

Iota

A tool for managing student organizations

Cole Cassidy
Dr. Wang CS 489

Abstract

There are many organization management tools available on the market today. Some are purpose built and designed for specific applications. Some are more flexible and can be configured to fit the needs of a particular student organization. The problem is that there are not many purpose-built applications for student organizations and the more flexible systems require some sort of administrative oversight. Most student groups do not have the time or resources to maintain an IT service for their organization. Students need an application that “just works” and requires little to no configuration. The Iota solution features group membership, shared minutes/reports, user profiles, and a modern webapp. These features give student organizations the core tools they need in a reliable format that behaves consistently across various devices. Two programs were created to provide these features. The IotaEngine software runs on a webserver. It provides the API, interfaces with the database, and handles authentication. The IotaApp software is a compiled static website that is stored in the user’s cache. It connects to IotaEngine and provides the user interface. These programs were built using a combination of Angular2, Bootstrap, and Express.js. They are coded in JavaScript and IotaEngine runs in the Node.JS runtime. The database software used is MongoDB.

Contents

Introduction	3
The Database	3
The API server (iotaEngine).....	4
Express and Callbacks	4
Security, Middleware, and JWTs.....	6
API Summary.....	7
The Front-End iotaApp.....	7
The Angular Framework.....	7
Bootstrap	8
Application Overview.....	9
Application endpoints.....	9
./login	10
./register	10
./dash	11
./register/join.....	11
./usr/<mode>/<userID>	12
./org/<orgID>	13
./min/<mode>/<ID>	14
Supporting Application Files	14
app-routing.module.ts	14
user.service.ts	14
auth.service.ts.....	15
token-interceptor.interceptor.ts	15
Application Summary.....	15
Project Summary.....	15
Bibliography	16

Introduction

In order for Iota to provide a meaningful user experience and become a practical tool it had to be reliable and behave in a way comparable to other organizational tools available today. This realization is what led to the selection of the libraries used in the project. All libraries utilized to build Iota were selected for their use in software maintained by companies such as Google, Facebook, Netflix, Airbnb and more. The architecture of the application needed to be reliable, fast, and improvable. For these reasons, the project was split into the two software components of a Front-End user interface (iotaApp) and a Back-End server software (iotaEngine). By splitting the software into its two core elements, the server load is reduced significantly as rendering the user interface is handled entirely by the user device. This allows the server to be built in a stateless way where sessions are replaced with authentication tokens. This increases the response speed of the server, as no local variables are referenced in processing responses. By using a document driven database (MongoDB) rather than a more traditional SQL database response time and compatibility is further increased.

The Database

MySQL and other SQL variants are an incredibly common database type and continue to be used in many large-scale projects. However, the relational nature of data stored on SQL servers can be a downside. Complex data can be difficult to store and recall with accuracy, and while many programming languages are object oriented, SQL is not. The benefit to a document driven database is that entries are treated as objects with associated values, rather than rows of values. The following screenshot depicts how an object is stored in MongoDB.

```

  _id: ObjectId('625cd6c0c314a7ae765e0a6d')
  uName: "TENBUCK4"
  fName: "Tucker"
  lName: "Barco"
  rank: 0
  joindate: 2022-04-18T03:10:56.453+00:00
  orgs: Array
    0: Object
      Name: "Sigma Nu Iota Beta"
      oid: ObjectId('6234ec073b13b586a6b369fc')
    1: Object
      Name: "VWU eSports"
      oid: ObjectId('6234ecf13b13b586a6b36a05')
  bio: "Current president of the Iota Beta Chapter of Sigma Nu Fraternity"

```

Figure 1- Document Database Example

This is a complex dataset that represents a user account. In a SQL database, the array of organizations would typically be a reference to a separate table of organizations. This would make it simple to create changes across all user accounts but requires either multiple or complex queries to return a dataset containing the same information. By storing the user data in the document format, all needed values are accounted for, and ID that references a different object can be stored in case additional retrievals are needed for more verbose information. This

reduces the number of API endpoints needed, allows for faster retrieval of essential data, and decreases the computational load placed on the database server.

The API server (iotaEngine)

The last several years have seen a shift in how webservers are built. Historically a webserver could be static, where it provided simple HTML, CSS, JS documents to the client to render a website, and there would be no dynamic content on the site. Alternatively, a dynamic website would send unique content to different users based off provided input. These dynamic sites were typically built in PHP. While PHP is still commonly used, the transition to native phone applications, and websites that share the same data has created a new web topology. API servers are built to send simple data from an application to the user interface such as a native phone application or webapp. The receiving application which made the request processes the response and adjusts the interface accordingly.

By removing the burden of rendering from the webserver, it is free to solely handle these API requests and authentication. The decision was made to develop the API server in JavaScript as MongoDB returns documents as JavaScript objects, and any web based front end would be able to process them using its own JavaScript environment. This meant that the server could easily form responses without frequent variable type conversions.

The most common runtime for server-side JavaScript is Node.JS. It is reliable, high speed, and there are a vast array of open-source libraries built for it. One of these libraries is Express. Express is a webserver library that allows developers to create webservers that return the output of different functions. The following code excerpt demonstrates the creation of an express webserver.

```
var express = require('express'),
    app = express(),
    port = 3000;
```

This segment creates a new express instance named "app" which listens on port 3000

```
app.get('/auth', (req, res) => {
  res.send("Future Authentication Endpoint")
  console.log("Auth Request Received");
});
```

Express and Callbacks

The second code segment appears more complicated but is actually rather simple. The Express `get()` function creates a new endpoint for HTTP get requests. It takes two inputs, a String for the URL path to listen on, and a callback function.

A callback function put simply is a function provided to another function to execute in place of a return statement. While it could be provided as a standard function it is possible to

use the error operator and define the callback inline. Express executes callbacks with the inputs *req* which contains the request data and *res* which can be used to send responses to a request. In short, the above function sends a response of “Future Authentication Endpoint” whenever a get request is received at *localhost:3000/auth*. The following screenshot depicts the browser response.

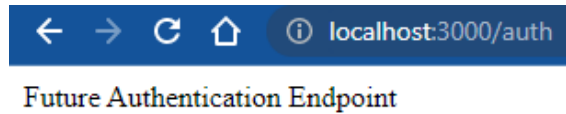


Figure 2- Simple API response

Developing in this manner allows for the creation of custom functions to be executed for each endpoint. These functions can call other functions to create a request and therefore allow for complicated responses that may compound multiple database values before ultimately returning an object. The following code demonstrates this behavior.

```
app.get('/user/:uid', async (req, res) => {
  try{
    console.log("User Request: "+req.params.uid+" From: "+req.ip);
    const userInfo = await users.findUser(req.params.uid);
    res.json(userInfo);
  } catch(e) {
    console.log(e);
  }
});
```

When the user makes a request to “/user/:uid” where *:uid* is the unique ID referencing a user document. The callback function logs the requested uid and the request machine's IP to the console. It then creates a *userInfo* object containing the response from the custom *findUser()* function which takes the *uid* as input. Once this function returns the document to be sent, the server responds with the *userInfo* object. The following screenshot depicts the browser response.

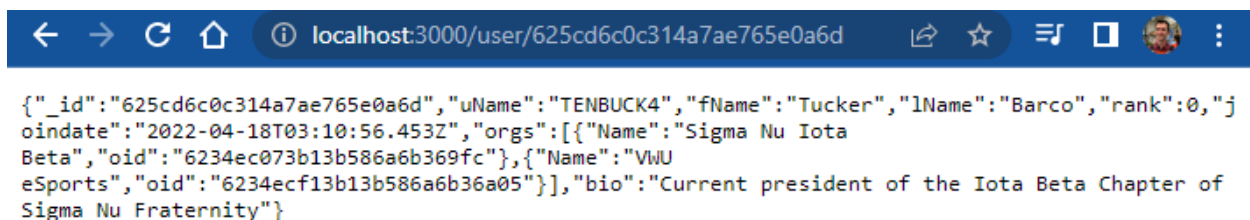


Figure 3- complex JSON response

Note that this is a raw JSON object which matches the previous database example of a user document.

Security, Middleware, and JWTs

In order to secure certain endpoints of the API, the user is provided with a JSON Web Token (JWT). This token is a simple JavaScript object, which is then encoded and signed by the server. It contains user information and is used to authenticate the user. The flow for JWT authentication is as follows.

1. The user sends the username and password for authentication.
2. The server validates the password against the database
 - If the password is valid it continues.
 - If not, it returns an error.
3. The server retrieves the document matching the username from the database.
4. The server encodes the document and generates a signature using a private key.
5. The server responds to the user with the signed token.
6. The user includes this token with all subsequent API requests.

Whenever a request is made to a secured resource, the server checks that the payload of the token matches the signature. If the token is a valid match the server uses the payload to confirm access to the desired resource. If there is no match, the token is considered invalid, and the connection is terminated.

The use of JWTs allows the service to be secured without the use of sessions. As long as the token is valid, access is permitted. This eliminates the need for tracking connection information in server memory or repeated authentication database lookups. Without the weight of session tracking and processing, the server runs with less overhead, and at a greater speed than cookies or sessions allow.

In order to authenticate the tokens with each request custom middleware functions were written. The Express library also provides a method for “middleware” these are functions that process input before the final callback function. The following code is an example of using middleware in an Express function.

```
router.get('/:oid/full', validator.checkToken, validator.authUser, async (req, res) => {
  res.json( await findOrgVerbose(req.params.oid) );
})
```

In the above example, the *validator.checkToken* is run first to check the token validity. Then the *validator.authUser* function is executed to ensure that the user has sufficient privileges. So long as neither of these functions returns an error, the callback function is executed. Additional details on these functions can be found in the codebook.

API Summary

By building the iotaEngine API server in JavaScript, and utilizing the Express library, the necessary functions of the API server are performed with speed and security. The design of the codebase allows for modular development, where new features and endpoints can be developed quickly.

An overview of the API endpoints is included in the codebook.

The Front-End iotaApp

Static webpages have historically had little use for projects with changing data. As mentioned in the previous section, the solution for websites that needed to have customized information was to use PHP. However, as the processing power of personal computers has expanded and JavaScript has risen in popularity, it has become more common to use “WebApps” in place of PHP servers.

What sets a webapp apart from a traditional website is that a webapp is a static webpage. The HTML, CSS, and JavaScript files for the website do not change after being loaded into the browser. In a traditional website, JavaScript would be used to add animations and effects. In a webapp, JavaScript controls the entire user experience. Functions are used to retrieve information from outside sources, and to update the user interface accordingly. This removes the rendering workload from the webserver. It also allows for faster load times, as the site can be cached to the user’s computer and recalled without loading the entirety of its contents from a remote server.

The Angular Framework

Building a reliable webapp from the ground up is an exhaustive task. Fortunately, there are several frameworks with which to do so. React.JS, Vue, Angular, and more are open-source projects which provide the groundwork for building modern web applications. Each one has its own benefits and drawbacks. Angular was selected for this project for several reasons.

- Extensive documentation
- Built-in development environment
 - o Debugging tools
 - o Component Compilation
 - o File minimization
- Built-in development server

While other frameworks offered similar features, Angular seemed to be the most commonly used framework. Its website includes multiple introductory projects, examples and guides, as well as thorough developer documentation.

To develop with the Angular framework, it must be installed on the local machine, after doing so it can be used to generate the file structure needed for a project. There are several

command line utilities that handle compilation, the development server, and minimization. The development and coding take place in the developers text editor of choice. It has built in debug tools that can be used by an advanced text editor to check for common errors. The environment also includes a compiler that combines multiple JavaScript, TypeScript, Stylesheet, and HTML files into individual files for deployment. It also can minimize these files by replacing human-readable variables with shorter names, removing unnecessary newline and space characters, and removing unused functions. These features are useful not only for building a reliable application, but also for reducing file size to decrease loading times.

The local development server allowed for rapid development. Once running, it hosts the webapp so that it may be tested in a browser, and it monitors the files so that the site is refreshed as changes are made.

All these features made Angular an ideal environment to build the *iotaApp* front-end. Using its component driven design, each core element of the site could be developed, and then integrated into pages to allow for dynamic content with reusable code.

Bootstrap

While Angular makes building web applications more convenient. It does not provide any type of styling, UI customizations, or color palette. Angular does support CSS, LESS, and SCSS or “Sassy CSS”. LESS and SCSS are variations of CSS that are more friendly for building large stylesheets, and they allow for variables, imports, and basic math. For this project the focus was on creating a usable application, and while an interface was necessary, there was no time to waste on building a custom color palette and aesthetic button/form themes.

Bootstrap is a widely used “design framework” it provides a stylesheet and JavaScript library that strips away the differences between various browsers, and then applies several unique options for making modern interfaces. This includes a simple color palette, font selection, and customized page elements. Bootstrap is designed to allow developers to build and test a visually pleasing application, without wasting time on interface design. It also is designed to be modified. Its stylesheets can be overridden, and colors, shapes, and behaviors can be changed without rewriting the design.

Iota utilizes the Bootstrap 5 design framework by including its files in the import statements for the application. As part of compiling the application Angular includes the Bootstrap files and minimizes them with the rest of the code base. This made it easier to build a functional application while focusing on the programming that was the focus of this project. Future plans include modifying or replacing Bootstrap in order to build a more customized user experience.

Application Overview

The following flow chart demonstrates the navigation through the web application. The red lines indicate routes that require an authentication token to be used.

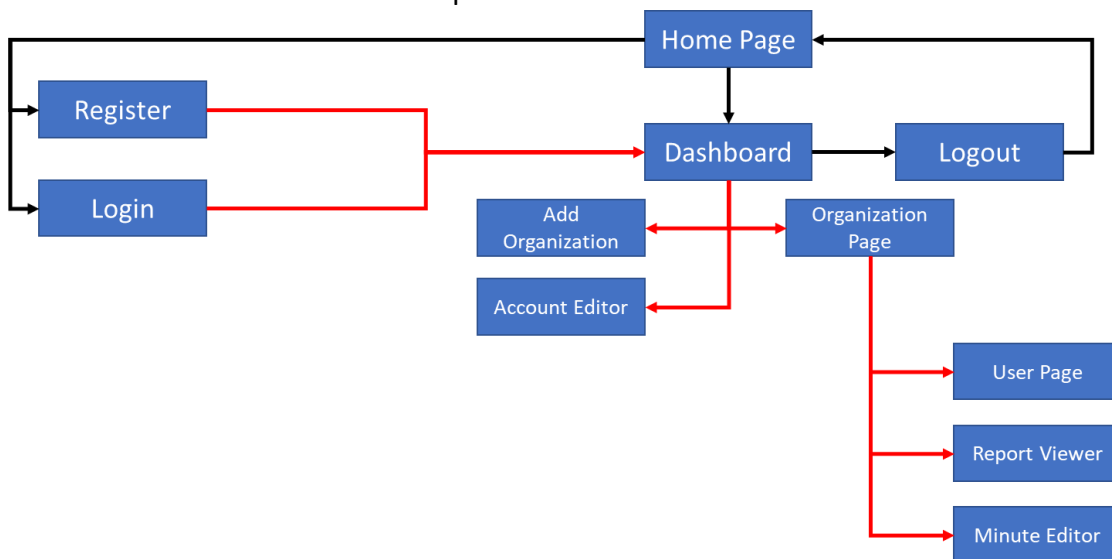


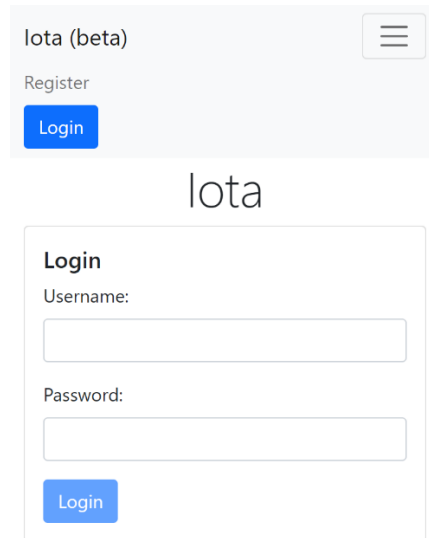
Figure 4- Application flowchart

Application endpoints

The following subsection provides screenshots of the Iota user interface, their url endpoints, and the context of each. The screenshots provided depict the mobile version of the application. However the behavior is

`./login`

This url endpoint provides the application login interface. Users input their credentials here to gain access to the application. This is provided by the *login-form* and the *authpage* components.



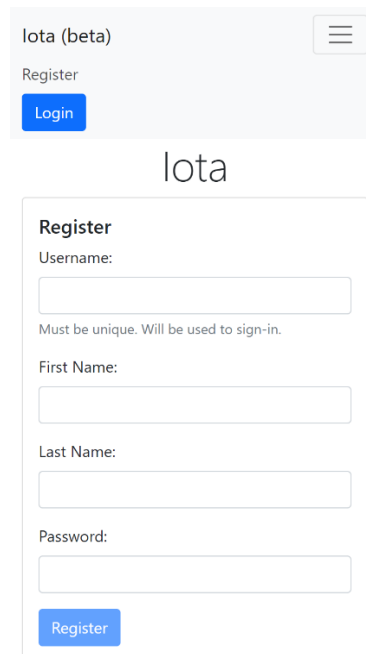
The screenshot shows the login endpoint interface. At the top, there is a header with "lota (beta)" on the left and a hamburger menu icon on the right. Below the header, there is a "Register" link and a blue "Login" button. The main content area features the "lota" logo in the center. Below the logo is a "Login" form with the following fields and buttons:

- Login** (Section Header)
- Username:
- Password:
- Login (Blue Button)

Figure 5- Authentication endpoint

`./register`

This endpoint provides a means for user registration. Creates an account and signs the user into the application. It is provided by the *regpage* and *register-form* components.



The screenshot shows the user registration endpoint interface. At the top, there is a header with "lota (beta)" on the left and a hamburger menu icon on the right. Below the header, there is a "Register" link and a blue "Login" button. The main content area features the "lota" logo in the center. Below the logo is a "Register" form with the following fields and buttons:

- Register** (Section Header)
- Username:
Must be unique. Will be used to sign-in.
- First Name:
- Last Name:
- Password:
- Register (Blue Button)

Figure 6- User registration endpoint

./dash

This application endpoint provides the user dashboard. It only populates with user information if the user has been signed in. Otherwise it protects the other endpoints by not allowing further access. It is provided by the *dashboard and org-panel* components.

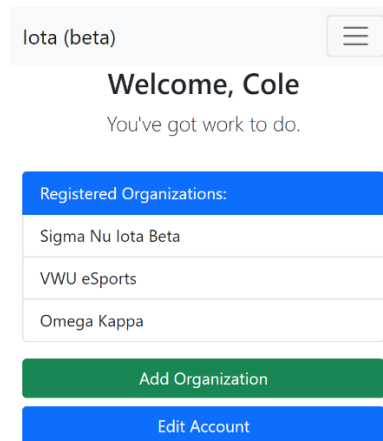


Figure 7 - Dashboard endpoint

./register/join

Provides an interface for users to join an organization using a registration code. This code is provided by a user that is actively a member of the organization the new user wishes to join. This is provided by the *add-org* component.

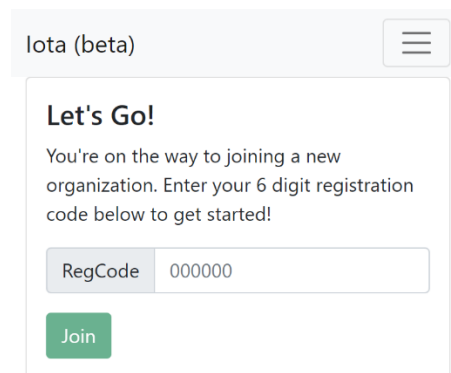
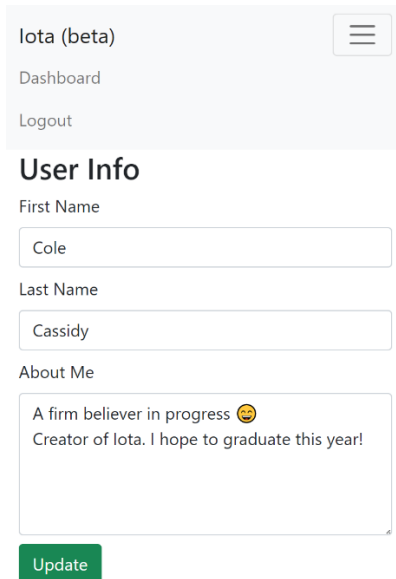


Figure 8- Registration endpoint

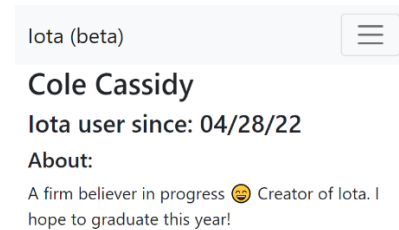
./usr/<mode>/<userID>

This is one of the more complex endpoints. It allows for a user to view and edit profiles. The *mode* variable in the path indicates whether the endpoint is in view mode (0) or edit mode (1). The *userID* variable selects the user that is to be edited. The API server prevents users from writing edits to other user profiles. The flowing screenshots demonstrate both these modes. These views are provided by the *user-panel* component.



The screenshot shows the 'User Editor' interface. At the top, there is a header with 'lota (beta)' and a hamburger menu icon. Below the header, there are links for 'Dashboard' and 'Logout'. The main section is titled 'User Info' and contains three input fields: 'First Name' with the value 'Cole', 'Last Name' with the value 'Cassidy', and 'About Me' with the text 'A firm believer in progress 😊 Creator of lota. I hope to graduate this year!'. At the bottom of the form is a green 'Update' button.

Figure 9- User Editor



The screenshot shows the 'User Viewer' interface. At the top, there is a header with 'lota (beta)' and a hamburger menu icon. Below the header, the user's name 'Cole Cassidy' is displayed in a large font, followed by 'lota user since: 04/28/22'. The 'About:' section contains the text 'A firm believer in progress 😊 Creator of lota. I hope to graduate this year!'.

Figure 10- User Viewer

./org/<orgID>

This endpoint displays information for the organization ID provided if the user has access to that organization. It includes a list of members that links to the user viewer for each member, a list of reports that leads to the report viewer, and a button for generating new minutes to report. At the bottom the application provides a “join code” to be used for registering new members. This is provided by the *org-page*, *user-panel*, and *minutes-panel* components.

lota (beta)
☰

Sigma Nu Iota Beta

The VWU chapter of Sigma Nu

Registered: 04/08/22

Members:

Barco, Tucker	
Cassidy, Cole	Me
Joslyn, Ian	

Reports:

Example Report
Port Day
Sentinel Report 4/28

New minutes

Join Code:

C26AC5

Figure 11- Organization Page endpoint

`./min/<mode>/<ID>`

This is another complex endpoint provided by the *minute-editor* component. It provides an interface for viewing and editing minutes. Mode 0 is used to submit new minutes, and the *ID* field is used to select which organization to attach the minutes to. Mode 1 is used to view reported minutes and the *ID* field is used to identify which report document should be displayed.

Figure 12- Minute Editor

Figure 13- Minute Viewer

Supporting Application Files

Angular builds web applications that defy the traditional website structure. The user navigates through router endpoints that display various pages. However, not all files in an angular project are represented as pages and there are many supporting files that are used to ensure the application behaves properly. The following section describes some of the most notable files that are not represented with a user interface.

`app-routing.module.ts`

This supporting file provides all the routes to various endpoints. When the user visits the site, this module determines which components to load to the viewport using the URL path. It also handles redirecting users when they land on an invalid endpoint.

`user.service.ts`

This file generates a service used by the application. A service in Angular runs in the background and shares its data with any component which imports it. In Iota, the user service serves as the connection between the application and the API server. When an API call needs to be made by a component, it executes a shared function in the user service, and reads the value from a shared variable within the service.

This topology is useful, as any component can request an API call, but all components subscribed to a variable effected by the result receive the change. It also centralizes all the functions needed to make API calls to one central file.

`auth.service.ts`

Similar to *user.service.ts* the auth service handles any API calls that deal with authentication. It maintains the login state for the application and stores the token for reference by other functions.

`token-interceptor.interceptor.ts`

The interceptor is another special component in Angular. Whenever the application makes an API call, the interceptor is run immediately before the request is made. It injects the current user token into the request header to ensure that the API is provided with the token on each request.

Further documentation on the utility of other project files is included in the codebook.

Application Summary

Building the application using Angular and Bootstrap allowed for more focus on the mechanics of the application, rather than rewriting commonly used functions. Angular specifically allowed for the development of modular elements which could be employed repeatedly in various locations in the application and thus improved code reusability. Bootstrap made building a user friendly interface easy, and allowed for quick deployment of developed features.

Project Summary

The Iota application has been a success. By employing modern design standards it has become a responsive, secure, user-friendly application. The project is currently hosted on Amazon AWS servers and is available at iotaengine.org. Future plans for the project include continuing to develop new features, building a more customized user interface, and rewriting portions of the codebase for increased efficiency. Using Node.JS and Express to build the API server has proven to create a stable, responsive server and has itself been considered a success. The Angular/Bootstrap frontend has proven to be reliable and performs equally well across devices. Early testing with students on campus has shown that they find the interface easy to navigate and use.

Bibliography

Amazon. (2022, April). *AWS*. Retrieved from AWS: <https://aws.amazon.com>

Auth0. (2022, April). *JWT*. Retrieved from JWT: <https://jwt.io/>

Bootstrap Team. (2022, April). *Bootstrap*. Retrieved from Bootstrap: <https://getbootstrap.com/>

Cassidy, C. (2022, April). *Iota*. Retrieved from Iota: <http://iotaengine.org>

Google. (2022, April). *Angular*. Retrieved from Angular: <https://angular.io/>

MongoDB. (2022, April). *MongoDB*. Retrieved from MongoDB: <https://www.mongodb.com/>

OpenJS Foundation. (2022, April). *Express*. Retrieved from Express: <https://www.expressjs.com>