

Developing a Cross-Platform Mobile Application

Bizi Marketplace

May 13th, 2018

Kyle Bargo and Bruno Rosa

Faculty Sponsor: Dr. Gerald Kruse

[**1. Project Overview**](#_30j0zll) **2**

[**2. Bizi Marketplace**](#_3zfc3c1rdmd8) **3**

[2.1 What is Bizi?](#_z9szwkeudxaf) 3

[2.2 Business Model](#_suv5ew77xmni) 3

[**3. Project Management**](#_2vmnrp5n5ayy) **5**

[3.1 Project Definition](#_k02ahicuug6s) 5

[3.2 Project Objectives](#_nwf6fgh1espm) 6

[3.3 Project Constraints](#_z24lgrdd4frn) 6

[3.4 Communications Plan](#_j1p9tw6fqes6) 7

[3.5 Milestones List](#_7m12rgcg9u79) 7

[**4. Identity**](#_k36wib54im63) **9**

[4.1 The Problem](#_re91jp36zae1) 9

[4.2 The Solution](#_3sq3dttqamq4) 9

[4.3 Google Survey](#_hanrbcg2l7oo) 10

[**5. Design**](#_5lirupqnqhsx) **11**

[5.1 Design Evaluation](#_eni8mrh60tw7) 11

[5.2 Design Discovery](#_63d0xkwd6rnx) 11

[5.3 Wireframing](#_y41du84q1wpd) 13

[**6. DevOps**](#_t8e48iyovxo) **14**

[6.1 Cross-Platform Frameworks](#_nkvnyg3bj6cl) 14

[6.2 Integration](#_f1s34c2fyhdu) 15

[6.3 Tools](#_s89im041e5pm) 15

[**7. Development**](#_9aitkngxi9ar) **19**

[7.1 Frontend](#_ibzwxlcn2xwc) 19

[7.2 Backend](#_3q0sqjjw19gb) 27

[7.3 Functionality](#_ryy9mae69qyi) 30

[7.4 Future Development](#_os079qw4y9rw) 33

[**8. References**](#_8h1butayovap) **35**

# 1. Project Overview

This semester, we have been working to develop a cross-platform mobile application that will be released on the iOS and Android platforms, as well as on the web. This application, Bizi, is a student-focused marketplace where Juniata College students can list products to be sold to their classmates. In order to accomplish this from the development side, we did thorough research into cross-platform frameworks, learned various languages, created mockups and wireframes, and utilized our technical backgrounds to create this application from the ground up.

Each of us had different roles throughout the project, with some assistance from others in various capacities. Bruno was the lead developer and worked with many of the technical pieces that this project had to tie together. Kyle served as the project manager and worked with other components such as the wireframing and design for the UI. For the business side of the project, Kyle was assisted by Dallas Huff, Terry Anderson, and JCEL. Finally, Jerry Kruse oversaw the project and research that we have delivered.

We both decided that we wanted to do a project that impacted students on campus and that would continue to progress once our research concluded. Due to this desire, we incorporated additional components into our project that would help to roll the application out to the campus community. Some of these include conducting competitor evaluations, surveys, and interviews that allow us to get feedback on what works and what doesn’t. As our target market is the students, this feedback is vital to the success of the application in the long run.

# 2. Bizi Marketplace

## 2.1 What is Bizi?

Bizi Marketplace is a mobile application that allows Juniata College students to buy, sell, and exchange products. These products can be anything within reason, such as books, clothing, and furniture. The application will allows students to sign up with their Juniata emails and upon additional information (name, dorm/location) and verification, they will be able to see products listed by their classmates.

## 2.2 Business Model

The business model canvas is made up of 10 components. We have already discussed the problem identification and solution in previous segments. These are the two most important parts of each proposal or plan, but the other components are also vital. They are as follows:

I. Customer Segments: all of the people who we are creating value for

Students

Faculty

Local Businesses

II. Competition: who is selling a similar product

LetGo

Mercari

eBay

OfferUp

Amazon

Akademos

III. Value Propositions: the value that we provide to our customers

Students: as students ourselves, we understand that students would benefit from an in-house application to buy and sell products.

Faculty: more students will bring books to class and be prepared to learn if they could get these resources at cheaper prices.

Local Businesses: By allowing local organizations to advertise within the app, we are able to provide this service to students at no cost, while also generating business and traffic to the Huntingdon area.

IV. Customer Relationships: how we keep our customers satisfied

Provide the students with a platform that is secure, reliable, and satisfactory in UX.

Provide the businesses with a simple way to advertise at an affordable rate.

Use a CRM to track relationships, Use a ticketing system to keep track of user issues.

V. Revenue Streams: how we maintain the business

Advertisements within the application that are paid for by local organizations.

VI. Key Resources: assets indispensable

Information to create and maintain the application, the application itself, and us developing the application.

VII. Key Activities: what needs to be done to be successful

Continue the development of the application.

Marketing to businesses and students.

Creating sales and using data to improve the product.

VIII. Key Partners: who will help to leverage our product

JCEL with funding

Jerry Kruse / Terry Anderson with mentorship

# 3. Project Management

## 3.1 Project Definition

This project would consist of four main components or phases that would lead to the completion and success of our project. These are identity, design, devops, and development. Below are descriptions of each of the components in detail:

In the identity phase, we discovered the problem, how we planned on solving it, what functionality was needed to solve the problem, and we also solicited feedback from students.

In the design phase, we researched similar platforms to see what they used, how they were set up, and created wireframes that would serve as the model for our development.

For devops, we researched cross-platform frameworks, what tools were needed for development, and how everything would be integrated together.

Finally for development, we constructed the backend functionality, built a frontend that would create a suitable user experience, and managed the resources needed to build upon the development after the project was completed.

Although a large percentage of the tasks for each phase were done sequentially, that was not always the case. At the end of the semester, all of these phases will be completed and we will move more towards adding additional functionality and design, rolling out the application, marketing it to both the students and local businesses in Huntingdon, and further maintenance.

## 

## 3.2 Project Objectives

During this semester, we are to complete the following objectives:

1. Research cross-platform mobile application development and languages involved.
2. Create a functioning mobile application on both the Android and iOS platforms.
3. Ensure that users of the application have a responsive UX by creating a suitable UI.
4. Document the process, all findings, and problems/solutions during the project.

## 3.3 Project Constraints

Below are the six most prevalent constraints or issues that arise during a project, the impact we expect them to have, and how we plan to move past them.

|  |  |
| --- | --- |
| Scope | The project scope will change as we start to develop the application. Some functionality might be difficult to program, so that may change as we move forward. |
| Time | We are full-time college students who have other classes and jobs on/off campus. Bruno is also in Pittsburgh for the semester, so location is also a similar constraint. |
| Cost | As we move further into development, the cost of the hosting, deployment, and maintenance could become an issue without funding. |
| Risk | There is a risk that the project may not be completed on time due to unforeseen factors. |
| Resources | Cross-platform mobile application development isn’t exactly new, but credible paper and resources aren’t widely available. |
| Quality | If problems arise, we might see a decline in quality due to the time constraint. We will avoid this by focusing on each component one at a time. |

## 3.4 Communications Plan

We will be meeting twice a week until Liberal Arts Symposium to discuss task designation, task progression, and problems that have arose during that time. There is also a weekly meeting between Kyle and Dallas to talk about the design of the application, marketing materials, and business management for Bizi LLC.

We have used Freedcamp as our project management software to keep track of the majority of our information. We have integrated with Google Calendar and Google Drive in order to manage our meeting schedule and files from the project. All tasks have been placed on Freedcamp and their progress is updated as work gets done.

## 3.5 Milestones List

This project has been very dynamic so we have taken the approach of “sprints” in order to meet our goals. We have a list of what we hoped to accomplish in each week, but these expectations weren’t always met each week. As we will talk about in the development section, much of the coding is front loaded, so once we have one or two pages written, the others got easier to add and integrate into our application.

|  |  |
| --- | --- |
| **Week 1** | Ionic Research, Evaluation of Platforms |
| **Week 2** | Ionic Research, Evaluation of Platforms |
| **Week 3** | Ionic Research, Initial Drafts |
| **Week 4** | Ionic Research, Rough Wireframing |
| **Week 5** | Boilerplate Application, Final Wireframes |
| **Week 6** | Boilerplate Application, Logo/Color Scheme |
| **Week 7** | Finalize Boilerplate, Survey Development, ***Presentation 1*** |
| **Week 8** | Login/Signup, IRB Approval |
| **Week 9** | ***Spring Break*** |
| **Week 10** | Add Product, Integrate Firebase DB, Survey Deployment |
| **Week 11** | Search Products, Native Features, Survey Filtering |
| **Week 12** | Attach Products to User, UI/Design, ***Presentations (LAS)*** |
| **Week 13** | *Any additional adjustments:* ***Liberal Arts Symposium*** |
| **Week 14-16** | Chat, UI/Design, User Guide, ***Final Documentation*** |

# 4. Identity

## 4.1 The Problem

When we first started looking for ideas for our research projects, we had no idea what direction we wanted to go in. Upon discussing with other students and through our own experiences, we found that each semester, we were spending far too much on college textbooks. For example, within the Information Technology and Computer Science department, we rarely needed these books, but they were still required to be purchased as a part of the course. Many of the resources stay the same from year to year, so why would we as students continue to overspend on them? Students could sell their books to the bookstore or to a third-party seller for a fraction of the cost, only to have their friends buy the same book three years later. Through this realization, we found out that there was no platform catered to college students that addressed this need. After discussing the idea of Bizi Marketplace to other students, they agreed that it could potentially solve many of the needs of students on campus. Thus, the idea of Bizi was born.

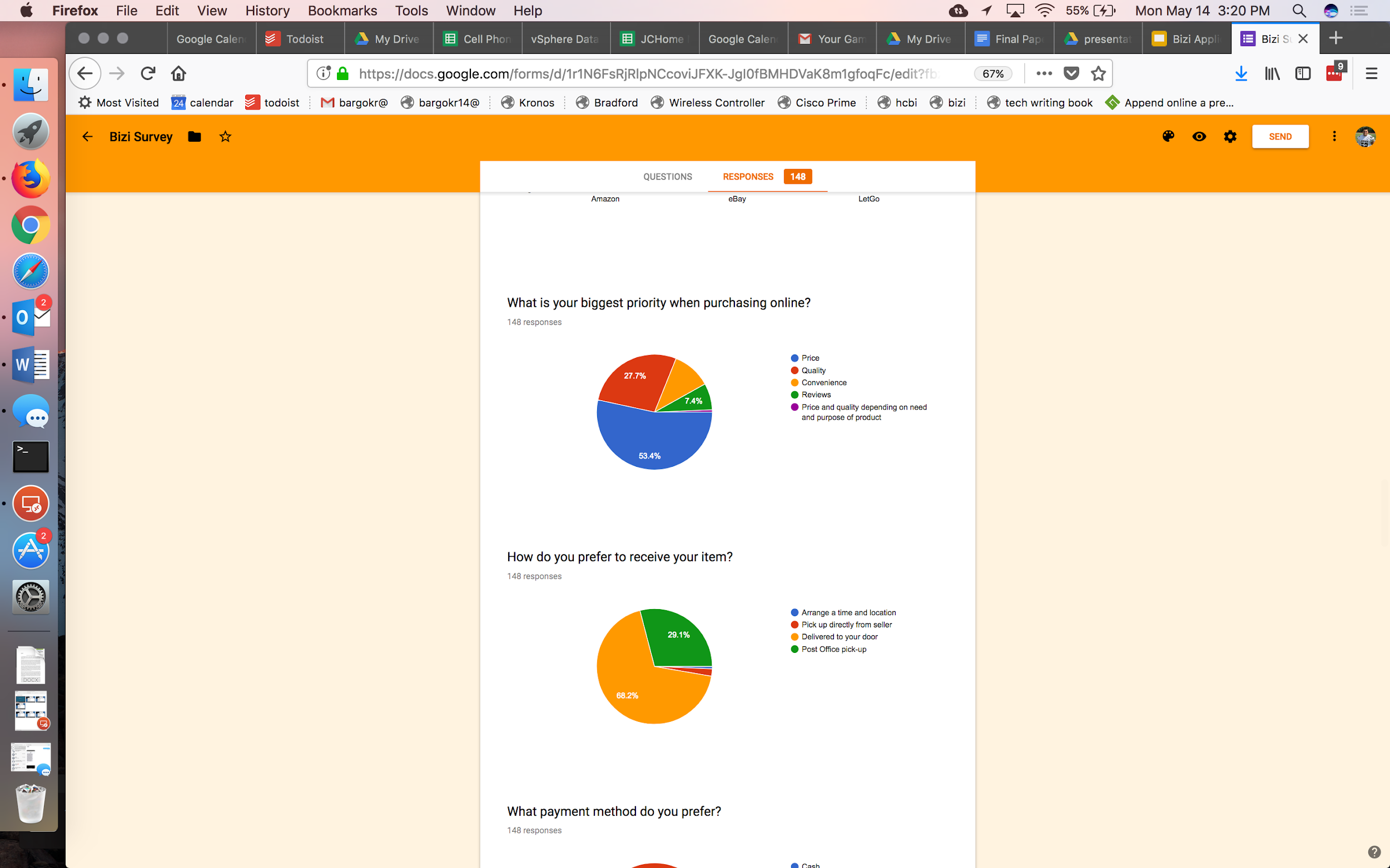
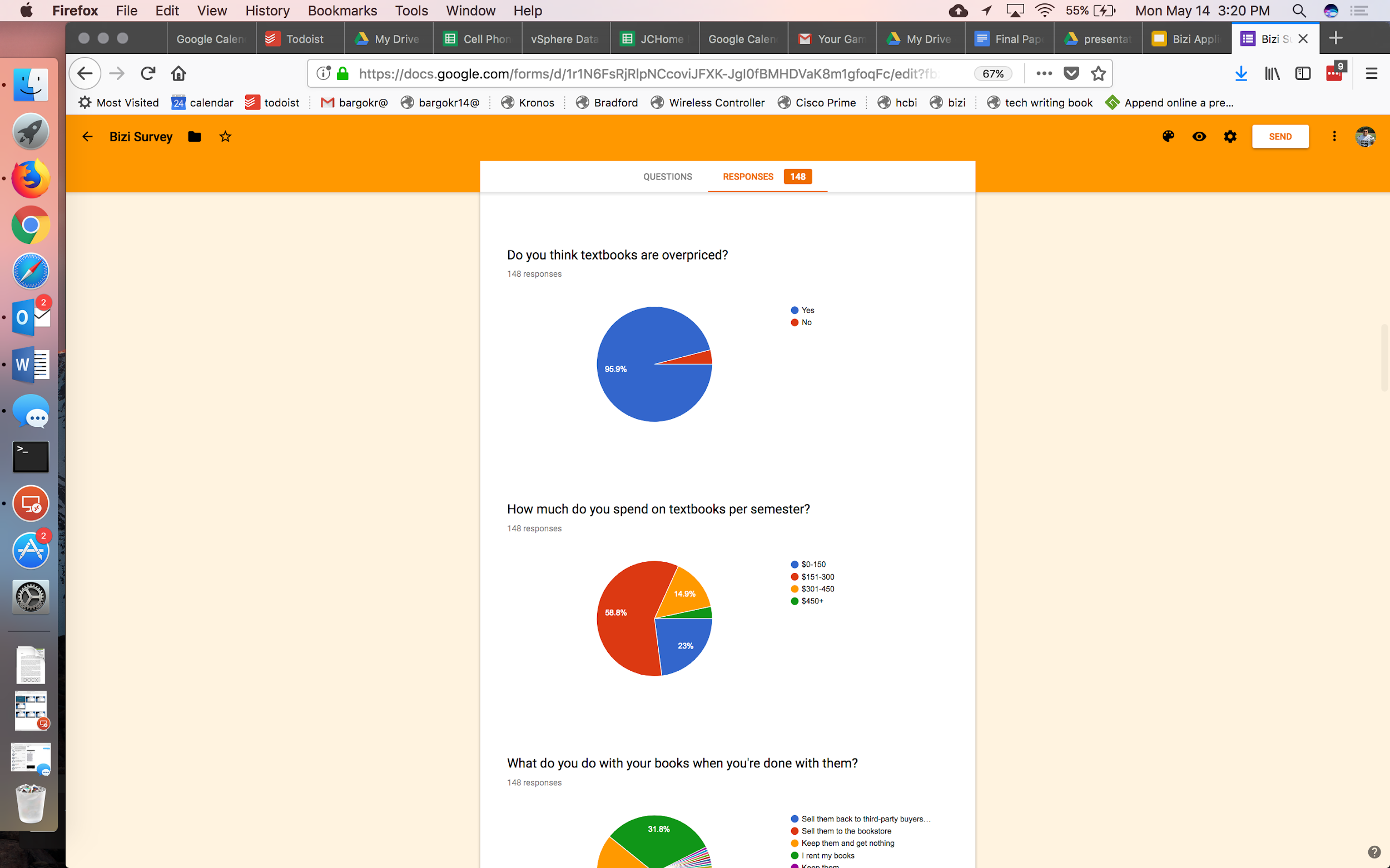
## 4.2 The Solution

Bizi Marketplace will be created as an application with a multitude of different functions. It will allow a user to browse products, like/favorite a product, offer or purchase a product, sell a product, follow and unfollow other users, and chat to meet up with another student. These are the features that we plan on having to start, but as we grow, this may increase. The goal is for Bizi to go global and expand to other campuses once we get our proof of concept at Juniata. As we deploy the application and start to get feedback, we will use this to better tailor it to the students that are using the application.

## 4.3 Google Survey

We created a Google Survey to identify the target market, ask users what they are looking for, how much they are willing to spend each semester on textbooks (our initial primary product), and other questions relating to advertising with organizations and how frequently they visit them. After a two week deliberation with the Insitutional Review Board at Juniata College and minor changes, the survey was deployed and taken by 148 students on campus. Incentives were given as a reward for participation.

The majority of these questions gave us some great insight on what we would need to produce with our application. Some examples include learning that 95% of students believe that textbooks are too expensive, 77% of students spend more than $150 each semester on books, and of the 70% of students that buy their books, 40% choose to keep them and get nothing back in return. We also learned that both Sheetz and Walmart were the most frequented corporate businesses, while Standing Stone and Downtown OIP did the same as local businesses. We will use this data to figure out what local businesses we should target when we start prospecting for sales. This data is still being filtered, but should be used for data-driven decisions this summer.



# 

# 5. Design

## 5.1 Design Evaluation

As part of the initial effort to create this application, we researched the mobile applications of other companies that had similar marketplace structure. The ones we primarily focused on were LetGo, OfferUp, and Mercari.

LetGo and OfferUp both operated very closely to what we are striving for. They don’t provide in-app purchasing, so everything is local pickup and exchange. We believe this to be the best way for college students to buy and sell products, as most campuses are compact and also to avoid shipping prices and dealing with the liability in that area. LetGo has been advertising heavily in the media, alluding that their market is middle-aged individuals looking to offload various electronics and other objects. OfferUp is also catered to this group, but they have a slightly different design and have more real-time posting mechanics. Instead of asking you to fill in most of the information, their app attempts to find what you are selling and asks if it’s correctly described and placed into the correct category.

Mercari is a completely different application when it comes to function and purpose. We chose this because it is catered to college students, has a nice design, and Kyle has been using it for quite some time. They recently rebranded to what we believe is a less-appealing design, but the old design offered some ideas to us in terms of how to integrate the color scheme and logo, how to organize the products, querying within the app, and some profile features that add another layer of personalization to each profile.

## 5.2 Design Discovery

Through screenshotting and evaluating each of these platforms, we were able to come up with a general design and put it on paper. We decided that once signed up through the login/signup screen and on-boarded into the main view of the application, you would have the choice between five different tabs. They are as follows:

**Home -** displays the products in a grid-like fashion where you can scroll

through all products and click into each for a description and more

information.

**Search -** search based on category and keyword.

**Add Product -** create a product listing with picture, title, description,

ratings, and tags.

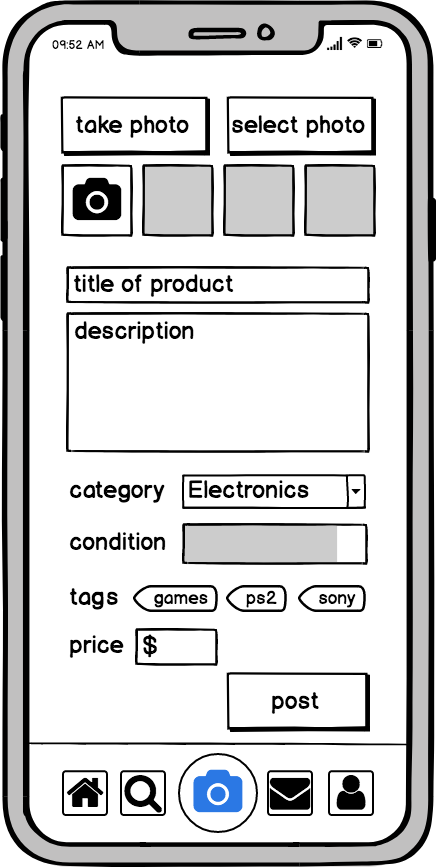
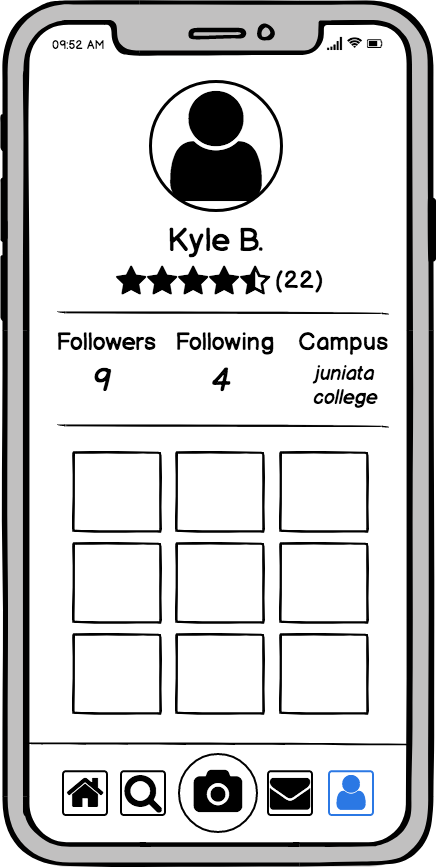
**Notifications -** access notifications and messages from both buyers and

sellers.

**Profile -** see all of your products and information about yourself.

These served as the foundation of our application and the development of it. We created paper drafts highlighting this functionality to then create wireframes to use for the design and development.

## 5.3 Wireframing



# 

# 6. DevOps

## 6.1 Cross-Platform Frameworks

Developing a cross-platform application gives the developer plenty of options when deciding on how they will be building their app. There are two main approaches: building a native app for both platforms and a website, or building for both simultaneously using some sort of framework. Building the app natively means that you will build the iOS version in Objective-C and Swift using Xcode and you will be building the android version in Java using Android Studio. Choosing to utilize some sort of framework to build for both platforms not only decreases complexity, it also saves time. The two most popular frameworks for cross-platform development are the Ionic framework and React Native. Both of these frameworks are based on other Javascript frameworks, Ionic using Angular and React Native using React.

The two frameworks are fundamentally different but provide the same result. An Ionic app is a website being run as a website inside of an app container. So, instead of building a true native application, you are building a responsive website that runs and specifically styles certain elements based on whether the device is running Android or iOS (González). Ionic apps are built using Angular meaning that if you are already familiar with Angular then getting up to speed will take no time at all. Apps built with Ionic follow a very similar model to how a normal website is made by following the standard of building in HTML, CSS, and JavaScript, while utilizing SASS and Typescript.

The React Native framework works with basically the same model of HTML, CSS, and JavaScript, but what happens behind the scenes is where Ionic and React Native start to show their differences. React Native compiles all of the code into a native app, meaning that you no longer have a responsive website in an app sized container, you now have a true native application. React Native builds for both platforms as you go in a very similar fashion as Ionic where there are specific tags for these frameworks that mean certain things and work for both automatically. A good example of this is with how React Native uses its ‘<StatusBar>’ tag; you do not need to build a status bar from scratch, both languages just know when the code is compiled to grab a true status bar element from their native platforms (González, 2018).

Deciding on which way to develop lies in the user’s preference. We chose Ionic because of Bruno’s familiarity with both Angular and Ionic before we began working on the app and since we were going to make a website down the road to serve alongside the app, and Ionic gave us the most reusable code for that. While Ionic is a solid choice, building the apps natively or using React Native gives the developer and user the best experience from the application besides building both apps natively. Using either of those techniques gives the app faster run times and delivers a user experience that the typical users are used to. Ionic tends to be slower and have strange glitches every once in a while, but still worked well and best suited our skills and prior knowledge.

## 6.2 Integration

Choosing the Ionic framework allowed us to easily attach other complexities to our application with little or no hassle. The nature of the app we were building required us to not only store lots of textual data but images which are famously known to hog up storage. We tried to find not only the cheapest way to connect all of the services that we needed, but also the most effective way which would give us the most out of our app. Some of the outer complexities that needed to be utilized were Firebase, Cloudflare, Nodejs, and Cordova. Each of these respective services provided a unique service that made development and storage so much easier. All of these apps will be further explained later when discussing the frontend and backend of the application.

Our application working similar to that of a website enabled us to utilize some services, that are mostly focused towards websites, with little to no trouble. Since React Native focused on building native apps, it would have been slightly tougher to use some of the services that we used with it which would have really halted development and forced us to begin a constant cycle of developing workarounds to utilize every external app that we wanted or found necessary.

## 6.3 Tools

Our project depended heavily on external services and products to handle basic functions and store the majority of our data. The tools listed below will be further explained and we will provide you with the reasoning for each choice.

*Visual Studio Code*

Our choice to use Visual Studio code was quite a simple one as VS code has really separated itself from others when it comes to any sort of Javascript development. VS Code had many angular, ionic, and typescript plugins that allowed us to grab snippets of code quickly and spend less time trying to think of a tag name or a method call, but rather they were all easily accessible and provided to us.

*Github/Github Desktop*

Choosing Github as a code repository was another obvious choice since Github is the most popular code repository there is. Bruno had a Github premium account so we were even able to keep our repo private so that only the two of us were able to see commits and what we were pushing. Github Desktop is also their client GUI that really simplifies the process of pushing and pulling files and gives you plenty of options for what to do with your changes. Also, we were sure to start using Github from the start because having some sort of backup beyond just our two machines is always the safe play and made sure that nothing too terrible would happen.

*Firebase*

Firebase is a product created by Google that is meant to serve as a backend to websites and mobile apps. We utilized Firebase for plenty of practical uses in our app that would have been a pain to handle manually. Firebase provided us with authentication services, database storage, free hosting, email verification, file storage, and others which really helped us find a place that is free currently for keeping tons of our most important data. Not only was Firebase possibly the best choice all around it also offered a very simple integration process into an Ionic app, which we obviously decided to make the most of.

*Cloudflare*

Cloudflare is a storage service that we have decided to use as our CDN (Content Delivery Network) to store product and profile pictures. Making use of a CDN meant that we would be able to save lots of room in our database for more profile storage and product storage. By keeping all of our images in Cloudflare, we are able to only store an image as a URL and then once the time has come to access that data, we pull back the record and then pull the image from its hosting site name.

*Nodejs*

Nodejs served as the backbone to this entire application. Node allowed us to install whatever packages were necessary for running our Ionic app and since all of the deploying and file maintenance are done from the command line, having Node made package management and file creation much simpler and saved us plenty of clerical coding errors.

*Cordova*

Cordova is an apache plugin that allows you to write HTML, CSS, and Javascript code for multiple platforms. Ionic utilizes Cordova to build its applications, but there are other options that provide similar services. Xamarin, which was acquired by Microsoft in 2016, allows you to do the same thing as Ionic and Cordova, but instead of using web languages, you use C# and XML while utilizing Visual Studio and Xcode as IDEs (Clarity, 2017). When it came to choosing one of these ‘compilers’ for lack of a better term, we did not really have a ton of options since Ionic comes shipped with Cordova.

*Ionic DevApp*

The team at Ionic (along with building the actual framework for development) chose to build an app that allows you to test your Ionic apps on a native device to make sure that everything is working properly. Though this same effect can be achieved using an emulator, it is always nice to see what the app is going to look like on your phone in your hands. With certain features like accessing the camera, you are unable to test on an emulator or in a web browser, so without the use of this app, trying to test that part of our application would have been quite difficult. Along with allowing for native feature testing, using the app also allowed us to test how the app would react with more than 1 user using it at the same time. Since our app utilizes a real-time database of products, it was important that once someone deletes, adds, or edits a product, you can see the results as soon as the page loads again.

*Google Chrome Dev Tools*

Google Chrome’s Developer Tools made debugging and testing our app very simple in the browser. Chrome offers lots of features like a debugger, responsive design, a console to log anything that we wanted to print to see what values variables held at a certain point, and many others. Since you ‘served’ the app to your localhost you were able to see changes on each save that you did of the code without having to reserve and recompile, which was a huge time-saver when it came to small tweaks to the code.

*Macbook Pros*

We decided to list Macbook Pros as a tool since you are unable to actually deploy an iOS app without a device running OSX that has Xcode tools installed. You are able to write the code and even compile and view the app on other os’, but you cannot take the final steps without an Apple product.

# 7. Development

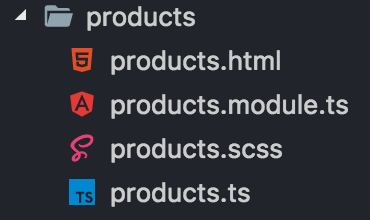
## 

## 7.1 Frontend

*Angular*

Since Ionic utilizes AngularJS, we were able to make use of some of the features that make Angular such a great language for frontend development. The way that Angular works is you are given 3 files for each page that you build: an HTML file, a SCSS/CSS file, and a Typescript file. These files all serve different purposes, the HTML file contains all of the elements that are displayed on each screen, the CSS file contains all of the styles for that individual page, and the Typescript file is where the database calls, button click methods, and other functionality for that file are written.

Each page had its own folder of files:



The products.module.ts file is used more in Angular than Ionic since it is used to set lifecycle hooks and other configuration info.

*Ionic Elements*

Ionic comes loaded with plenty of their own tags and elements for building a nice UI rather quickly. Rather than wasting time styling our own elements and creating classes, we grabbed the ionic elements that we thought would fit best together and plugged in our content to see how it would turn out. We spent a great deal of time deciding on which elements would best fit our app and the design that we had envisioned. The navigation bar, the tabs for each view, the buttons, the text fields, and so on, are all elements that ionic provided for us to use.

An example of this that we used constantly was the ion-button element, shown below and the code needed to use it is also shown:



<button ion-button (click)="onEditProfile()">Edit Info</button>

Though we gave this button our own colors and styles, this is still pretty close to what ionic provides out of the box, and using something like this quite simple:

*Ionic/Angular Features*

Ionic and Angular provide many different features that we used for just about every part of our app. The three main features we used were data binding, the navigation controller, and using navigation parameters. There were other classes and features that we used here and there, but these three served as the building blocks to our frontend. Data binding is how you can display elements from your Javascript in an HTML page. For example, we had to find a way to display all of the products in our database on the products page and that is not something that can be hardcoded in. For this, we used things like ngFor loops, which loops through an array and displays each element on the page. So to do this, we would build something like a div container and include an H1 element in each div. Instead of actually typing text for the H1, you would use the Angular syntax of {{ element.title }} to display the title of every object in the array.

<ion-content padding>

<div \*ngIf="products === [] || products === null">

<p>Looks like you are the first one here!</p>

</div>

<div>

<ion-grid>

<ion-row>

<ion-col col-6 col-md-4 col-xl-3 \*ngFor="let product of products; let i = index">

<div class="image-container">

<img src="{{product.image}}" (click)="onProductClick(product, i)">

</div>

</ion-col>

</ion-row>

</ion-grid>

</div>

</ion-content>

This ion-col example shows how you would display all of the current products. From the backend, you call the pullProducts() method which populates the products variable which is what you loop through using an ngFor in the HTML file.

pullProducts() {

this.as.getActiveUser().getToken()

.then((token: string) => {

this.ps.fetchProducts(token)

.subscribe((prods: Product[]) => {

if (prods === null) this.products = [];

else {

this.products = Object.keys(prods).map(i => prods[i]);

this.products = this.products.filter(function(x){

return (x !== (undefined || null ));

});

}

},

error => {

console.log(error);

});

})

}

As soon as you enter the products page, this code executes in the background which calls a method from our product service that pulls down the current list of products. This info is stored in the ‘products’ variable, which is an array of Product objects. From here, you are able to reference all of its properties like image and name by simply binding the data, using the ‘{{ }}’ method.

Along with looping through elements and data binding, we also needed way to write conditional statements to handle an absence of items or data. To do this, we used ngIf statements for checking whether a form was properly filled out, or an object had an empty field, or an entire array was empty. Based on the result of the conditional, you can select what you want to display. For example, if the products page is empty, you display, “there are no products yet”, but if there are products, you display the products.

<div \*ngIf="products === [] || products === null">

On our products page, we used an ngIf statement to ensure that we were in fact pulling back products from our database. If there were no products, we chose to show some sort of message instead of just displaying a blank screen. So when the page is being rendered, we check to see if the products array is populated with items, if it is not, then we print a nice statement that says that person is the first one there. We also used ngIf on any page where a form is involved because it works as a checkpoint to validate all form information. If the form is missing values, you can disable the submit button and make sure that all required items are filled out before they are allowed to try and submit anything.

<button ion-button block type="submit" name="new\_product" [disabled]="!f.valid || imageUrl == ''">Add Product</button>

The code above is what is in our sell page that is the submit button when entering a new product. If there is no image or the form is not valid, you are unable to click the submit button.

The next feature that we used rather heavily was the navigation controller. The navigation of an Ionic app is set up as a stack, pages are pushed on and popped off of the stack giving a feeling as if you are really switching pages within the app but really you are just putting one page on top of another. This allows us to move around the app by writing a method called something like goToHomePage() that includes a line that says navCtrl.popToRoot(), which will take you from your current page to the root page, which is the home page. Along with moving around between pages, it also gives you the ability to nicely just push on top of another when it logically makes sense. A good example of this shows up on our products page, if you click on an individual product, you do not go to a whole new page that just shows the product details, you simply push the details page onto the stack and it looks as though you are looking at a subview of the products page instead of a brand new page, which is what we wanted. By pushing pages onto the stack, Ionic then creates a back button in the navigation bar that lets you easily pop off the current page and return to the previous page. This allowed us to let the user easily move around the app and create a feeling of just flowing between pages instead of abrupt page breaks and loading screens.

onProductClick(p: Product, i: number) {

this.navCtrl.push(ProductPage, {product: p, index: i});

}

Using the NavController allowed us to easily move throughout the app, using it on the products page allowed a product click to quickly take us to another screen while passing through arguments to be pulled down in the NavParams, which I will talk about next.

Along with the NavController, a lot of what we did involved passing objects and arrays through navigation parameters. So along with pushing a new page onto the stack, we can also pass certain data with it. For example, when we are moving from the products page to the individual product page, you pass the individual product object through the navigation parameters and then on the individual product page store that object and then display it for the user. By being able to easily pass this data between pages we are able to avoid making lots of database calls and showing loading screens.

ngOnInit() {

this.product = this.navParams.get('product');

this.index = this.navParams.get('index');

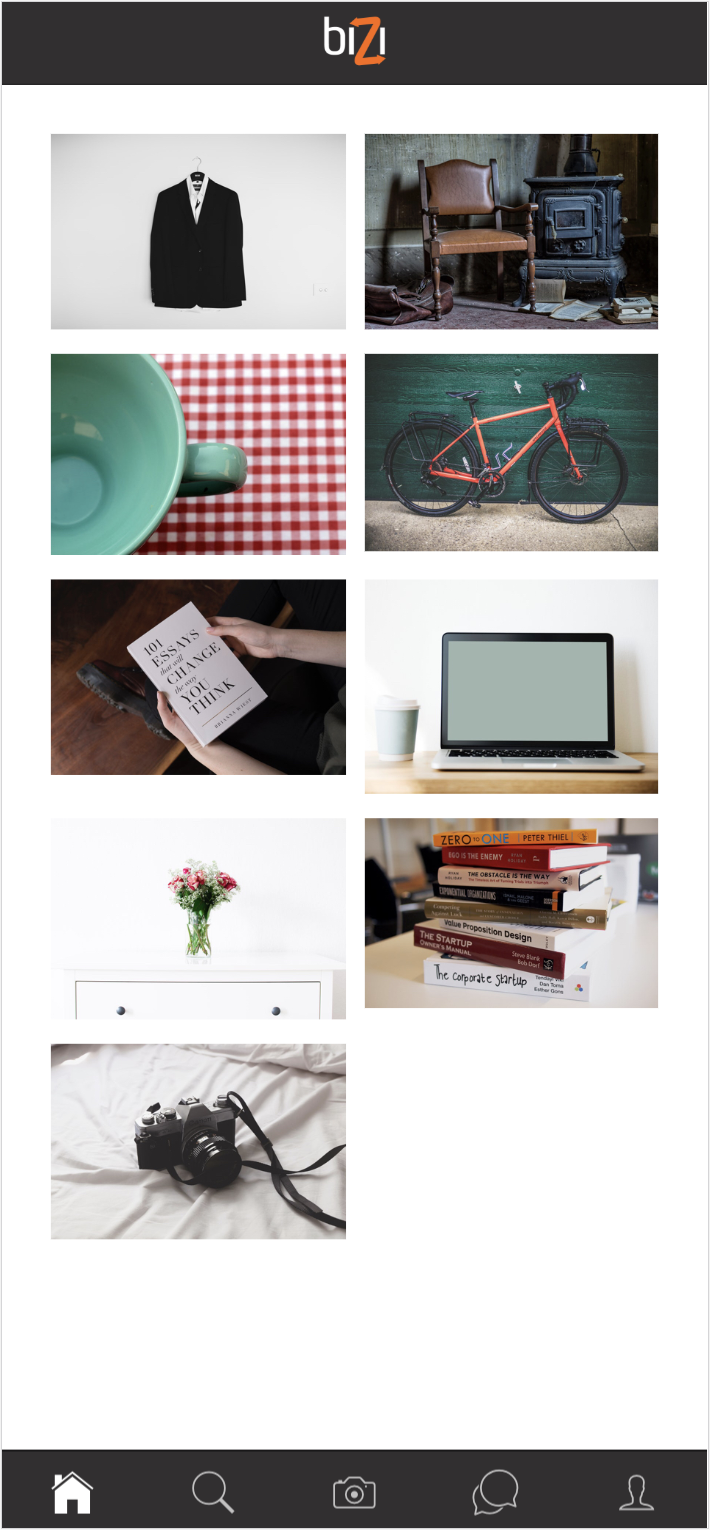
}

This code allows us to pull whatever information was passed through to the page as parameters. This method runs each time that page is initialized and always sets these values so that you are seeing the right product and information.

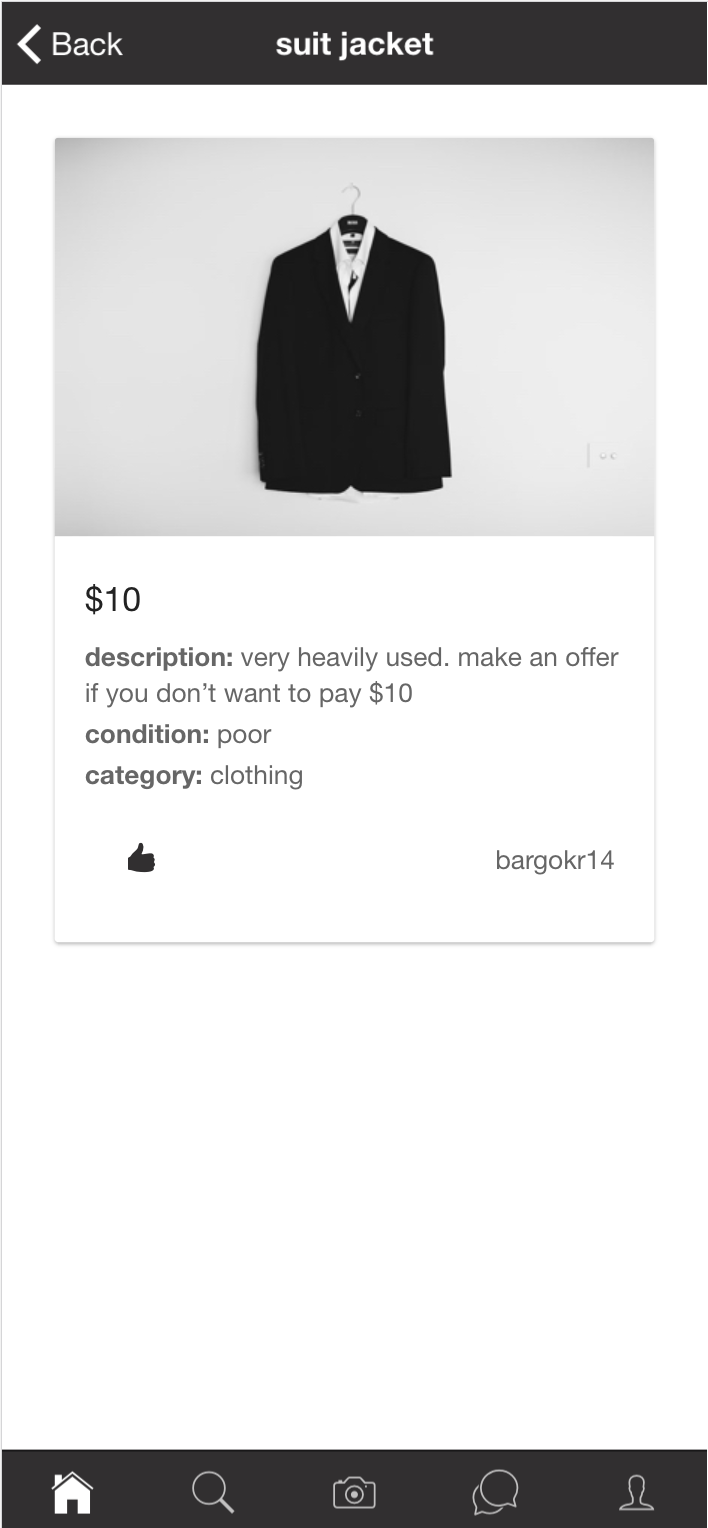
*User Interface*

Our app actually began with a lot of backend processing, authentication, and database configuration and testing, meaning that we really did not begin focusing on how our app looked until we had a fully functioning application. All the time that we spent wireframing and sketching out different views made building the UI quite simple. The majority of putting together the user experience was spent testing different ionic elements. We tried our best to align what we had wireframed with certain Ionic elements and for the most part, we got it pretty close. There were plenty of places where we wrote our own custom styles and tailored the app to something that was unique and did not just look like an Ionic stock application. We were able to easily manipulate the style of the whole app by just changing the app theme colors, which sounds super simple, but made a huge impact when looking at the uniqueness of the app. The stock primary color of these applications is an off-white, but by making that black and changing the supporting colors to orange and other colors that matched the Bizi logo and color scheme we quickly had an app of our own that looked nothing like the starting point.

Since most of our wireframing ideas had a similar looking Ionic element, we were not left with too many other choices when deciding how everything was going to look, but there were a few on-the-fly decisions we made that ended up working out nicely for us. Using the ion-card element for each product was the plan from the beginning before we even saw what an ion-card looked like, since we knew we wanted it to have the block structure of items. Deciding on the rest of the content holders and text boxes was really a trial and error process until we found what we thought looked best. We went pretty simple with just ion-items most places and added a few custom styles to spice up the look while keeping it simple enough that it was not too much. Most views have their own unique look like products, profile, and search, but you can definitely see similarities between the signup screen and the new product screen, which is what we wanted with similar views. The products display page is a our drop screen when opening the app (assuming you are already logged in) and because of that, we spent the most time working on giving that a modern, simplistic look.



The page that is the most important for our users is the products page. This is where they are seeing the products that they are trying to buy and if it does not have a clean design, they are going to quickly become uninterested.

We worked rather hard to develop what we thought was a clean and modern design that would attract the user and give them a way to quickly browse items to find whatever it is they were looking for.

Once you click on the item, it is important that we gave the user all of the information that they could need. Provided is the price, the description, the condition, photos, the category, and the poster.

*User Experience*

As any developer would say, we think that the UX of our app is pretty great and intuitive, but we realize this is not really for us to decide. We have not really had the finished product deployed anywhere where we could get some real testers and get feedback, but we are working towards those stages. Since we designed it, we know what buttons to click and where to go, but for someone unfamiliar, we want to make sure we are designing something that explains itself to the user. Our main design goal was to create something that is simple to use and not give the user millions of options to clutter the app, we just want the core functionality and to be to the point.

Once the app is released we will obviously be consulting App Store reviews and asking for honest opinions about the app and really try and take into consideration what people want and see how we can make changes to enrich their experience.

## 7.2 Backend

*Firebase*

The backend and core of our application lies within Firebase, which is a product by Google. Firebase, as previously explained in the DevOps section, is a service which provides developers with free databases, file storage, authentication handling, hosting, and many other options.

The most helpful feature Firebase has offered us so far is the user authentication. Firebase has methods that you can call to signup a user, signin a user, delete users, and edit user info which saves us plenty of work since we no longer need to write any of that logic. Firebase keeps a table that the project owners can see of everyone who has signed up for the app and everyone who has ever logged in. Although we wrote additional signup logic to check that the user’s email is in fact a .edu email and that it is an active email, Firebase has handled the majority of this for us. Firebase also allows us to verify a user’s email account on signup which is one of the main separators of our app from the rest of the market. We want to make sure that if you are using our app you have a valid .edu email address that you still are able to access and use, verifying that you are in fact a college student.

Firebase databases are NoSQL databases meaning that they are not a relational database and not structured like a typical database. Instead of an SQL database, we basically have a large JSON file that includes all of our database info. This may sound like a bit of an unorthodox way to go, but it provides us with options to store data in quirky ways. There are a few examples that jumped out to us when deciding on a backend as to why we would go this NoSQL route. The first thought was on how we would be keeping track if a school has branch campuses. Yes this sounds like a simple thing that we could have just added another lookup in the table for, but it is much easier to give the school object an attribute of ‘branches’ and either store false if it has no branch schools, or store an array of objects of branch school data. Another way the NoSQL route has made things easier is storing information and images for a product. Instead of having 3 image fields for a product and leaving some unfilled and having to check that later after we run a query, we simply store an array of image URLs and then use an ngFor to access all of them at once. The same process is done for our messaging between users. Instead of storing messages in a table, we are always just going to have an array of messages between two users and continuously append the next message to the current array and display that to the users. Storing this as an array saves time and means we have no complexity to worry about, we simply just store the array in a .JSON file that contains only messages between the two users.

Configuring Firebase in our application was also a super simple process. The only thing required of us was to add the API key from Firebase that is generated automatically from the project we created to our components.ts file for our project. So adding Firebase to our project required us to add one line of code, which is very quick and easy.

To configure our app with the Firebase API for our project, we simply just added this to our app.component.ts file which handles all setup for the app (these values are supplied to you by Firebase):

fb.initializeApp({

apiKey: "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",

authDomain: "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",

databaseURL: "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",

projectId: "\*\*\*\*\*\*\*\*\*\*\*\*\*\*",

storageBucket: "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",

messagingSenderId: "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

});

Firebase also, if we choose, gives us the option of file storage which is where we have strongly been considering storing our images. Though we currently plan on using Cloudflare as our CDN, we have been thinking about using just Firebase to reduce the number of other technologies required to run this app. The file storage would work just like a CDN would, we store the image URL as an attribute of a product, then store the file in the storage, and then just pull back that URL and display.

Firebase provides free web hosting as well for each project that you create, meaning that once we decide to build the website to work alongside our app, we would be able to host that for free with the only caveat being that we keep the ‘.firebase’ string in our URL.

*Cloudflare*

Cloudflare is a free CDN that we are planning on using to store and host our images for the app. We know that trying to store these images in our database is going to waste all of the free space that Firebase gives us to store actual user and product info, so by outsourcing this to another free service, we are saving ourselves from upgrading our Firebase plan. From conducting research on the topic, Cloudflare is one of the most popular and widely-used CDNs for websites. Since Ionic apps are basically web apps put into a phone sized container, we are able to use a CDN built for websites which is easier than trying to find a CDN for native apps which could differ because of how you would actually display the images when called in a native app.

*Nodejs*

Nodejs is a Javascript framework that allows you to create a full backend for a website using only Javascript. We really did not make use of this ability of node, we strictly used it for package installation and adding classes/features to our app. Ionic has lengthy documentation and tons of auxiliary classes available for use, but must first be installed via node.

We used node in a variety of ways though, we ran all of our core commands from the Ionic Command Line Interface through node.

To serve our app to our local server:

ionic serve

To install any packages that we needed (installing the package globally):  
npm install -g PACKAGENAME

Serving our app to not only our local server but our phones:

ionic serve -c

Building our app for ios (same for android, just replace ios with android):

ionic cordova build ios

## 7.3 Functionality

*Writing Custom Methods*

The way that Ionic and Angular work is that you are able to write custom Javascript methods and call them as though it were just a function that Ionic included which allowed us to customly handle all of the complexities of the application. We wrote plenty of methods to handle email checking, searching products, database calls and modifications, and even checking to make sure the current user is authenticated. The core of our app’s functionality did come from these custom methods that we have written.

One of our more important custom methods that we wrote is our pullProducts() method that reaches out to our Firebase database and pulls back a new list of products every time that the user enters the page.

pullProducts() {

this.as.getActiveUser().getToken()

.then((token: string) => {

this.ps.fetchProducts(token)

.subscribe((prods: Product[]) => {

if (prods === null) this.products = [];

else {

this.products = Object.keys(prods).map(i => prods[i]);

this.products = this.products.filter(function(x){

return (x !== (undefined || null ));

});

}

},

error => {

console.log(error);

});

})

}

This is one of our more complex methods and it is really handled in only a few lines of code. This method checks to make sure the user is authenticated, then pulls everything from the products table, using the fetchProducts() method written in a different service class, then returns an array of products. One of the most important parts of pulling back this data is handling whether or not you are returning an empty array or not. \*NgFor loops and the data binding does not play nice with empty arrays or null objects, so a lot of pulling back data is writing try, catch statements to ensure that the data you are pulling is there.

Ionic allows you to call methods from an abstract class on any page within the app and on any sort of event. The most used event that we run other methods from is the ionViewCanEnter() event which means that the app is now about to enter whatever specific view and all of these things need to happen before the page loads. A good example of this is within our products page. Since this is the first page that loads when you open the app, Ionic does not immediately realize that you are authenticated since the method call to check authentication takes a few seconds to process. To handle this quirky error, we added a loading spinner to the ionViewCanEnter() the requires that the loader take around half a second to make sure that the user is authenticated first. This is very important because all Firebase database calls require that an authenticated user is making them, so with each database call we pass a unique key that is given to each user that lets Firebase know that they are in fact a user and are logged in. So by waiting that half of a second to establish that the user is authenticated, we are able to wait just long enough to successfully reach out to Firebase and pull back an array of products. Without this wait time programmatically set, the list of products is null. Once the user is authenticated, the products page will not longer show that waiting spinner, it simply knows you are authenticated and makes the database call.

We also have worked to implement a searching feature within the app to lookup products by name and as long as you type something close you are able to see the product. This is accomplished by pulling down the array of products and running a filter on the array for that certain string within the array elements and if there is a match, return an array of elements that contain that string.

Using Angular and Ionic, creating searching functionality really only takes up about 15 lines of code. On the backend, you need to pull from the database of products everytime that you enter the page, so after that is accomplished, you can write a search method in your typescript file like so:

filterItemsByName(ev: any) {

this.setItems();

let val = ev.target.value;

if (val && val.trim() !== '') {

this.items = this.items.filter(function(item) {

return item.title.toLowerCase().includes(val.toLowerCase());

});

}

}

This method is called on any change to the search textbox, so everytime a letter is entered a list of items that have those letters is returned and displayed using an \*ngFor loop like so:

<ion-content padding>

<ion-searchbar placeholder="Search by Name" showCancelButton color="danger" (ionInput)="filterItemsByName($event)"></ion-searchbar>

<ion-list>

<ion-item \*ngFor="let item of items; let i = index" (click)="onProductClick(item, i)">

{{ item.title }}

</ion-item>

</ion-list>

</ion-content>

All that is happening here is that there is an \*ngFor loop that controls how many ion-items are being displayed for the user, since this is pulling from the ‘items’ array which is populated in the typescript file, only the items returned from the filtering method are displayed. This piece above is from the search.html file.

Something quirky that we do that possibly needs fixed in the future is that we are not simply appending products or info to the database, which each update or push, we take the current array and overwrite the existing database. Since this is actually a JSON file, there is not really a clean way to just keep adding data to the .JSON file, so we run a full product pull right before processing, add the current product, and then overwrite the list. This really should not cause any issues down the road since you make the database call right before processing and adding the product, and this whole processing takes like milliseconds.

Luckily for most of the forms we did not need to write post-processing to handle bad input or required files not being filled out since the ngIf statements took care of that before it was even possible to signup.

# 

## 7.4 Future Development

*Messaging*

Our next step is adding the functionality to allow users to message other users. This is going to be something that is important because we need a way for users to interact on the app, express interest, and set up times to buy. As for actually building this, which I somewhat touched on earlier, the majority of the work is just going to be storing an array of Message objects. The Message object is going to include the message and the sender. We need to include both so we can place the user’s message on the correct side of the screen like any messaging app would work. We should also be able to store the objects in our database and create a new table called messages and then create .JSON files based on who the message is between.

*Social*

We are interested in adding some sort of social aspect to the application allowing users to favorite products and follow users to get notifications when they post something. Adding this ability to interact with people is something that is going to help users get the products that they want from who they want to get them from. If you know there is someone that has the same major as you and they often sell their books, you obviously want to be notified when that person puts their stuff up for sale.

*Notifications*

We are also going to be working on setting up a notification system so users can be notified when someone they follow has posted, someone has followed them, when somebody liked one of their products, or whenever they have received a message. Giving people a heads up when stuff like this is happening will let the user know that they have people who are interested in buying their stuff and will lead to selling more products.

*Ratings*

Along with following other users, we also want to implement a ratings system that will give other users an idea of how reliable and how well someone’s products worked. This will work very similar to Uber. We want a way for people to be able to judge their interactions with others and weed out people who are trying to scam others or sell bad products.

*Optimization*

Once we have a much larger body of products and users, there are changes that need to be made to really optimize the performance of the app and make sure there are never loading screens or performance drops because of something the user is looking for. This will most likely involve some sort of sorting algorithm to sift through products and users when a search is done to make sure that we are returning the correct results very quickly. For something like a search, it would take a noticeable amount of time to filter through an array of 1000 items, which is why we are going to need to work on how we are looking up products and displaying them to the user.

# 8. References

González, Roberto. “React Native vs Ionic vs Native: What's Best for My Mobile App?” Aerolab, Aerolab, 22 Feb. 2018, aerolab.co/blog/react-native-ionic-native/.

“Xamarin Vs. Titanium Vs. PhoneGap Vs. Cordova: A Comparison.” Clarity, Clarity Ventures, 2017, www.clarity-ventures.com/resources/xamarin/xamarin-vs-titanium-vs-phonegap-vs-cordova-a-comparison.