

Conclusion

The mobile app could successfully pull any information needed from the database through the Ruby API. A fundamental framework is in place for the database and server. The mobile side also contains a fundamental course framework and can dynamically account for any courses and students being added onto the database. The frontends reached a level of user friendliness where information is direct and extremely familiar to the degreeWorks currently being used by the school system.

References

“Axios” <https://github.com/axios/axios>

“Dart API Docs.” *Flutter*, api.flutter.dev/.

“handlebars” www.npmjs.com/package/express-handlebars

“Introducing JSON.” *JSON*, www.json.org/json-en.html.

Jones, M., et al. “JSON Web Token (JWT).” IETF Tools, tools.ietf.org/html/rfc7519.

“Ruby.” *Ruby Programming Language*, www.ruby-lang.org/en/.

“SINATRA.” *Sinatra*, sinatrarb.com/.

