# Shift, Global Research and Programmer Collaboration

A Thesis
by
ROBERT JOSEPH COMPTON

Submitted to the Honors College
at Appalachian State University
in partial fulfillment of the requirements for the degree of
BACHELOR OF SCIENCE

May 2017
Department of Computer Science

# Shift, Global Research and Programmer Collaboration

A Thesis
by
ROBERT JOSEPH COMPTON
May 2017

APPROVED BY:

Dr. Rahman Tashakkori
Chairperson, Thesis Committee

Professor Sharareh Nikbakht
Member, Thesis Committee

Dr. Rahman Tashakkori
Chairperson, Department of Computer Science

Jefford Vahlbusch, Ph.D.
Dean, The Honors College

## Abstract

Shift, Global Research and Programmer Collaboration

Robert Joseph Compton
B.S., Appalachian State University

Chairperson: Dr. Rahman Tashakkori

In the competitive job environment of software development, experience and a hard portfolio are the most valuable assets a person can have. A large number of students do not begin their career in development until they begin their higher education. This leads to the predicament where after graduation they often fail to meet requirements for positions even though they possess the skills to perform the job. Other times they are forced to take part-time or internship positions. At the same time, researchers in academia often create and store large volumes of data, yet lack the tools and/or knowledge to effectively utilize it. The goal of Shift is to create an online environment where said researchers can post the research goals and objectives for which they lack the ability to meet, so that programmers with the appropriate skills can pair with the researchers. These partnerships will benefit both parties, opening more effective means for researchers to achieve their goals while also building real-world portfolio and experience for the programmers.

## Acknowledgements

Sincere appreciation to my readers, Dr. Tashakkori and Professor Nikbakht for their willingness to read and provide feedback on the content of this thesis and project. My special thanks to the NSF S-STEM program for providing scholarship funding during my studies at Appalachian State University. I would like to also like to thank all of the authors of the research projects, documentation, and frameworks used in the development of this project (listed in the bibliography). Furthermore to the support of my family during my time of transition and duration of this project.

# Contents

# List of Figures

# List of Tables

# Chapter 1 - Introduction

This is a project driven thesis inspired by the disparity between the difficulty of gaining the real-world experience many software development jobs require and the high volume of need for these skills. Many of the jobs Computer Science graduates pursue after graduation require prior job experience, and often internship experience is not sufficient. Furthermore, there are many talented and skilled programmers who are still in primary education or who are simply not able to afford higher education.

The goal of this project is to create an online marketplace for researchers in need of computational skills and/or analysis tools to find those people who are capable of carrying out these specialized tasks. Congruently, it would be a place where programmers both in and out of academia can put their knowledge and skills to work on real-world applications and therefor gain valuable experience while building their resume. As the point is to offer opportunities to programmers while furthering research potential, the services provided by the programmers must be provided free of charge, and the research group or individual must give credit where it is due.

An example of such a collaboration would be as follows. Let there be a group of researchers with a large data set in irregular format, say some sort of big data collection. They would like to correlate this data set with some set of results in order to more accurately predict the behavior of the system in the future. A programmer with specialized experience and knowledge in unsupervised machine learning could design and provide a trained system for correlating the data, allowing the research to utilize the data effectively. In return, the aforementioned programmer would become a contributor to the research, gaining a real-world application experience and cited works on their resume.

A variety of tools and platforms were used in the development of this project. The project is implemented with MongoDB as the document driven back-end. Mongoose serves as an object modeling tool for the database, as well as the middleware. A NodeJS Express server hosts the middle-ware and back-end, and the data is exposed through a REST API. This server supports email verification and authentication of submitted projects and other content. The frontend is a client side application built on the Angular framework, accessing the data asynchronously via the REST API.

# Chapter 2 - Design

The first step towards the creation of the platform is the design. The key features, design choices, and reasoning behind them are described in the following sections.

## 2.1 Database

The first decision that must be made in regards to back-end is the choice of database. This decision is between a traditional relational database, such as a MySQL system, or a modern document driven database. While a relational database enables efficient mapping between tables and quick filtering for complex queries of the data, document driven databases enable a flexible, non-strict data format with hash map driven Javascript Object Notation (JSON) documents. These documents closely model the style of data storage used in web applications, and JSON objects are quickly becoming the standard for data transportation between applications.

Due to the shallow level of objects needed to be stored for this project, quick relational functions are superfluous. The goal of the back-end is to expose a REST API, discussed in Section 2.3. A document driven database allows close modeling between the application level and database level data consumption; hence, it is the database style of choice.

The server is built on Express in NodeJS[1]. Express wraps traditional NodeJS server routing, allowing for modular routing to handle various requests. This simplifies the code to handle different routing, such as separating the web server from the API, increasing ease the of maintenance. Such is the case in this application. The web server serves the application and the API route handles requests to the REST API.

As a document driven datbase is preferable, and the server is developed in NodeJS, MongoDB[2] is the database of choice. The server runs on NodeJS with Express, and the Node Package Manager (NPM) MongoDB package is a high level API for end users allowing an object driven connection to a MongoDB server.

## 2.2 Middleware

Middleware, like the database, exists on the server. It lays between the server and the back-end database storage. The job of the middleware is to handle any necessary pre-processing of data received from the server to ensure that it is in the proper format to be consumed by the database. Furthermore, it also handles the interpretation of non-serialized data, such as buffers, converting them to the proper format needed by the server platform, NodeJS, in the case of this project.

The progression of data flow as the server accesses data stored by the backend is shown below in Figure 2.1.

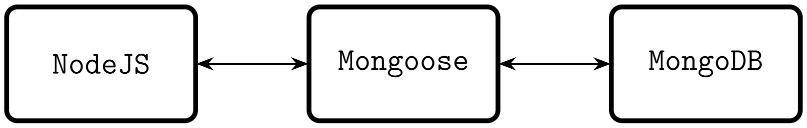


Figure 2.1: Dataflow between the Server and the Backend

Mongoose[3] is a middleware for MongoDB that allows the definition of document databases through schemas These schemas are derived from traditional JSON objects. Additionally, Mongoose creates an object class to represent instances of documents within a database, allowing the database to be used as if the documents existed as instances of this class. These classes are called models. As well as allowing construction of documents through a model, there are also static method to perform database wide functions. Mongoose models support pre-processing (middleware) and validation hooks to ensure database integrity and handle errors.

An exmaple of creating a document using a model is shown in Listing 2.1.

Listing 2.1: Example of Mongoose Model Usage

```
// Define the schema
let schema = new Schema({
    attribute: {
        type: String,
        default: "Default Value"
    }
});

// Create a model
Model = mongoose.model('databaseName', schema);

// Declare a new instance
let instance = new Model();

// Save new document instance to database
instance.save();
```

## 2.3 Application Programming Interface

The application programming interface (API) is used to expose data in the backend to applications which create, mutate, or otherwise consume the data. Web application APIs often support create, read, update, and delete (CRUD) operations. In this project, a Representational state transfer (REST) API will be used. REST is a standard for application interactions which is driven by the Hypertext Transfer Protocol (HTTP) and implements CRUD operations using the HTTP methods as they were designed to be used.

The process of mapping HTTP methods to CRUD operations can be seen in Table 2.1.

Table 2.1: CRUD Operations via HTTP Method

| CRUD Operation | HTTP Method |
|---|---|
| Create | Post |
| Read | Get |
| Update | Put |
| Delete | Delete |

An API route is used to define what data is being accessed through the REST API. In most cases, the last element of the path determines what data is being accessed, and requests which target a specific document rather than a database have an *id* appended. This identifier maps to the target document as a unique identifier, usually determined by the database.

For example, the REST API for the accessing projects could be implemented as shown in Table 2.3

Table 2.3: Example REST API for Projects

| Method | Path | Function |
|---|---|---|
| GET | /api/projects | Get all projects |
| POST | /api/projects | Add a project |
| GET | /api/projects/:id | Get a single project by :id |
| DELETE | /api/projects/:id | Delete the project with id :id |
| PUT | /api/projects/:id | Update the project with id :id |

In addition to these HTTP methods, interactions with the API can be further specified with various HTTP parameters and data. Requests without a body, such as HTTP GET and DELETE requests, can be customized by specifying query parameters embedded in the Uniform Resource Locator (URL). In this case, the query parameters are used to filter the data from the server. Examples of such parameters include limiting the number of items returned or filtering by some attribute, such as a date range. If data is carried in the body of the HTTP request, such as POST and PUT requests, then the header parameters are used instead. For requests which need to send an object and more complex data to the server, such as a project in a POST request, the data is stored in the body of the request.

For example, a project stored as a JSON object on the client application might be represented as in Listing 2.2.

Listing 2.2: An Example JSON Project Object

```
{
    'title': 'Shift',
    'author': 'Robert J Compton',
    'description': 'A platform for programmers to pair with researchers to further
        research potential while building experience and resumes for the programmers.'
    'tags: [
        'angular',
        'typescript',
        'node'
        ]
}
```

In this case, the project JSON object shown in Listing 2.2 would then be stored in the body of the request as shown in Table 2.5.

Table 2.5: Example Object as HTTP Request Body

| Key | Value |
|---|---|
| title | Shift |
| author | Robert J Compton |
| description | A platform for programmers to pair with researchers to further research potential while building experience and resumes for the programmers. |
| tags | ['angular', 'typescript', 'node'] |

## 2.4 Client Side Application

Implementing the API as a REST API allows for the frontend to be implemented as a total client side solution. This application will function purely on HTTP as the single protocol, both being served by HTTP and asynchronously carrying out CRUD operations through HTTP requests, without requiring further website data to be transferred.

The client side application will act as a place to access the posted projects. Various ways to filter and explore projects will be available, such as tag and institution filtering, date restrictions, and traditional text based searches. Markdown will be supported, allowing for versatile descriptions of the projects to be given, allowing the user to see information for each project. Relevant references and sources will also be present along with an email to contact the research group in question.

The application is developed on the Angular[4] platform. Angular is a total client side solution which is built on ES6 Typescript and Web Components. Its goal is to extend Hypertext Markup Language (HTML) to model application components for modern web applications rather than static content. The Angular application is written in Typescript[5], a superset language of Javascript developed my Microsoft in tangent with Google and the development of Angular. It supports optional typing and ES6 standards, which includes classes, decorators, and promises. Typescript transpiles to ES5 Javascript for distribution, using poly-fills achieve ES6 standards.

## 2.5 Data Structure

Various information must be kept on each project. It is desirable for projects to be grouped and filtered by relevancy to specific topics or other attributes. The actual implementation of the project data storage will be discussed in Section 3.7.

Projects will contain many values. Relevant details to be kept are listed below:

- *Title*: The title of the project.
- *Author*: The author(s) of the project.
- *Institution*: The institution or organization carrying out the research.
- *Date*: The date the project was submitted.
- *Short Description*: A short, non-formated summary of the project, for quick reference.
- *Description*: A markdown file giving a full description as needed for the project.
- *Tags*: A collection of tags which relate to the topic of the project.
- *References*: A collection of useful resources for understanding the topic of the project, including official and external information.

## 2.6 User Authentication

This platform is not meant to require user registration. To achieve this goal, email authentication will be used to verify the authenticity of posted projects. Since researchers are the intended audience to create the content (projects) of the application, it is safe to assume that they will have a publicly available email address belonging to a top level .edu domain. Hence, for a project to be added, a .edu email address will is required to be provided. A confirmation email will be sent to finalize the projects addition. If at any point the project needs to be modified or removed, email verification will be the method used to validate those actions as well.

# Chapter 3 - Development

The following sections describes the final state of the project and its structure.

## 3.1 Project Structure

This project consists of a full web stack, which contains a backend database, middleware, a web server, an API, and a frontend client side application. The interaction of these components can be seen below in Figure 3.1.

$$\begin{tikzcd}
& \texttt{API} \arrow[ld, leftrightarrow] \arrow[d, dash, dashed] & & & \\
\texttt{Angular} & \texttt{NodeJS} \arrow[r, leftrightarrow] \arrow[d, dash, dashed] & \texttt{Managers} \arrow[r, leftrightarrow] & \texttt{Mongoose} \arrow[r, leftrightarrow] & \texttt{MongoDB} \\
& \texttt{Web Server} \arrow[lu] & & &
\end{tikzcd}$$

Figure 3.1: Data Flow Across the Stack

## 3.2 Document Schema

The final structure of the database was close to as discussed in Section 2.5. In this section the schema defining each of the databases can be found.

The schema for the project database is as below in Listing 3.1.

Listing 3.1: Project Object Schema

```
{
  _active: {
    type: Boolean,
    default: false
  },
  title: {
    type: String,
    default: 'Unnamed project'
  },
  author: {
    type: String,
    default: 'Unknown author'
  },
  email: {
    type: String,
    default: 'Unknown email'
  },
  shortDescription: {
    type: String,
    default: 'A short description'
  },
  description: {
    type: String,
    default: 'No description'
  },
  institution: {
    type: String,
    default: 'No institution'
  },
  date: {
    type: Date,
    default: new Date()
  },
  tags: {
    type: Array,
    default: []
  },
  references: {
    type: Array,
    default: []
  },
  sources: {
    type: Array,
    default: []
  },
  comments: {
    type: [CommentSchema]
  }
}
```

Note that there is an attribute here not mentioned in Section 2.5. The *_active* attribute in Listing 3.1 is used for authentication of projects. This will be discussed further in Section 3.5.

The schema for the comment database is shown in Listing 3.2.

Listing 3.2: Comment Object Schema

```
{
  name: {
    type: String,
    default: 'hahaha'
  },
  date:{
    type: Date,
    default: new Date()
  },
  buff: Buffer
}
```

The schema for the validation token database is shown in Listing 3.3.

Listing 3.3: Validation Object Schema

```
{
  project: {
    type: String
  },
  email: {
    type: String } }
```

Since the validation documents contain the email registered for the project as well as the project's identifier, the validation model may be searched by email, allowing a user to request that verification emails be sent again if so desired.

## 3.3 Middleware Managers

While Mongoose handles pre-processing and validation between the server and MongoDB, there is often more complex underlying data related logic which exists between the API and backend. For this reason, custom managers were implemented. The managers are constructed in the Express application as singletons, and accessed upon request by the

API routing. These managers wrap database operations or other functionality in ES6 promises, and provide high level application specific operations. The API route can use the high-level managers to achieve its end-user logic, and simplifies the task of the server to simply parsing the HTTP requests. Each manager always returns its work as an ES6 promise.

Different levels of managers exist, and some depend on others. The lowest level manager is the database manager. This manager handles the initiation of the MongoDB connection and constructing and maintaining the Mongoose schemas and models. The higher-level managers, such as the project manager, use the database manager and offer high level business logic operations specific to the data which they wrap.

The roll of the managers can be seen below in figure 3.2.

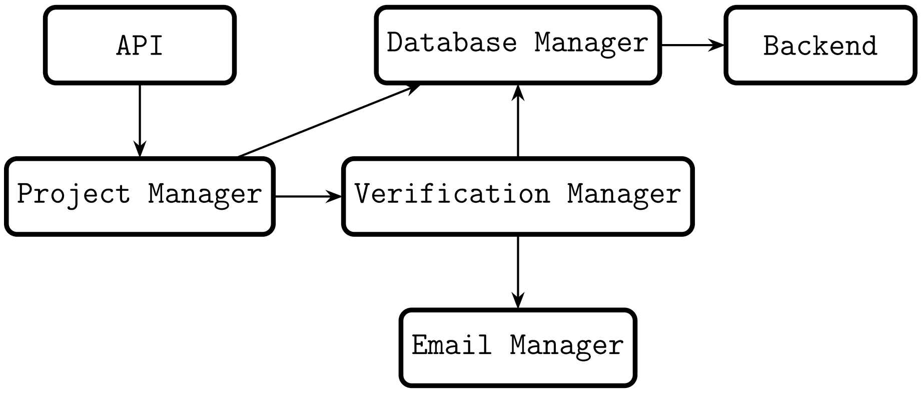


Figure 3.2: Manager Structure

The use of this manager structure separates the implementation of different parts of the stack from each other, while establishing a uniform way to achieve logic, ES6 promises. This allows for the underlying structure of any part of the stack to be adjusted without disturbing the business logic which is built upon it.

## 3.4 Application Programming Interface

Due to the scope of this project, it was not desirable to implement a full REST API for each database schema. Since not all CRUD operations are available to users, the API endpoints for those operations do not exist, but are maintained when needed using internal managers on the NodeJS server.

A complete listing of implemented REST API endpoints are shown in Listing 3.1.

Table 3.1: REST API

| Method | Path | Function |
|---|---|---|
| GET | /api/projects | Get all projects |
| POST | /api/projects | Add a project |
| GET | /api/projects/:id | Get a single project by :id |
| PUT | /api/projects/:id | Update the project with id :id |

As mentioned in Section 2.3, HTTP requests can be further specified. In this case, any and all attributes of a project can be used as filters by setting HTTP query parameters for the GET request to */api/projects*. There are many uses of this API feature in the client application. In addition to supporting server side filtering to minimize data transfer and optimize client load times, it also can also be used in more unique ways.

For example, while the title of a given project does not need to be unique, it is recommended and desirable that each title is unique. This aids in a user's ability to find projects and come back to them at a later date. An API query can be used to check for title conflicts without the application needing to hold all projects. While completing the project submission form, the application makes a GET request to the API, specifying the current given project title from the user as a query parameter. If the application receives a response containing any projects, it warns the user of this non-fatal conflict. Conversely, if no projects are returned, then there is not a conflict of title.

## 3.5 Authentication

As discussed in Section 2.6, no user records are kept by the application. Instead, email verification is used to determine if a project is valid to be added. By default, projects are added with the internal attribute *_active* set to the value of *false*. All API endpoints are filtered to only access elements with this attribute set to *true*; thus, projects are inaccessible past the middleware when first added.

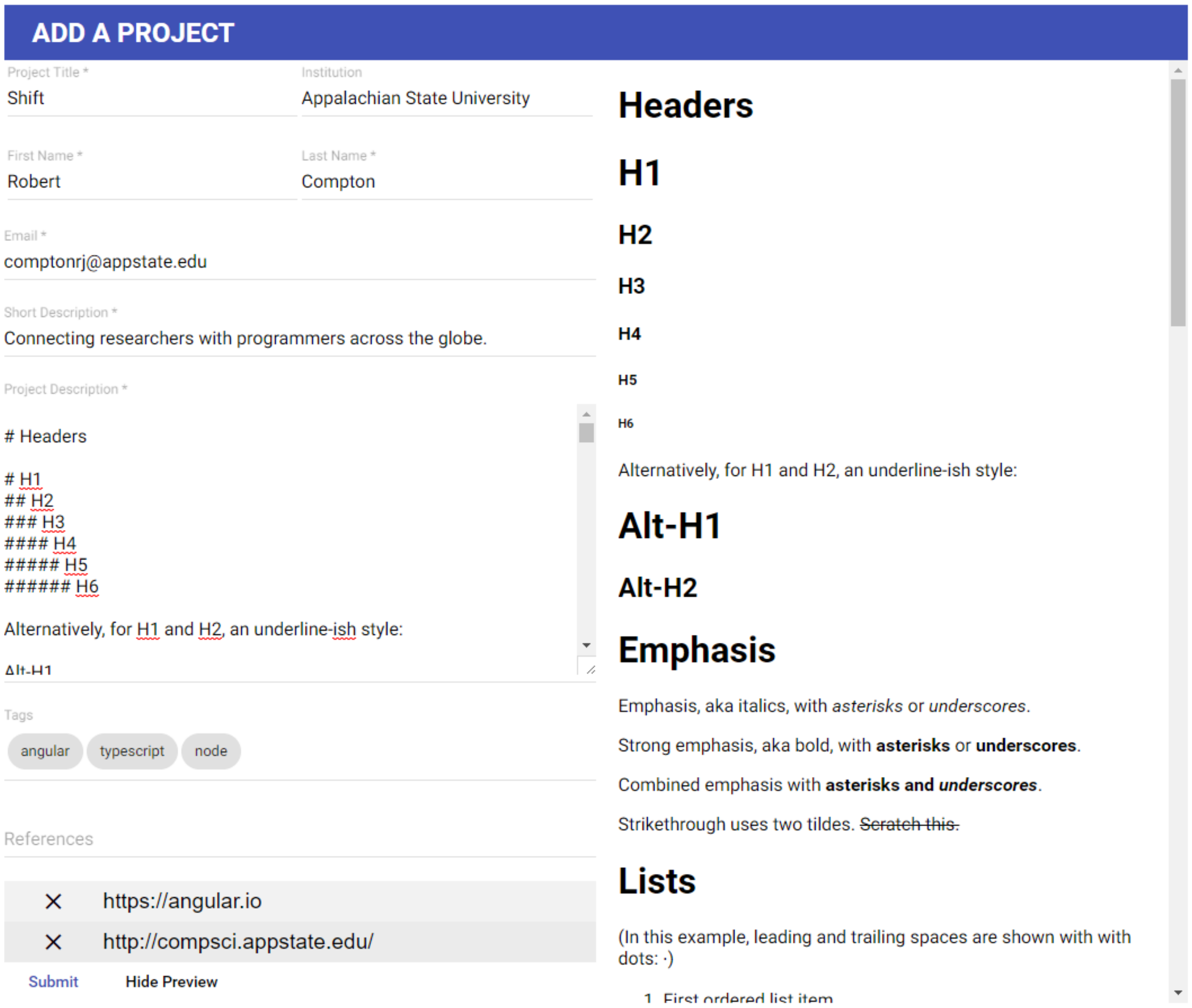
When a project is added, a random hash is created. This hash is used as the key to store the *id* of the newly created project in a document. When the path */verify* is accessed with a HTTP GET request, the HTTP query parameter with key *vid* is used to access the verification document of the same id. If the document exists, the project id stored is used to set the respective project's *_active* attribute to *true*, enabling it to pass through middleware. Finally, the validation document is removed from the validation database.

A hypertext reference is generated for this HTTP GET request, and is emailed to the address which was provided for the project in question. In this way, the user is able to activate their project.

## 3.6 Client Application

User interaction with the application consists of three primary activities. The first is the creation of content, occurring when a research group submits a project to the platform. The second is the browsing and filtering of said projects through the projects list. Finally, each project can be viewed in detail, including the markdown description, author, institution, and references.

An example of a project submission form ready to submit with a preview of the markdown description is shown in Figure 3.3.



ADD A PROJECT

Project Title *
Shift

Institution
Appalachian State University

First Name *
Robert

Last Name *
Compton

Email *
comptonrj@appstate.edu

Short Description *
Connecting researchers with programmers across the globe.

Project Description *

```
# Headers

# H1
## H2
### H3
#### H4
##### H5
###### H6

Alternatively, for H1 and H2, an underline-ish style:

Alt-H1
```

Tags: angular, typescript, node

References
- https://angular.io
- http://compsci.appstate.edu/

Headers

H1

H2

H3

H4

H5

H6

Alternatively, for H1 and H2, an underline-ish style:

Alt-H1

Alt-H2

Emphasis

Emphasis, aka italics, with *asterisks* or *underscores*.

Strong emphasis, aka bold, with **asterisks** or **underscores**.

Combined emphasis with **asterisks and *underscores***.

Strikethrough uses two tildes. ~~Scratch this.~~

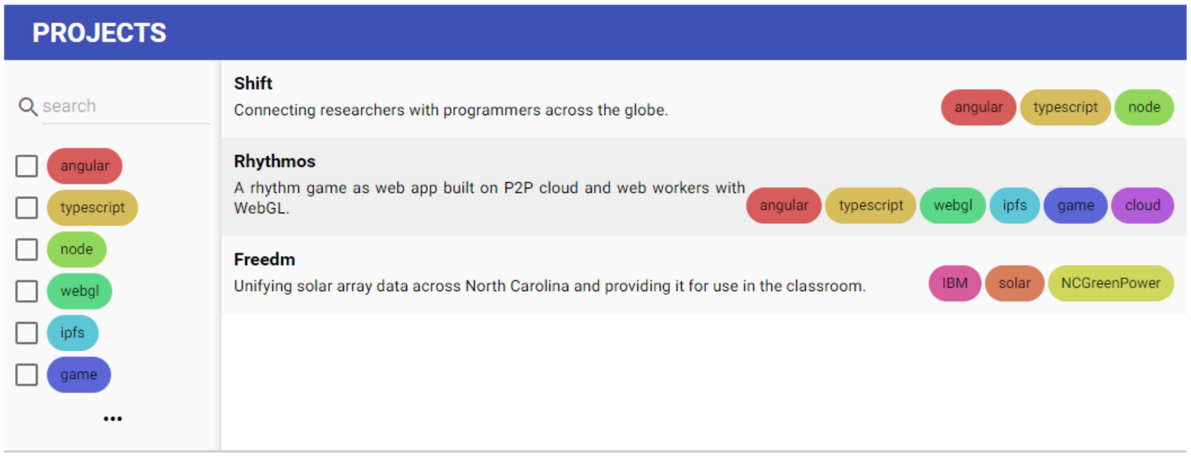
Lists

(In this example, leading and trailing spaces are shown with with dots: ·)

1. First ordered list item

Figure 3.3: Adding a New Project with Preview

An example of the project list without any filtering terms defined is shown in Figure 3.4.


PROJECTS

search

☐ angular
☐ typescript
☐ node
☐ webgl
☐ ipfs
☐ game
...

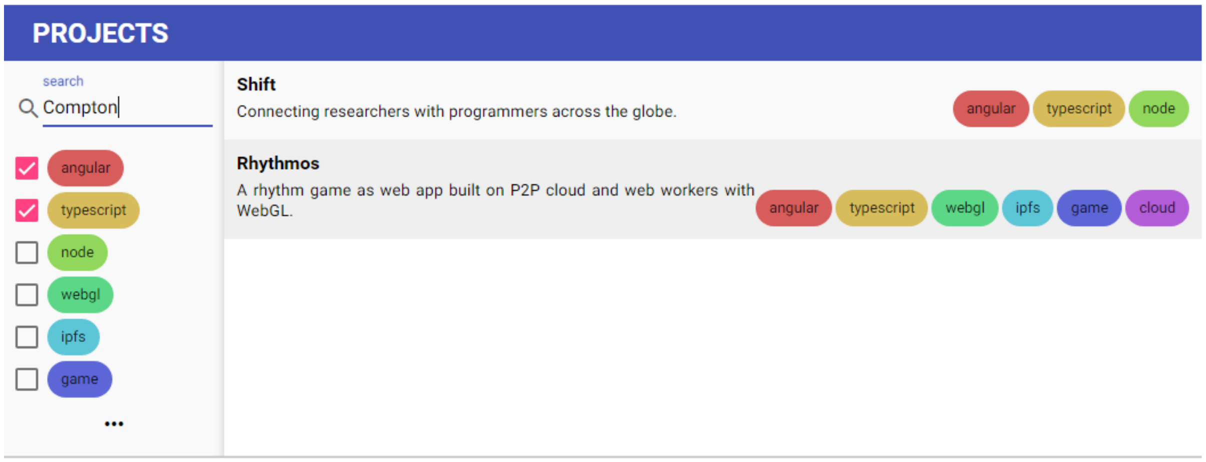
**Shift**
Connecting researchers with programmers across the globe.
angular typescript node

**Rhythmos**
A rhythm game as web app built on P2P cloud and web workers with WebGL.
angular typescript webgl ipfs game cloud

**Freedm**
Unifying solar array data across North Carolina and providing it for use in the classroom.
IBM solar NCGreenPower

Figure 3.4: Viewing Unfiltered Projects

Once filtering is applied by the user, the project list updates instantly to reflect matched projects. This can be seen in Figure 3.5.


PROJECTS

search
Compton

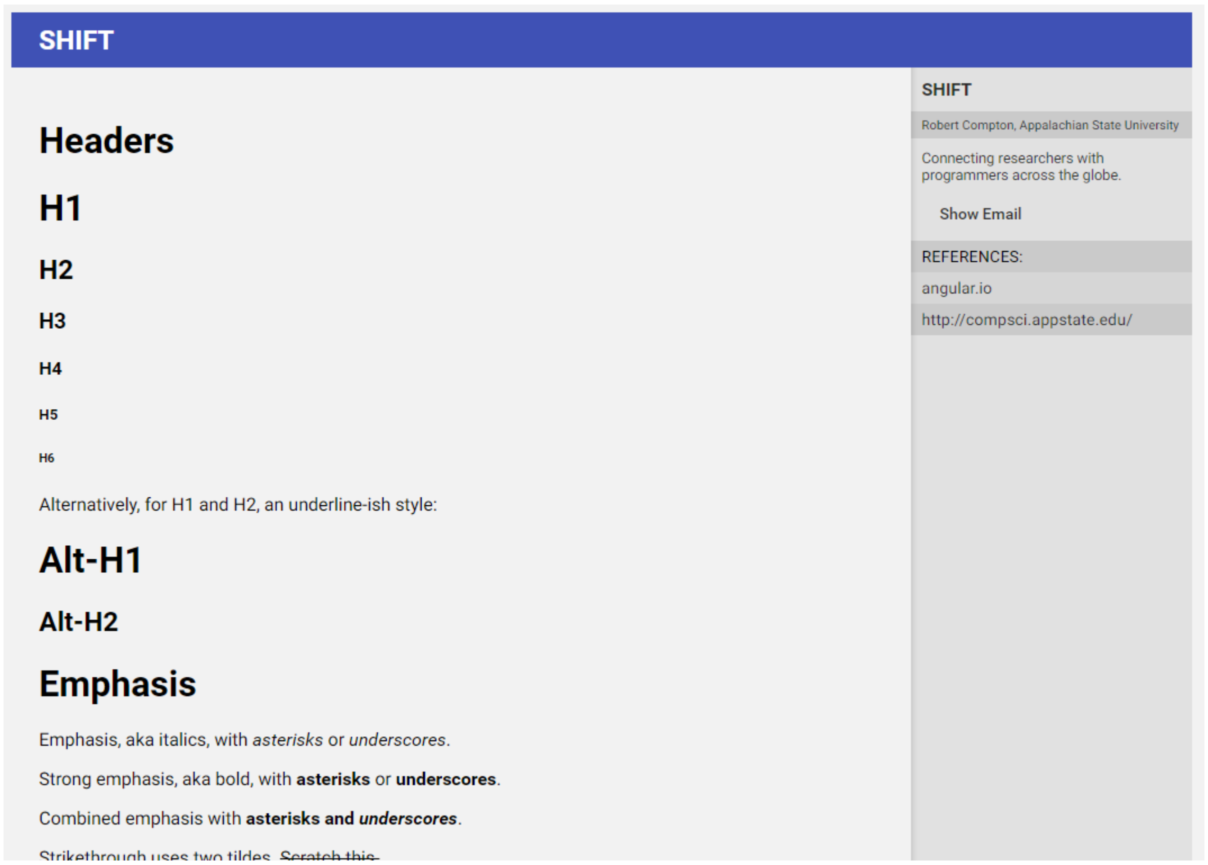
☑ angular
☑ typescript
☐ node
☐ webgl
☐ ipfs
☐ game
...

**Shift**
Connecting researchers with programmers across the globe.
angular typescript node

**Rhythmos**
A rhythm game as web app built on P2P cloud and web workers with WebGL.
angular typescript webgl ipfs game cloud

Figure 3.5: Viewing Filtered Projects

When a project is clicked on, the user is taken to the project's detail page. This page shows the description, references, and the contact email. This page is shown in Figure 3.6.



SHIFT

Headers

H1

H2

H3

H4

H5

H6

Alternatively, for H1 and H2, an underline-ish style:

Alt-H1

Alt-H2

Emphasis

Emphasis, aka italics, with *asterisks* or *underscores*.

Strong emphasis, aka bold, with **asterisks** or **underscores**.

Combined emphasis with **asterisks and *underscores***.

Strikethrough uses two tildes. ~~Scratch this.~~

SHIFT

Robert Compton, Appalachian State University

Connecting researchers with programmers across the globe.

REFERENCES:

angular.io

http://compsci.appstate.edu/

Figure 3.6: Viewing a Specific Project

## 3.7 Improvements

While the original idea of not requiring registration of a user account to use the service seemed like a good idea, this opinion has changed after the initial development. Due the likelihood of organizations or departments posting multiple projects, it is logical to allow an account to exist which could have a number of projects linked to it. Furthermore, as the need for maintenance on more complex or evolving projects arise, the email verification becomes a clumsy rather than convenient way to interact with the service. For this reason, a user account system should be added. Utilizing external authentication, such as Google, could maintain the goal of convenience.

While the search system for projects is adequate, it is still naive. A more sophisticated search engine would increase the ability for users to find relevant projects, and thus improve the effectiveness of the application in achieving its goal.

Also, while the backend to support comments on projects was implemented, the API was not added. The reason for this was that supporting comments on projects would benefit the application towards its goal in any meaningful way.

# Chapter 4 - Conclusion and Future Work

In conclusion, the idea of this project remains powerful, and the application as is achieves the goal of aligning the needs of researchers and programmers to some degree. While the basic needs for the platform are satisfied, more complex interactions and needs will arise, and the platform will need to evolve. There are many improvements could be made upon the current implementation, and many more will become apparent as time goes on. Despite these short comings, the project still achieves its goal.

The further development of this application will benefit both academia and growing developers. This platform will significantly reduce barriers of skill encountered by many researchers, and help prevent talented developers without exposure from being overlooked.

# Bibliography

[1] "Docs | Node.js." [Online]. Available: https://nodejs.org/en/docs/

[2] "MongoDB for GIANT Ideas | MongoDB." [Online]. Available: https://www.mongodb.com/

[3] "Mongoose Schemas v4.11.5." [Online]. Available: http://mongoosejs.com/docs/guide.html

[4] "Angular - What is Angular?" [Online]. Available: https://angular.io/docs

[5] "Documentation · TypeScript." [Online]. Available: https://www.typescriptlang.org/docs/home.html

# Vita

Robert J Compton was born to Robert and Marion Compton in 1995 in the State of Georgia, United States of America. He was home schooled until middle school, after which he attended various home school programs before graduating from public school. He attended Appalachian State University starting in 2013 and studied at *Kansai Gaikokugo Daigaku* in Japan towards achieving his Bachelor's of Science degree in 2017. He plans to pursue a Master's of Science in Computer Science at Appalachian State University and graduate in 2018.