Endicia Proof of Concept: A Link Between Endicia and E-Commerce Buyers

A Major Qualifying Project Report

Submitted to

the Faculty of Worcester Polytechnic Institute

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science

By

Michael LoTurco

Jeffrey Sirocki

Sponsored by

Endicia

Approved by

Professor Mark Claypool, Advisor

March 4, 2017

Abstract

Endicia operates in the shipping software solution market space providing services to warehouses and e-commerce merchants. This project aims to branch out from Endicia’s shipper oriented solutions, and instead form a connection with the recipients of packages, the e-commerce buyers. For an effective connection to be formed, buyers must be given enticing reasons to connect, offered an easy method of registration, and provided valuable services. By conducting market research, establishing product requirements, and pursuing feature driven implementation, we have developed a buyer link prototype and laid the groundwork for Endicia to expand into the e-commerce buyer market. The prototype includes a mobile application, a web application, an API backend, and a database built with modern frameworks and technologies.

Acknowledgements

We would like to thank everyone who influenced the outcome and success of our project. Specifically, we would like to thank Amine Khechfe, our project liaison and CEO of Endicia, for providing us with beneficial insight, clear direction, and essential Android testing devices for our application prototypes. We would like to thank Varun Sood, our project manager, for his instrumental guidance and support throughout the entire project. His technical expertise greatly enhanced our end product. We would like to thank Professor Mark Claypool for organizing the Silicon Valley ’17 project site and his sensible advice which proved to be invaluable. Finally, we would like to thank Endicia and Worcester Polytechnic Institute for providing us with this opportunity.

Disclaimer

Table of Contents

Table of Figures

Table of Tables

1. Introduction

As the global population becomes more comfortable with the Internet as a marketplace, people are more frequently turning to online vendors to supply their needs. Vendors of all sizes are offering goods available through online purchase. Massive vendors like Amazon operate and control many of these sales, and provide services like direct delivery, package tracking, and order history. Smaller scale vendors do not have the same funds and infrastructure to support a distribution and warehousing network. Instead, small and medium size vendors look to outside services to provide mailing, delivery and package tracking.

Endicia provides these vendors with the software solutions to offer direct shipping and package tracking to their customers. Endicia's primary offering to online vendors is the ability to print shipping labels, but they continue to develop new services as vendor needs arise. Endicia works with vendors in an attempt to provide their customers with the best purchasing experience possible. One service customers have come to expect is an ability to track their packages as well as view their order histories. Endicia enables vendors to send individual tracking emails to customers, but there is no centralized order history for users to view that information. For each item purchased through a vendor working with Endicia, the customer receives an email with links to both an Endicia based package tracker and the USPS package tracker, which gathers data from various package stopping points across the country.

There is an opportunity for Endicia to expand this service and enter the market for providing services to e-commerce buyers. To enter this market, Endicia must first begin establishing relationships and building trust with buyers. With this in mind, our project goals are:

- To establish a prototype that can connect e-commerce buyers to Endicia
- To ensure the prototype can appeal to users across mobile and web platforms
- To explore new technologies that can expedite the development process of mobile applications for Endicia's developers
- To create a scalable architecture that can incorporate data from Endicia and its partners to provide valuable services to users

Our solution includes four parts: a cross platform mobile application offering cross-carrier package services, a web application that enables users to register easily and track packages directly from emailed links, a shared backend API, and a database that supports our applications. The backend opens the door for Endicia to work with its shipping partners to collect more in depth data about each package, like item details or seller information. By exploring new development technologies, creating a prototype ready for demonstration, and defining a system architecture for the application, we have laid the groundwork for Endicia to expand into the e-commerce buyer market.

The rest of this report is organized as follows: Chapter 2 describes technologies our project utilized, Chapter 3 provides our project goal and requirements, Chapter 4 shows our iterative design process, Chapter 5 describes the implementation of our mobile and web prototype, Chapter 6 covers our conclusions, and Chapter 7 suggests future steps that could be taken with this project.

2. Background

Prior to designing any application, the development team must conduct research on technologies and frameworks that may be useful, to determine the best stack with which to develop. This section covers the technologies explored for potential use throughout the project and gives an overview of the benefits and structure of each technology.

2.1. Angular / SPA

Angular is a structural JavaScript framework for dynamic web apps (Angular, n.d.). AngularJS was first released in October 2010 and soon became one of the most popular JavaScript front-end frameworks. In 2015, Google announced Angular 2 and made it available for developers to preview. Angular 2 is a complete rewrite of the framework and as such has different syntax as well as structure. This section will cover some of the core features of both AngularJS and Angular 2 and the differences between the two.

Both Angular JS and its successor Angular 2 make it easier to build Single Page Applications (SPA). SPA’s are web apps that load a single HTML page and dynamically update that page as the user interacts with the app. Figure 1 demonstrates the difference between calls made by a traditional website and calls made by a SPA. Note how a traditional web app pulls a new HTML page from the server every time a user navigates to a new section of the site. Conversely, SPAs use Asynchronous JavaScript and XML calls (AJAX) to obtain only the needed information to respond to each user action, thus avoiding full page refreshes. Whenever the user navigates to a new section of the site, or requires additional information, the client sends a request to the server. The server then responds with data, typically in the form of JavaScript Object Notation (JSON) or XML. Overall, the responsiveness and the user experience offered in an Angular SPA make it a great choice for web applications and hybrid applications.

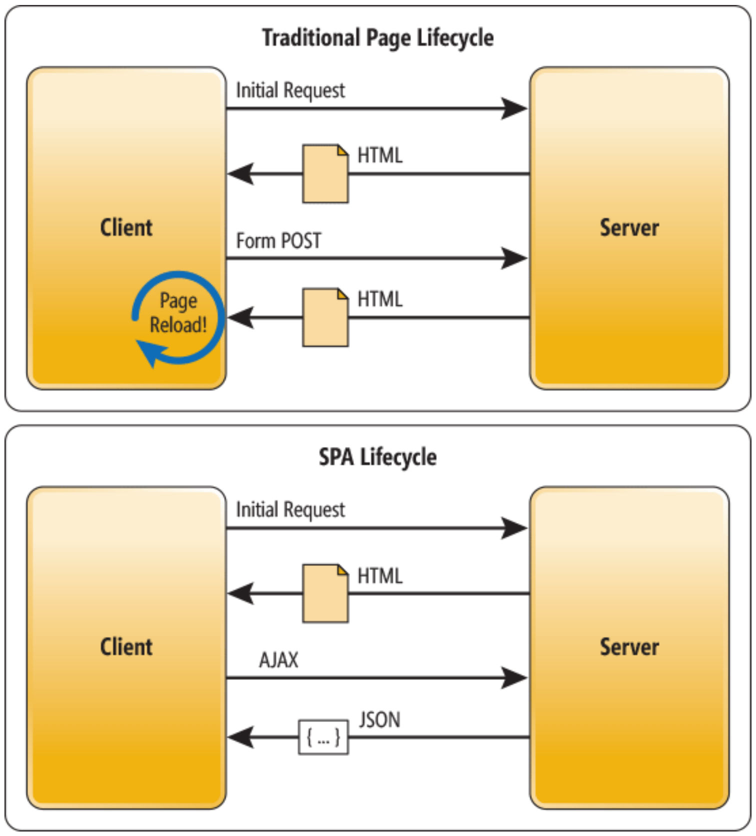


Figure 1 - Traditional vs. SPA Lifecycle (Covert, n.d.)

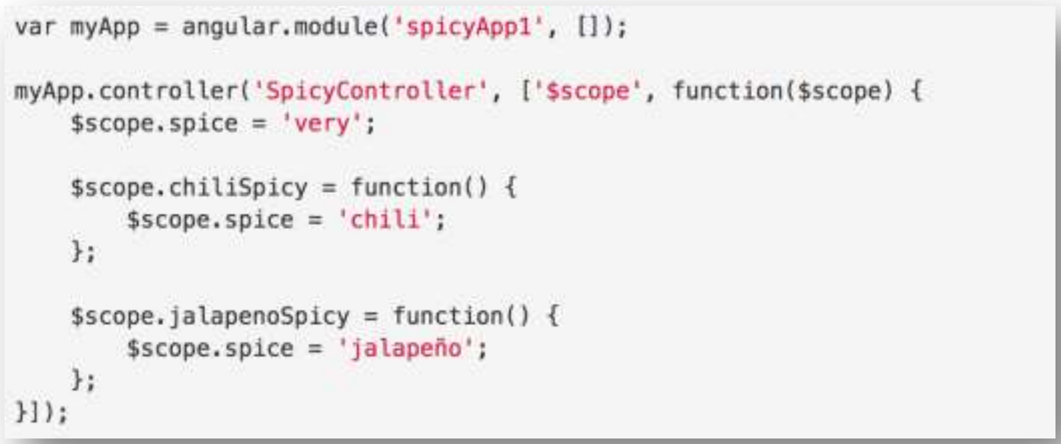
2.1.1. AngularJS

Developers build AngularJS applications with items called modules (Google Angular Team, n.d.). An Angular module is a logical segment of code that is often grouped based on functionality or tight coupling. Angular modules serve as containers for controllers, services, directives and other parts of the app. In a module, a view is what the user sees such as a HTML template. The controller handles business logic and modifies data in the model to update the view. Services contain code used throughout the application such as login authentication or any useful helper method. Modules gain access to other modules through dependency injection.

An AngularJS view defines components that the browser or web view presents to the user. A view is an HTML template composed of plain HTML and Angular directives. Angular directives are extended HTML elements such as *ngIf and *ngFor that offer additional flexibility for displaying information. If no Angular directive exists, developers may create custom directives. Furthermore, Angular views are dynamic through the framework's two-way binding in which Angular links data elements from the view with the data model. In a two-way binding, the user modifies data in the view which in turn changes the data in the model. Conversely, if the system alters data in the model, then two-way binding updates data in the view to reflect that change. Overall, the flexibility of Angular directives and dynamic two-way binding significantly reduce the time to make web pages in Angular.

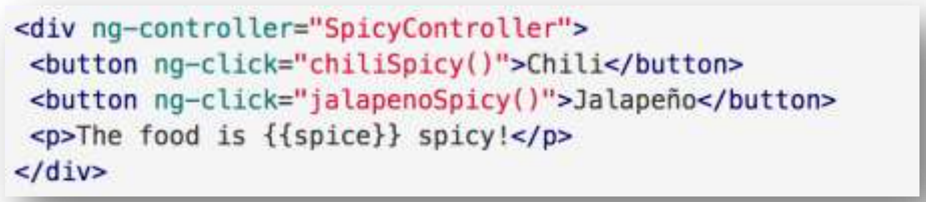
The second piece of an Angular module, the controllers, harbor the main logic of an angular application. Controllers are responsible for manipulating the model, which in turn changes the view. Controllers include functions which modify the data model and are typically triggered by events from the user interface or view.

Figure 2 shows a view, controller, and two-way binding. In this example, a user clicks the ‘chili’ button or the ‘jalapeno’ button, which changes the value of $scope.spice. Once the value of $scope.spice changes, the two-way binding causes the expression {{spice}} in Figure 3 to reevaluate. This event changes the value displayed in the view as shown in Figure 4.



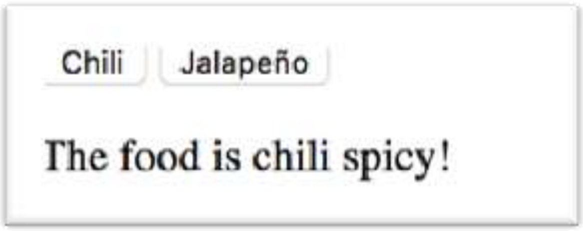
```
var myApp = angular.module('spicyApp1', []);

myApp.controller('SpicyController', ['$scope', function($scope) {
    $scope.spice = 'very';

    $scope.chiliSpicy = function() {
        $scope.spice = 'chili';
    };

    $scope.jalapenoSpicy = function() {
        $scope.spice = 'jalapeño';
    };
}]);
```

Figure 2 - Example controller with a model ‘spice’ of type string and two functions which change the value of spice (Google Angular Team, n.d.)



```
<div ng-controller="SpicyController">
 <button ng-click="chiliSpicy()">Chili</button>
 <button ng-click="jalapenoSpicy()">Jalapeño</button>
 <p>The food is {{spice}} spicy!</p>
</div>
```

Figure 3 - Example view using the defined controller and triggering the controller's chiliSpicy and jalapenoSpicy functions with buttons (Google Angular Team, n.d.)



The food is chili spicy!

Figure 4 - The displayed web page, after a user presses the Chili button. Pressing Jalapeno would modify the text to 'The food is Jalapeno spicy' (Google Angular Team, n.d.)

Angular's structure encourages developers to limit the purpose of controllers to only the modification of the data model as shown in Figure 2. If a controller begins to do more than strictly model modification the developer begins to clutter their code. To avert this, an Angular service is typically a better solution.

Angular services are reusable blocks of code that are generally removed from what a user may view. Controllers often inject such services to extend their functionality and reuse the logic already handled in the application. Services are singletons, so when multiple modules utilize a service only one copy of the service exists, created by the service factory. Services are ideal for implementing features needed through the application. Communication with a back-end via http, creating pop-ups or maintaining application-wide data are all tasks well-suited to being services.

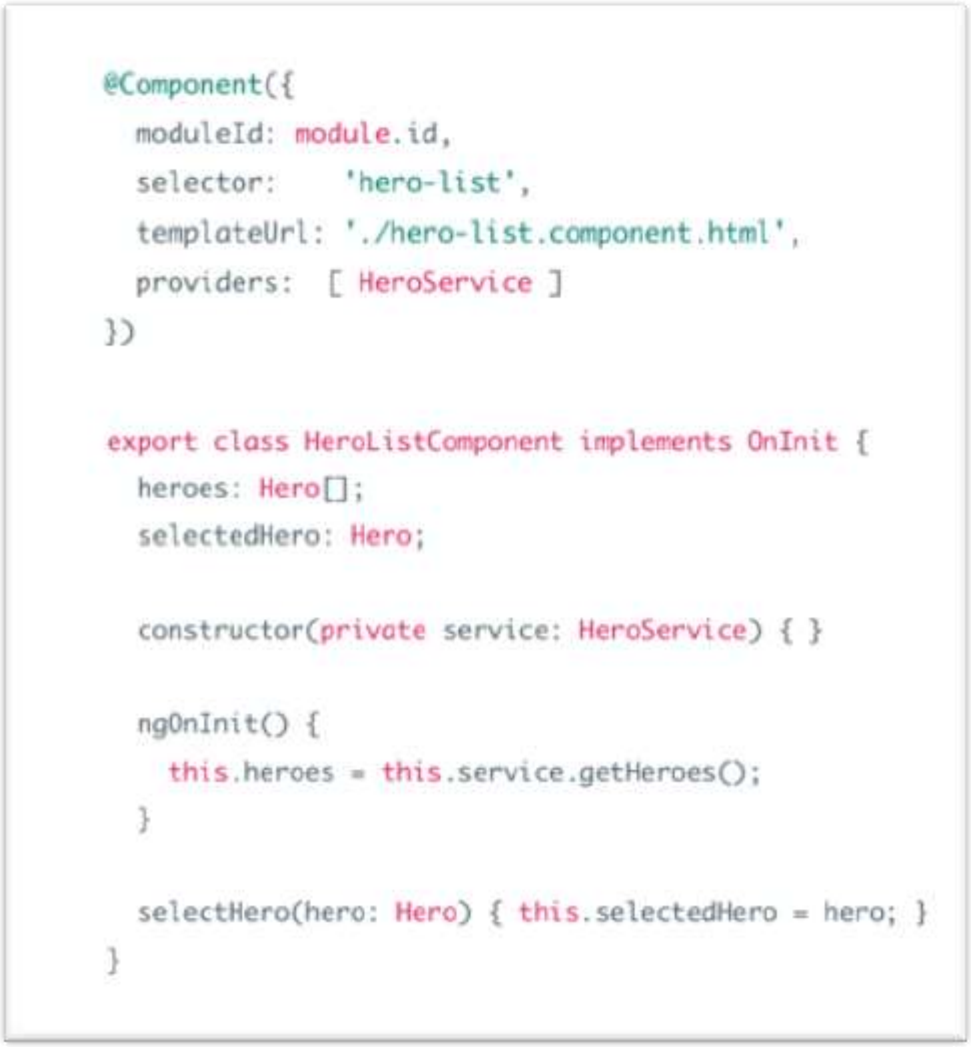
2.1.2. Angular 2

Angular 2 is a significant departure from AngularJS, though it is still a JavaScript framework and is best suited for SPA design. Angular 2 is widely written in Typescript, a Microsoft developed language which is a superset of JavaScript. While there are other languages compatible with Angular 2, the Angular team recommends Typescript for its significant tooling which expedites development.

Microsoft designed Typescript for ECMA Script 2015 (ES6), the standard specification for scripting languages which JavaScript follows. Most modern browsers only provide support for ES5 so in order for ES6 code to have reliable behavior the system transpile it down to ES5. Transpilation is the process of converting source code from one programming language into source code of another programming language. Tools like Babel transpile Typescript code into plain JavaScript, which meets ES5 standards. Typescript builds a number of features on top of vanilla JavaScript such as classes, strong typing, and generics (Savkin, 2016). This syntax is similar to object oriented languages like Java and reduces development time further.

The core structure that persists from AngularJS to Angular 2 is the module. Angular 2 modules are “a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities” (Google Angular Team, n.d). These modules declare the set of classes they utilize, import, and export as well as a set of providers, which create services. Each application must have a root module that includes a bootstrap property defining the main view. Additionally, the application uses the root module during startup, and is “bootstrapped” in a main.ts file to begin the application. The main view then hosts all other application views.

A major difference between Angular JS and Angular 2 is the use of components. In Angular 2, components, as shown in Figure 5, replace the AngularJS controller. Components hold application data that is relevant to what the view is currently displaying to the user in and include a link to an HTML template. Similar to a controller, components contain functions that manipulate data or methods user interaction trigger within the associated template. Angular creates components when pages request them and conversely destroys them as the user navigates to different pages.



```
@Component({
  moduleId: module.id,
  selector:    'hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})

export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

Figure 5 - An example component, with @Component decorator, components can implement (Google Angular Team, n.d.)

Angular 2 services are functionally equivalent to services in AngularJS, although the Angular team updated syntax to fit the object oriented nature of Angular 2. Developers inject services into components by creating a reference to the service in the component's constructor.

The Angular 2 framework includes many services. These services include those for sending and receiving HTTP requests or creating and displaying alerts.

Angular 2 templates are HTML pages which define what the user will see. Templates belong to a component and use data binding to access and display data from their host component. Templates use directives to further manipulate the user's view. Two common directives are *ngIf which displays a given HTML element if an expression evaluates to true, and *ngFor which iterates over a list to display HTML elements for each item in the collection. Views can nest templates, for example, an application which displays a list of products, and allows users to select a product to expand that product and see its details in the list. To accomplish this, the developer would nest the product detail template inside of the product list view. These features of templates and Angular 2, in general, increase code readability while aiding construction of complex and elegant user facing views.

2.2 Apache Cordova

Apache Cordova, formerly known as Phonegap, is an open-source mobile development framework introduced in 2011 (Apache Cordova, n.d.). The framework allows developers to use standard web technologies, HTML5, CSS3, and JavaScript, to easily develop hybrid apps for nearly every phone or tablet on the market. Cordova wraps an HTML, JavaScript app into a native container which uses the native platform's API bindings to access device capabilities. Developers can build Cordova applications for all mobile platforms from one code base (Apache

Cordova, n.d.). Apache Cordova is compatible with a diverse group of plugins, tools, JavaScript frameworks, and cloud services.

Apache Cordova technology makes hybrid apps possible. Figure 6 shows how Cordova provides plugins and a rendering engine for a web app. Cordova wraps the Web app in an HTML Rendering Engine which handles displaying the application by interfacing with the operating system. The rendering engine then communicates with Cordova plugins which provide access to native functionality by accessing the phone operating system as well. Cordova's architecture abstracts the work of interfacing with the device, allowing developers to make applications supported on multiple platforms with rich access to device capabilities.

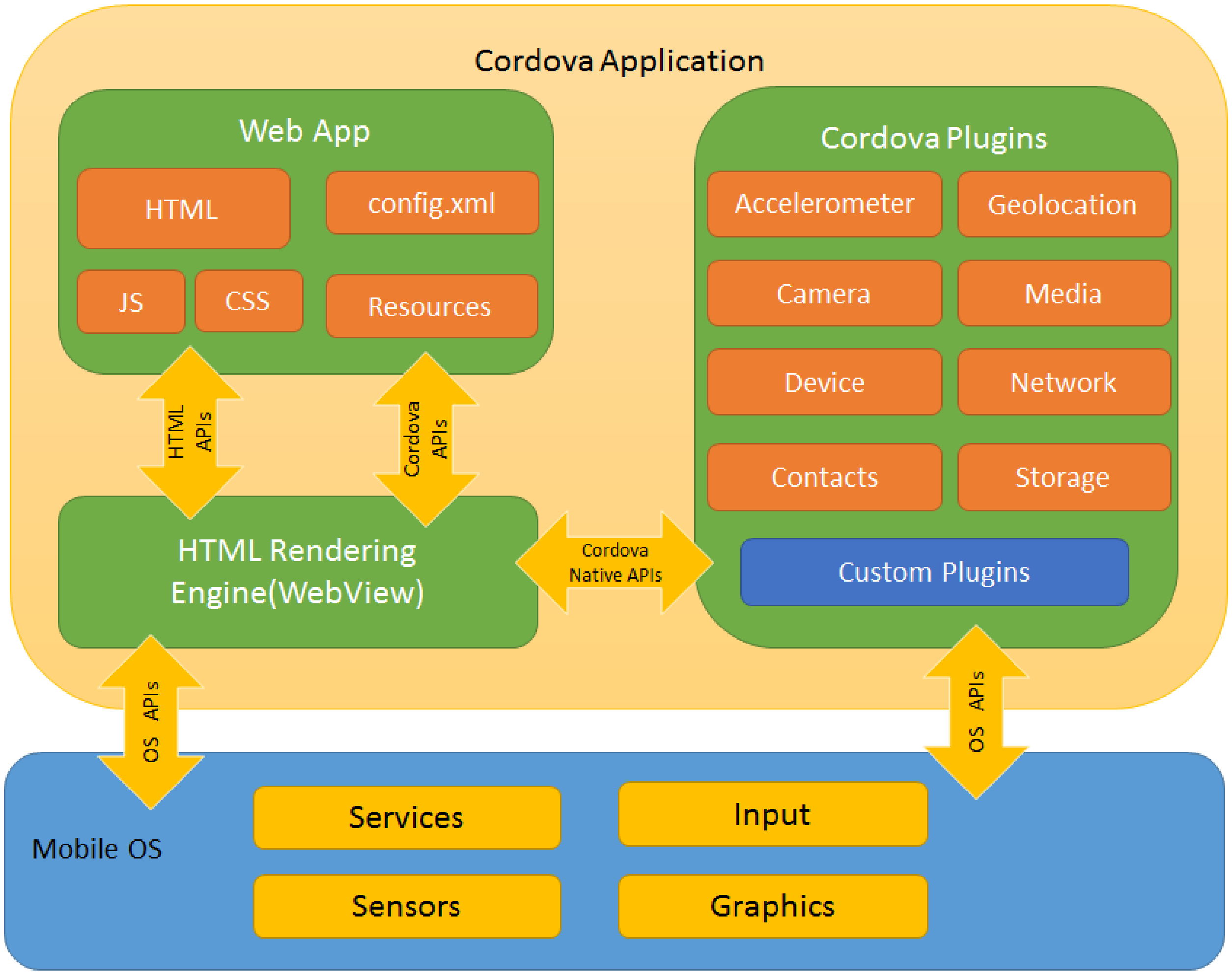
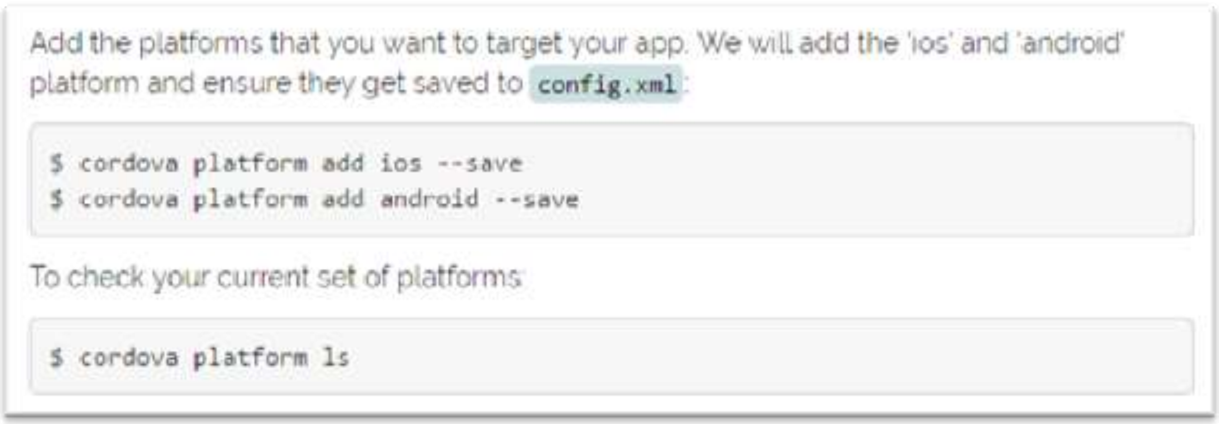


Figure 6 - High-level view of the Cordova application architecture (Apache Cordova, n.d.)

Cordova projects are typically created and updated through the Cordova Command Line Interface (CLI). The CLI allows developers to add a new target platform with a single command as shown in Figure 7. When the developer executes this command, Cordova automatically generates the needed files to port the web application into that platform.


Add the platforms that you want to target your app. We will add the 'ios' and 'android' platform and ensure they get saved to `config.xml`:

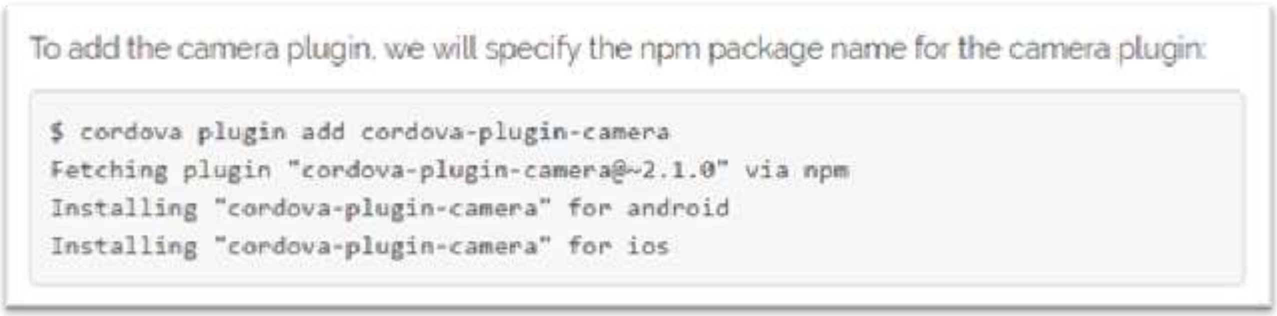
```
$ cordova platform add ios --save
$ cordova platform add android --save
```

To check your current set of platforms:

```
$ cordova platform ls
```

Figure 7 - Example of how to add a platform in Cordova CLI (Apache Cordova, n.d.)

Cordova plugins are important for developers trying to integrate native functionality into their hybrid apps. Plugins provide an interface for native features by wrapping phone operating system APIs with a callable JavaScript wrapper (Apache Cordova, n.d.). Developers can add plugins to a Cordova application through the Cordova CLI. To add the camera plugin, a developer simply executes one command as shown in Figure 8.


To add the camera plugin, we will specify the npm package name for the camera plugin:

```
$ cordova plugin add cordova-plugin-camera
Fetching plugin "cordova-plugin-camera@~2.1.0" via npm
Installing "cordova-plugin-camera" for android
Installing "cordova-plugin-camera" for ios
```

Figure 8 - Example of how to add the camera plugin in Cordova CLI (Apache Cordova, n.d.)

2.3 Ionic

When considering the user experience, Cordova does not provide assistance with building the UI. Fortunately, mobile frameworks such as Ionic are available to assist developers. The use of one of these frameworks drastically reduces the amount of work required to style hybrid applications to have a native look and feel across platforms. Ionic is an open-source framework for hybrid mobile app development. Max Lynch, Ben Sperry, and Adam Bradley of Drifty Co. created Ionic in 2013 with the following intent: “Ionic’s ultimate goal is to make it easier to develop native mobile apps with HTML5, also known as Hybrid apps” (Lynch, 2013).

The Ionic team built the framework upon AngularJS and Cordova. When Ionic was first released, its technology stack offered web developers familiar tools to build hybrid applications with reasonable performance, native look and feel, and responsiveness. Ionic’s command line interface, built on top of the Cordova CLI, provides additional commands such creating an application from a template and testing in browser.

After three years, Ionic claimed the top spot as the leading open source framework for app development with over four million apps made and five million developers (Ionic, n.d.). The success of Ionic v1 led to the release of Ionic v2 on January 25th, 2017 (Ionic, n.d.). The Ionic team built Ionic 1 and Ionic 2 respectively with AngularJS and Angular 2. Both are currently available under a community MIT license which freely allows anyone to use the framework to develop and publish apps (Ionic, n.d.). The Ionic team has developed a number of additional services like a drag-and-drop app editor called Ionic Creator, and cloud services like push notifications and authentication. Ionic offers these additional services at a price that is scalable based on app traffic and growth. This section will cover some of the core advantages and features of both Ionic 1 and Ionic 2.

2.3.1. Ionic 1

The DriftyCo team designed Ionic 1 to make it easier to develop native mobile apps with HTML5 (Lynch, 2013). Ionic 1 focuses on enhancing the look and feel of UI interaction in mobile hybrid applications. The developers of the Ionic framework wanted to target devices of the present and future, realizing that today's browsers and APIs would become exponentially more advanced with the increased performance of HTML5. Ionic 1 is not a replacement for Cordova, but rather an enhancement. Ionic, similar to a traditional web front-end framework such as Twitter Bootstrap, gives developers a collection of UI elements and structure to use as a starting point to develop high quality products.

Ionic comes prepackaged with a set of responsive, native-looking UI components. Ionic offers source code examples for each component and extensive documentation. The documentation provides a layout displaying information and source code for features in the middle, a phone display on the right, and a navigation bar to other features on the left.

The Ionic team styled their components with CSS as shown in Figure 9. This figure shows how default lists, cards, and ranges appear native. These components are additionally easily customizable. To do this, a developer can add custom CSS styling or override the defaults. For even more power, styling in Ionic uses Sass which introduces additional features to CSS like variables and element nesting. Sass allows developers to make large thematic changes with minimal effort.

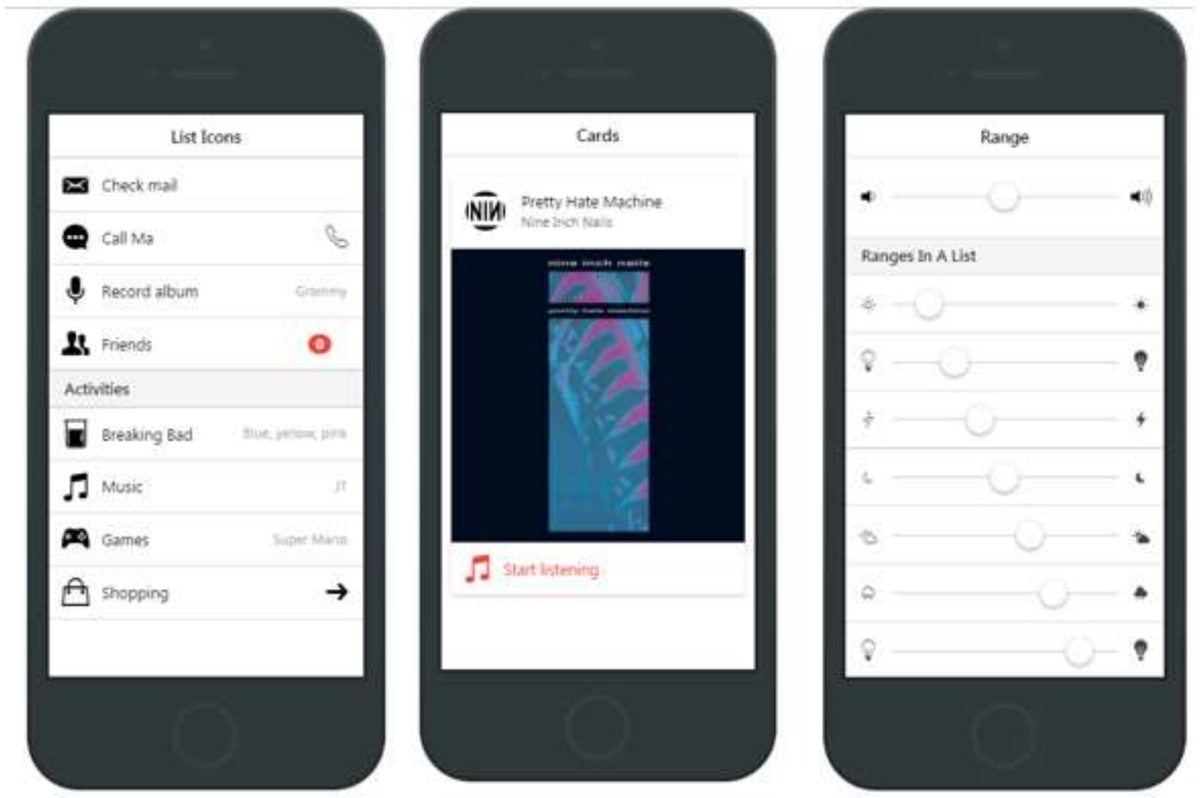


Figure 9 - Ionic 1 list, card, and range components (Ionic, n.d.)

Application scripting is also supported with Ionic to make apps feel more native. This includes animations such as opening a side menu or tab navigation between pages. By freeing applications from the URL bar and adding JavaScript transitions, Ionic shrinks the difference in user experience between native and hybrid application (Lynch, 2013). The responsiveness of AngularJS makes it ideal for creating these animations while operating at high performance.

The open source nature of the Ionic framework encourages a community of shared knowledge. The completeness of Ionic documentation and the constant growth of its community continually promotes best practices and design patterns. With 4 million apps built and community support on Stack Overflow, GitHub, and Ionic’s community forum, there is a wealth of information and support for developers facing bugs or design questions. The success of Ionic 1 led to the Ionic team working with the Angular 2 team to create Ionic 2.

2.3.2. What’s new in Ionic 2

Ionic 2, officially released on January 25th, 2017 offers considerable improvements over Ionic 1. Ionic 2 offers new and improved components, support for implementing native device capabilities with Ionic Native, and a new stack increased performance. The Ionic team improved components such as buttons to look cleaner in comparison to Ionic 1 as shown in Figure 10. Additionally, Ionic 2 components offer higher performance boosts over Ionic 1 and benefits such as styling that make components automatically adjust to each platform. An example of this is the button component for iOS, Android, and Windows platforms as shown in Figure 11. In this figure, there is a list of buttons on iOS, Android, and Windows side by side depicting the automatic native adjustments Ionic 2 does by default. The documentation for each component and examples are available on their website.

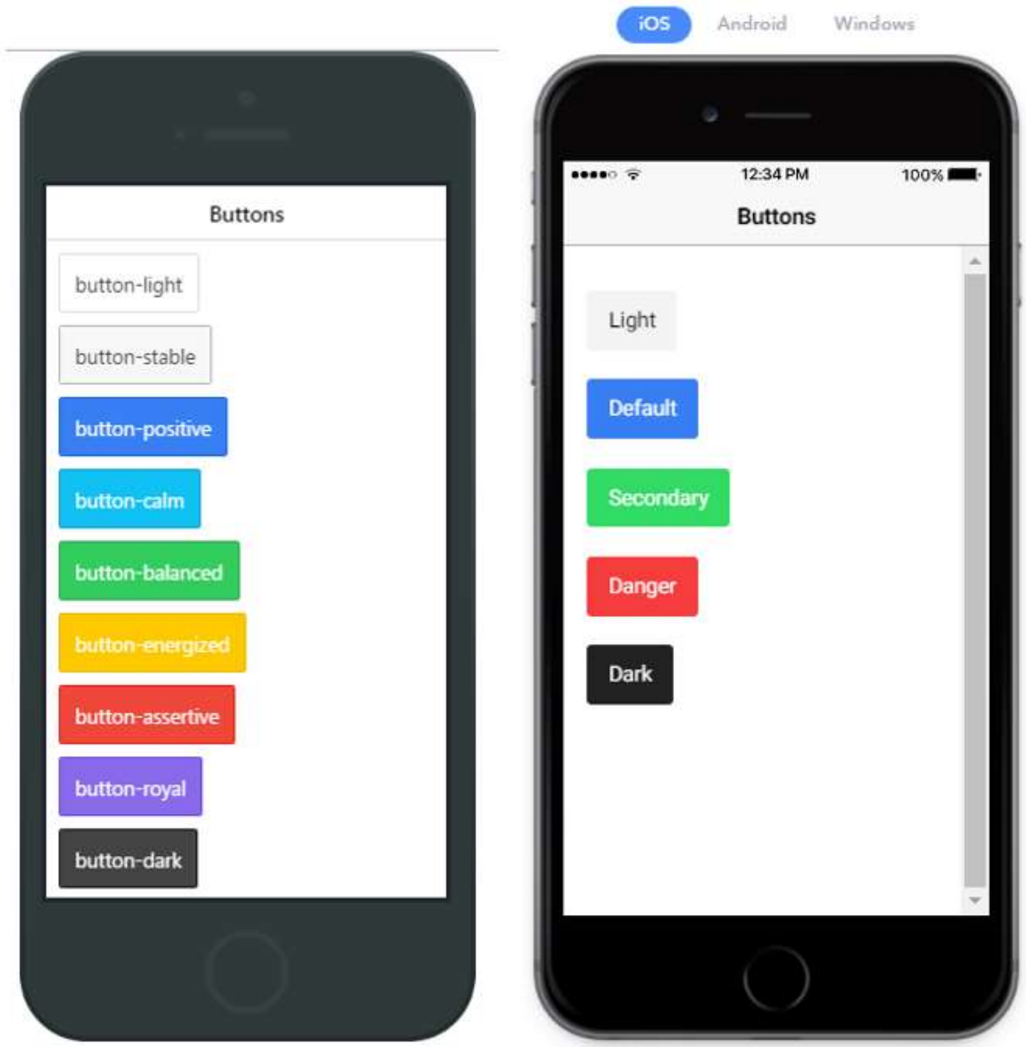


Figure 10 - Comparison of Ionic 1 button styling on left and Ionic 2 button styling on right. (Ionic, n.d.)

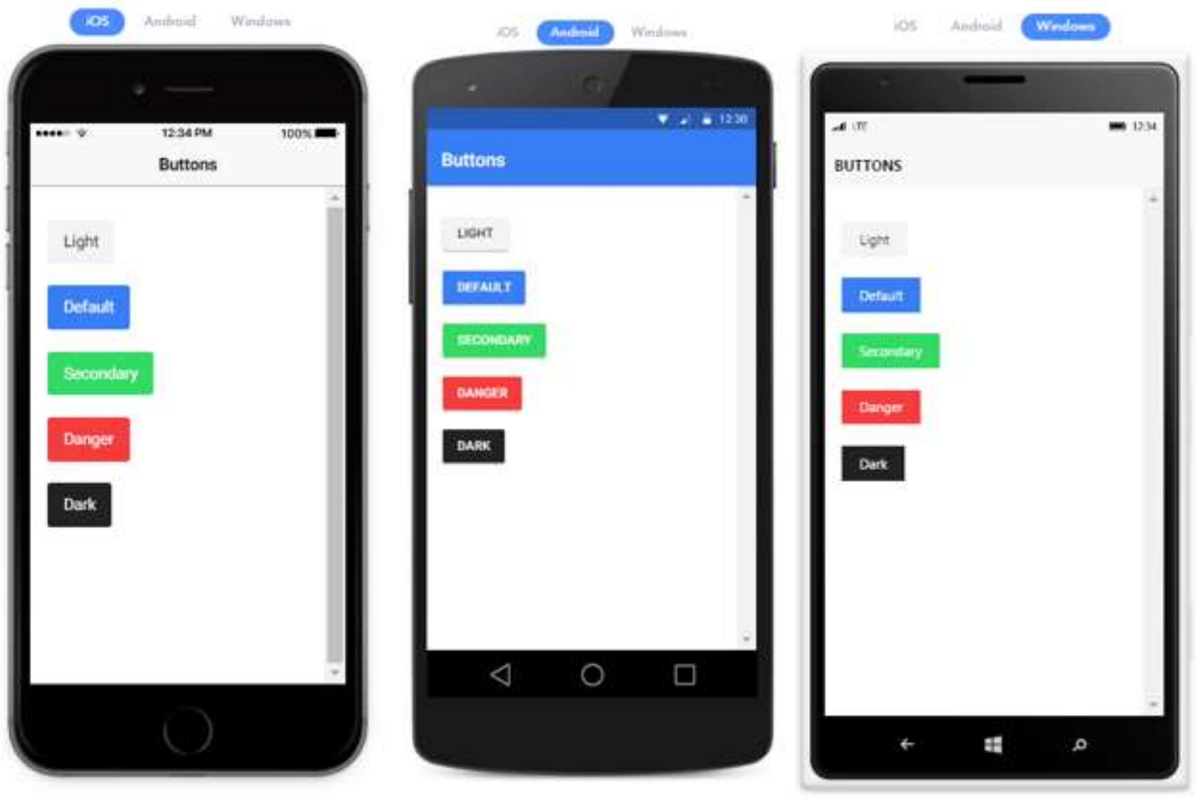
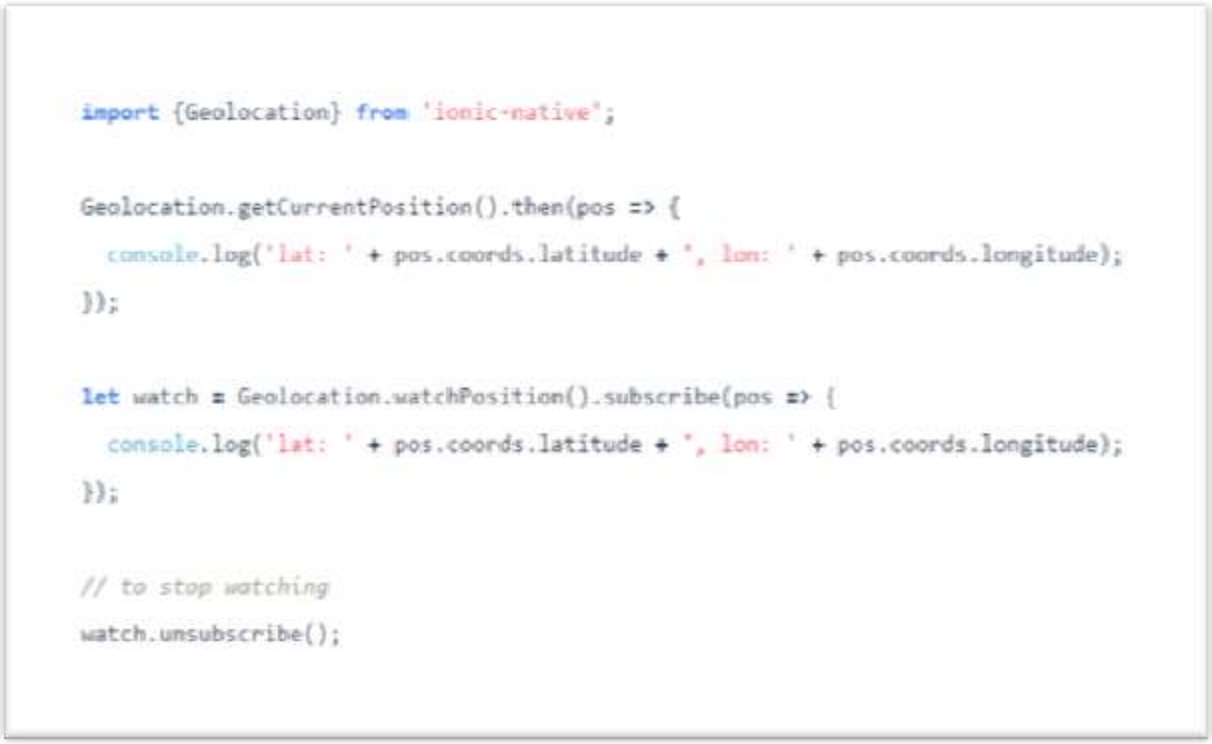


Figure 11 - Ionic 2 button styling for iOS, Android, and Windows (Ionic, n.d.)

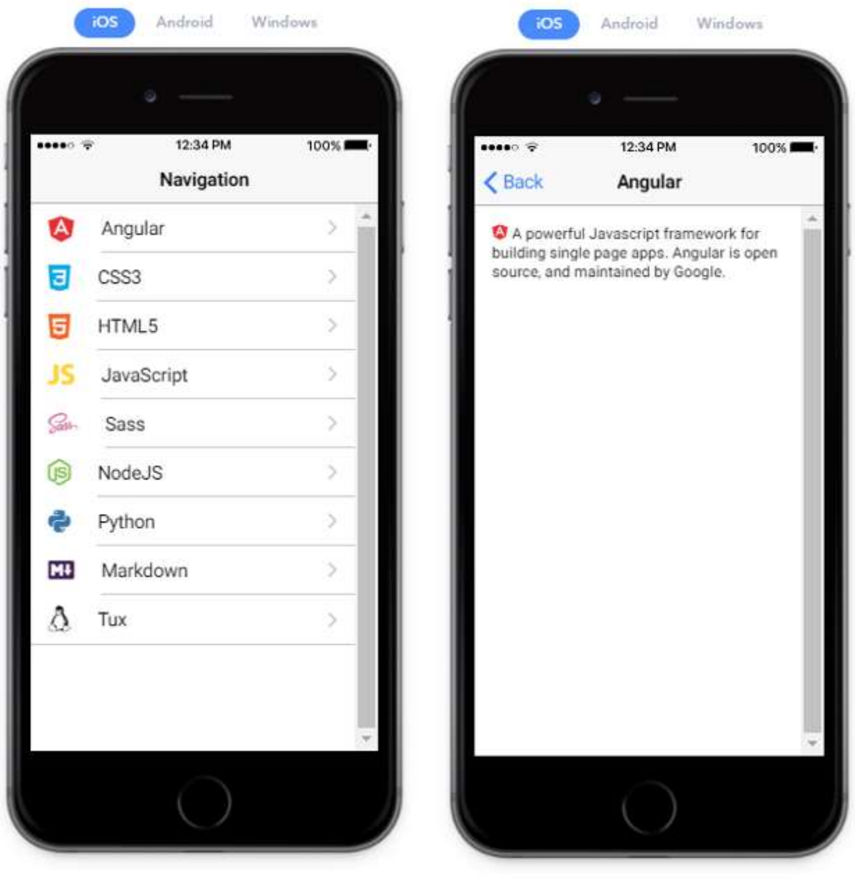
Ionic version 2 offers a fully integrated native plugin system called Ionic Native. This system includes over 70 native features ready for developers to use like other Web API's in Angular 2, complete with support for observables and promises (Ionic, n.d.). Ionic Native wraps all native plugin callbacks in a Promise or an Observable, JavaScript structures made for handling asynchronous functions, providing a common interface for plugins to ensure that native events trigger change detection in Angular 2. This improves the developer's ability to effectively add native features such as notifications, contacts, or camera to an application. Ionic 2's thorough documentation and examples provide additional assistance in implementing these native features. A developer, for example, can easily use Geolocation once they install it from the CLI and imported as a service as shown in Figure 12.



```
import {Geolocation} from 'ionic-native';

Geolocation.getCurrentPosition().then(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
});

let watch = Geolocation.watchPosition().subscribe(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
});

// to stop watching
watch.unsubscribe();
```

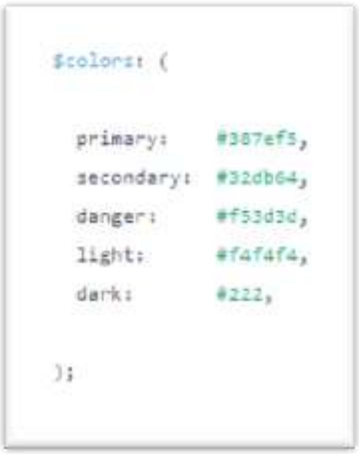
Figure 12 - Example of importing Geolocation plugin with Ionic Native and getting a longitude and latitude (Ionic, n.d.)

Ionic 2 revamped styling to make it easier to brand and style a whole app to fit a theme. The latest supports three modes: iOS, Material Design, and Windows (Ionic, n.d.). The Ionic team calls this Platform Continuity and this means that each platform has a matching look-and-feel and behaves as a user would expect. An example of this is the back button for Android as compared to iOS as shown in Figure 13. Ionic 2 also provides more support than Ionic 1 for creating custom themes that apply style throughout the application. The Ionic team built the framework core on top of Sass to encourage developers to change the defaults and create richer color palettes and themes. Most Ionic 2 components use the “primary” and “secondary” Sass color variables which a developer can modify in one file as shown in Figure 14. These changes then affect the entire project, changing headers, buttons, and other components to match one cohesive style. Ionic 2 additionally includes an extensive set of icons shown in Figure 15, giving developers a wide array of symbols with which to make clean and clear user interfaces.



A powerful Javascript framework for building single page apps. Angular is open source, and maintained by Google.

Figure 13 - An example of Ionic 2 navigation showing the Angular page (right) popped onto the root page (left) (Ionic, n.d.)



```
$colors: (

  primary:    #387ef5,
  secondary:  #32db64,
  danger:     #f53d3d,
  light:      #f4f4f4,
  dark:       #222,

);
```

Figure 14 - Example of the $colors section of the variables.scss file (Ionic, n.d.)

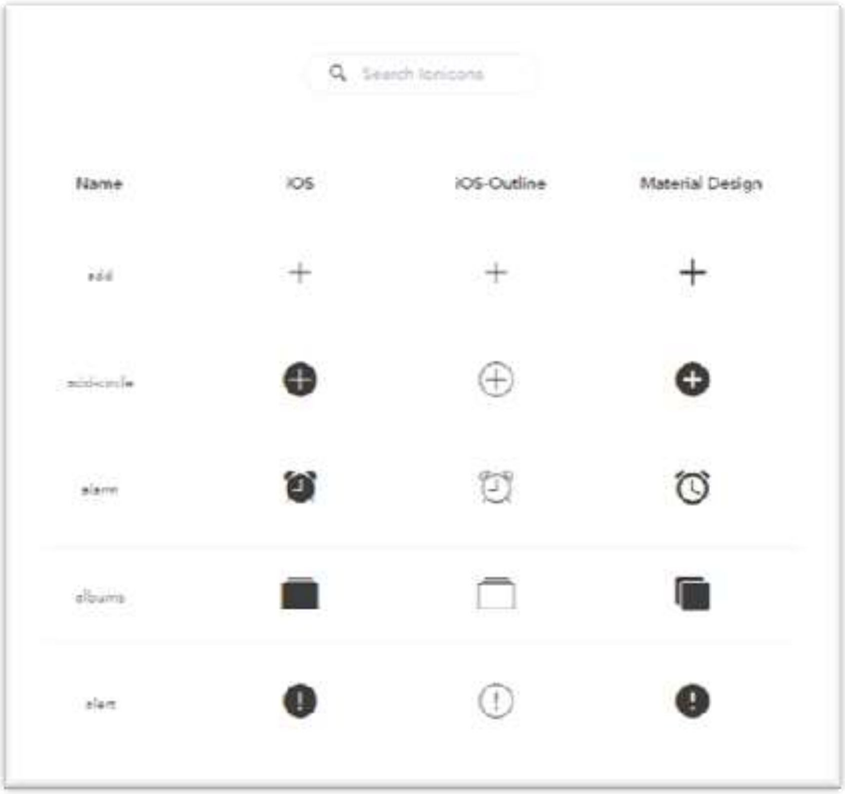


Figure 15 - An example of the icons offered with Ionic (Ionic, n.d.)

Ionic 2 also surpasses Ionic 1 in terms of performance. Features such as virtual scrolling lists improved scrolling speeds up to 60 FPS (Ionic, n.d.). This virtual scroll supports very large lists and even lists of images. The Ionic team also introduced an all new rendering pipeline that reduces layout thrashing and repaints by only redrawing portions of components that change (Ionic). Additionally, the use of Angular 2, which is considerably faster than Angular 1, allows Ionic 2 to inherit performance boosts out of the box. The Ionic team states their commitment to performance and continuously strives to improve everything from boot-time to animation sleekness (Ionic, n.d.).

Ionic 2 offers a navigation stack as opposed to Ionic 1's URL navigation. The navigation stack involves pushing views onto the navigation stack and popping them off. The stack initially starts with a root page and from this root page the system pushes and pops pages onto and off the stack (Morony, 2016). At anytime a developer can set a new root page. Pages not on the root of the stack are automatically updated to include a back button in the navigation bar. When a new page is set as the root it stands alone until the system pushes another page onto the stack. Note that the root component is typically the app.component.ts file and is different from the root page.

2.4 Bootstrap

Bootstrap is a free open source framework for creating well styled, highly responsive websites (Bootstrap, n.d.). While Bootstrap was initially released by and for Twitter employees in 2011, it has grown to be a prominent framework for front end web-development. The Twitter team designed Bootstrap to make creating web applications a fast and easy process with a current focus on mobile first. Bootstrap consists of a well-structured set of CSS classes. Along with highly readable and customizable code, all Bootstrap components are thoroughly documented, which greatly facilitates understanding and utilizing new classes.

The primary Bootstrap feature for designing layout and adding screen size responsiveness is its grid layout. The grid layout uses row and column CSS classes to organize HTML elements in the view. Row elements organize other HTML elements horizontally and column elements fill them, which HTML lays vertically. These column classes can be set to modify and dynamically arrange content based on the size of the screen, for example, on a wide screen 4 columns may span the width of the screen in a row, but on an extra small screen only one column would occupy each row. As shown in Figure 16, the end product is a Web application that adapts its appearance to best suit varying devices and display sizes.

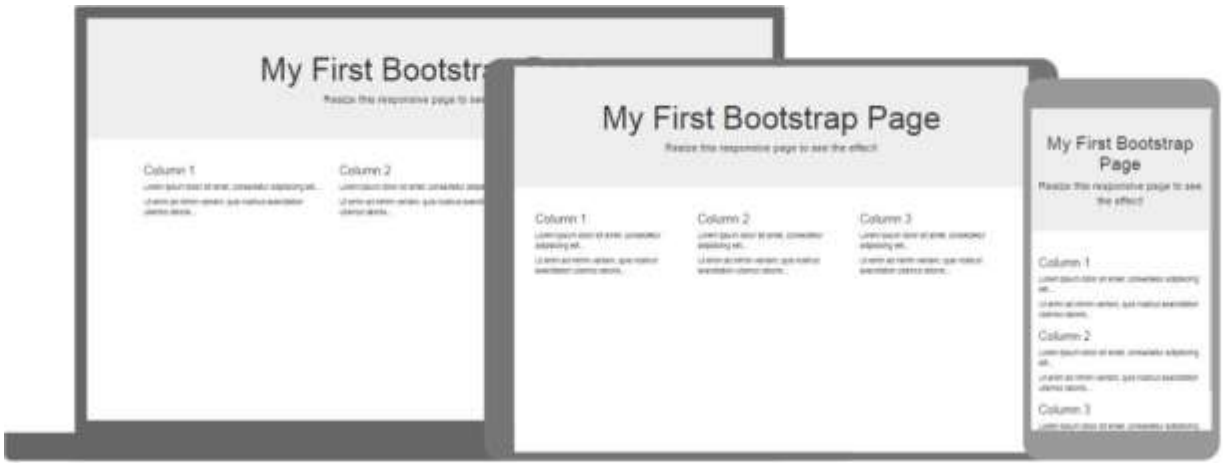
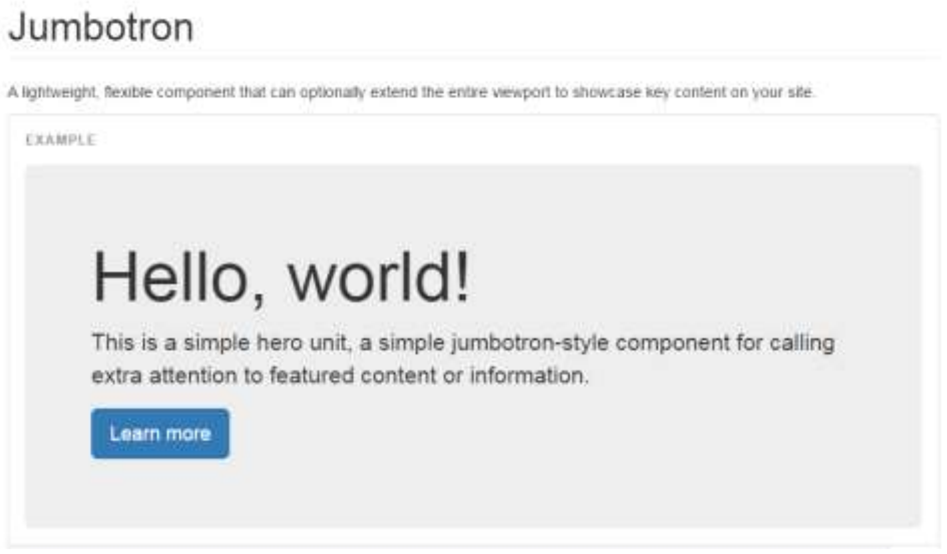


Figure 16 - Bootstrap resizing across multiple devices. (W3Schools, n.d.)

Bootstrap includes styling for common components such as toolbars, buttons, panels, and other displays found in web pages. Figure 17 displays a Bootstrap Jumbotron component, a common web showcase. The CSS classes included in Bootstrap make it easy to standardize the look and feel of a web application and supplementary JavaScript components offer animations for alerts, popovers, etc. Bootstrap has been around for over six years and there is a large community today and many websites built with Bootstrap. Additionally, resources such as Chrome dev tools, accessible from any web page with a F12 command, allow developers to investigate the HTML and CSS components of other web pages making it easy to implement similar styling. With many examples on the Web whether it be from a tutorial or website, Bootstrap is an attractive choice for rapidly styling or prototyping a website.


Jumbotron

A lightweight, flexible component that can optionally extend the entire viewport to showcase key content on your site.

Hello, world!

This is a simple hero unit, a simple jumbotron-style component for calling extra attention to featured content or information.

Figure 17 - An example of a bootstrap Jumbotron component

Another advantage of using Bootstrap is the constantly growing pool of additional templates, plug-ins, and mix-ins (Bootstrap, n.d.). Templates for any traditional login or registration page are widespread and easy to implement. Plug-ins are additional Bootstrap-compatible resources that developers can import easily into an existing project. Plug-ins can be of various sizes or complexity, some provide small features like social networking buttons, others can support more advanced functionality like presenting to the user a guided tour of the site.

2.5 ASP.net Web API

ASP.net is a server-side web application framework developed by Microsoft under the Apache 2.0 License (Wasson, 2015). ASP.Net Web API is specifically designed for creating Application Programing Interfaces (API's) in C#. A common structure for an ASP.net Web API has four main components; controllers, services, repositories and models. Breaking the API into these sections promotes code reuse and helps ensure developers can tailor each segment for one purpose. Note that while controllers and services are both terms used for AngularJS applications, their purposes are different when used to describe ASP.Net.

Controllers are the classes responsible for initially handling incoming HTTP requests. Controllers contain methods which are the endpoints for the various requests that web APIs handle. For example, a class PackageController may have an endpoint for getting a package, for posting a new package, for updating a package or for deleting a package. While these classes are responsible for the receiving the request, in this structure, the system parses requests and then invokes specific services to handle each request, leaving the main business logic outside of the controller.

Repositories are the data access layer for the API, so these classes are responsible for retrieving and storing data from a database. A package repository may provide a method for retrieving a package by id, or another method for retrieving all packages belonging to a user, or possibly still yet another method responsible for retrieving all packages in the database. The actual integration with a database can be standard SQL queries, or a variety of libraries like NHibernate that facilitate database interaction. In such a system, the repositories execute all logic involved in getting specific data from the database. This allows services to access data without needing to implement the specifics of how the system will retrieve the data from the database.

Services are responsible for executing the main business logic of the API. The controllers on the top level call these service methods which instantiate and use references to repositories to access data. An example UserService may have a method for taking in user details, creating a new user that satisfies a user model and then handing the new user to a repository which adds the user to the database. Services may employ multiple repositories in order to access all required data for some operation. A developer should design each service to support one set of operations, such as all operations for users, or all operations for packages.

Models contain the definitions of the data structures and objects present in the application. Services, controllers and repositories all utilize the same models to ensure that the system represents data objects in the same form throughout the API (Kearn, 2015). Models generally map to tables in the database, which allows repositories to easily retrieve and store data in tables by working with objects that satisfy the corresponding model.

2.6 Firebase

Firebase is a Google maintained cloud backend as a service (Firebase 2017). Firebase provides data storage and authentication on cloud hosted servers. Firebase apps are a free service at low usage, but scale with pricing as data access becomes more frequent or user base grows. Firebase provides services beyond storage such as authentication, web hosting, and analytics.

Firebase provides user authentication in two main ways, email and password logins, and social login authentication. Developers can use many different social login providers for a given Firebase application, a few of which are Facebook, GitHub, and Twitter. When a user registers using a social login, the user's credentials are not stored in Firebase; instead the social authentication provider verifies the user and informs Firebase. If a user registers using an email and a password, then the system stores those credentials in Firebase to validate future logins.

Additionally, Firebase offers Firebase Cloud Messaging (FCM), which was previously named Google Cloud Messaging (Aggarwal, 2016). This service provides support for sending push notifications and messages to devices. To use FCM the developer must configure the client side of an application to receive messages from the FCM server. This configuration involves registering the device with the FCM server as well as writing client side code to handle in coming messages. FCM can send messages by interaction with a REST API or by graphical interaction from the Firebase developer console (Firebase, 2017).

3. Project Requirements

The mission of this project is to establish a connection between Endicia and e-commerce Buyers. Endicia believes there is an opportunity to better support its customers, e-commerce sellers, by engaging with their customers, the e-commerce buyers (i.e. recipients of parcels). In this section we list the project goals, explain each goal, and list user stories describing the desired behavior and features to be in the application

3.1. Project Goals

The first step in establishing the project specification was developing a set of project goals to best accomplish the project mission. The goals we laid out are:

1. Establish a method for connection between Endicia and e-commerce Buyers
2. Provide proof of concept for services Endicia can provide to users
3. Develop a clean and responsive user interface
4. Accomplish goals 1, 2, and 3, on multiple platforms (iOS, Android, and Web)

In order for Endicia to connect to e-commerce buyers effectively, we considered the relationship from the side of the buyer. First there must be some initial contact from Endicia that attracts the buyer and provides a compelling reason to join this Endicia service. Next, the registration process must be lightweight to avoid losing a user's interest. Additionally users must be able to trust the security of the system they are joining.

For the connection with buyers to be meaningful, Endicia must offer services that provide value to users. A service is of value to a buyer if it saves the buyer time or money, or provides information that cannot be easily accessed elsewhere. This project focuses mostly on features that provide information about shipped packages, and services that allow users to act upon those packages.

Whether users are forming an initial connection with Endicia or are using services, the user interface and navigation must be logical, clean, and responsive. The user interface should present important services and features in the forefront of the application.

The last project goal is to determine the feasibility of developing an application across different mobile platforms and web browsers. With these goals and our time constraints in mind, we established that the scope for our project would be that of a prototype. In order for us to consider the project complete, the prototype must fulfill all project goals, include a proof of concept for a wide variety of features, and deploy on both web and mobile platforms.

3.2. Project Requirements

Once broad requirements were set, we created a set of user stories to determine what features and services we would implement. Table 1 shows the complete list of user stories. These user stories each generally pertain to one of three main groups: user accounts, package tracking, and mobile device features. Furthermore, during development each user story directly translates to a feature or features for implementation.

Table 1 - User Stories

Group	**User Story**
User Account	As a new user I want to be able to register an account.
User Account	As a user I want to be able to login to an existing account.
User Account	As a user I want to be able to log out.
User Account	As a user I want to be able to modify account information like name and address.
User Account	As a user I want to be able to validate my address.
Package Tracking	As a user I want to be able to add packages to my account so I can track them.
Package Tracking	As a user I want to be able to view a list of packages I am tracking.
Package Tracking	As a user I want to be able to view the details of a package I am tracking, including transit details, carrier and status.
Package Tracking	As a user I want to be able to opt in to automatically track packages mailed to my address.
Mobile Devices	(Push Notifications) As a user I want to be able to receive notifications when new packages sent to my address enter the system.
Mobile Devices	(Geo Location) As a user I want to be able to view a map.
Mobile Devices	(Camera) As a user I want to be able to submit an image of a damaged package for insurance purposes.

The first group of user stories covers user account creation and modification. Many prototype applications may skip such user stories, since account creation and modification is fairly similar and trivial across applications. However, to establish a meaningful connection with the user, the user must be able to create an account. Thus, implementing an effective account system is within the scope of this project.

The next group of user stories outlines the core services provided by the prototype application. These user stories enable a user desire to organize packages and track the status of inbound packages. A unique offering that we provide to users is an auto-tracking ability. This feature requires a user to first undergo a process of identity verification, after which the user can automatically track any package addressed to them that enters the application's backend.

The last group of user stories covers features which particularly well-suited for mobile devices. Each of these user stories utilizes a device specific functionality such as push notifications. We specify these feature names in the mobile device user stories.

4. Design

The next phase of the project after requirements was application design. During design, decisions were about stack architecture, application appearance, database design, and user experience. Each step of the design process strives to ensure that we met our project requirements.

4.1. System Architecture

Creating a cross platform application requires careful design of overall stack architecture. In order to reduce development time and provide consistent user experience across platforms, it is important to share code and resources wherever possible. Figure 18 shows the overall structure of the application. In the figure, rectangles with blue coloring indicate aspects the project team implemented. As stated in the key, solid lines show HTTP connections and dotted lines show a NHibernate connection between a database and the backend. We split the application into three main sections, the frontend, the backend, and the data layer.

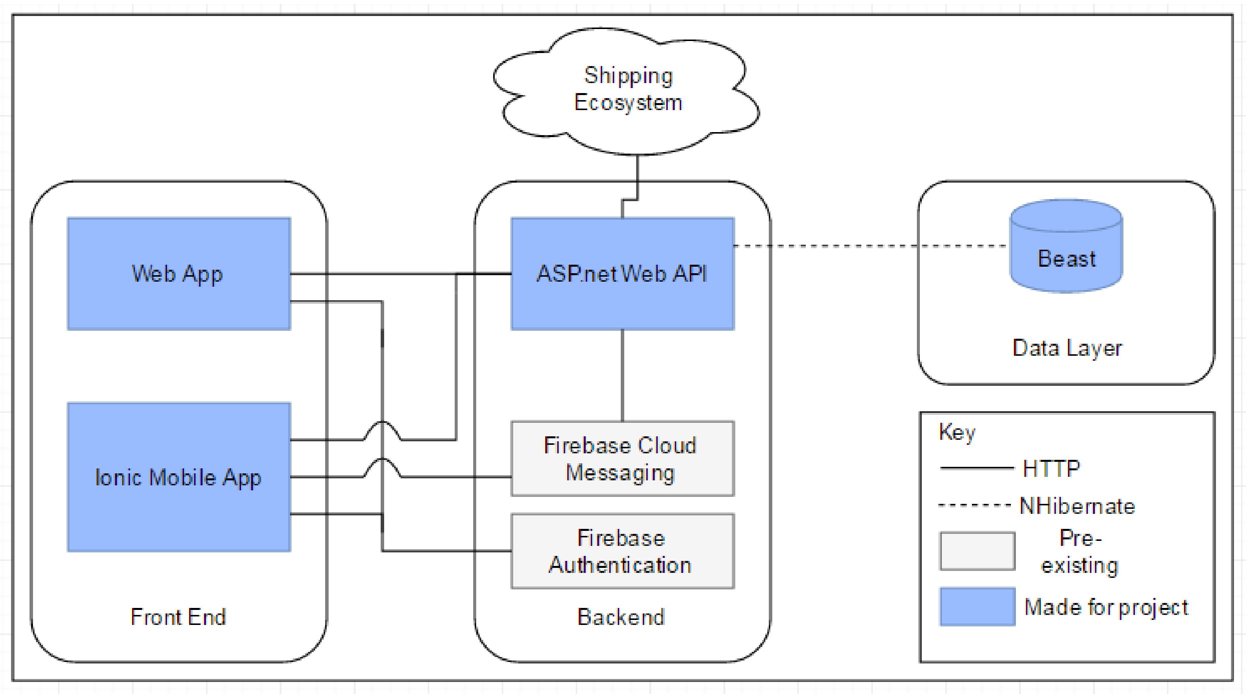


Figure 18 - An overview of the project architecture

4.1.1. Front End

The front end consists of two parts: the Web app portion and the Mobile app portion. Figure 19, illustrates the technologies used for each front end. Notably, we took advantage of increased performance and code modularity by writing the core of both in Angular 2/Typescript. Angular 2 allowed us to reuse common code such as Angular service and, HTTP calls. Building upon our core Angular applications we utilized different frameworks for UI styling and platform specific capabilities.

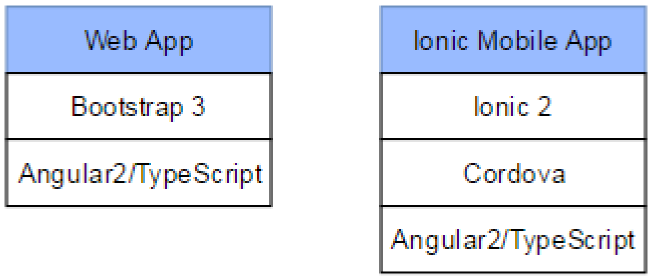


Figure 19 - Front end technology stack for Web Application and Mobile Application

The use of device specific capabilities such as a camera or file system led to a key diversion between applications. On mobile, we access the Cordova camera, geolocation, and push notifications capabilities. These features all require plugins specific to the mobile platform accessed through Ionic Native. To do this for web, a number of API's and plugins exist to carry out the same functionality, but the implementations are slightly different.

Furthermore, a major difference between platforms resides in the design of the UI. On mobile, we used the Ionic framework to design the UI. This framework allowed us to place components such as buttons and tabs already styled for each mobile platform. Similarly, the Bootstrap framework provided us with components that look and feel like a web site and easily adjust to different screen sizes. Both frameworks greatly aided our UI design; however, we could not reuse the code between the two.

In comparison, we developed the Ionic mobile app and web app with similar stacks and code structure, making it feasible to transfer features between the two. We implemented both with the same core business logic, but UI components and device capability implementations differ making maintenance for some features require double the work.

4.1.2. Backend

The backend contains two main sections, an ASP.net Web API, and two Firebase services, Authentication and Cloud Messaging, provided by Google. The Web API acts as the main backend and link between the data layer and front end and provides HTTP endpoints for both the Web app and Mobile app. Authentication and push notifications require significant infrastructure, the implementation of which was beyond the scope of this project. To provide authentication and push notifications we used Firebase Authentication and Cloud Messaging to supplement the Web API. The backend interfaces with the data layer by using the NHibernate framework to map C# classes to data in the data layer.

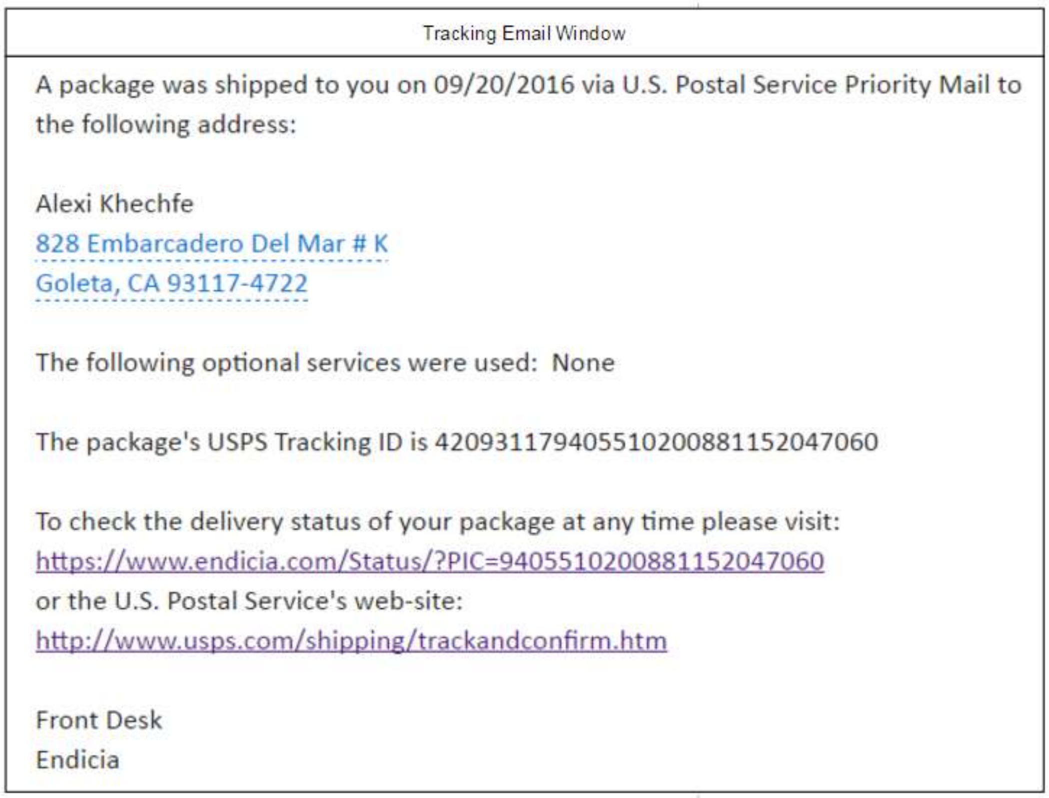
The data layer for this project consists of two sections: Beast, a database structured and populated by the project team, and Endicia's databases, VPO and VPO1Data. This prototype directly queries VPO, however in the future, Endicia's data would be considered part of the shipping ecosystem, and would be imported into Beast through the backend like data form Endicia's partners.

Endicia's databases contain real shipping data and extensive information about each package, tracking information and shippers. Beast contains only the information needed for the mobile and web applications. In order to test the applications, we populated Beast with a mixture of fabricated test data and real data ported in from Beast through the Web API.

4.2. Workflows

Stack architecture considers the technological aspects of design, however in order to begin fulfilling requirements and user stories, we established workflows. Workflows are the patterns in which the user will interact with the application, and provide a high level idea of what paths a user may take when navigating between application components.

Before jumping into workflows within the application, we first wanted to establish the method of capturing users. The initial exposure to users is particularly important since establishing a connection with the buyer is one of Endicia's main goals for the project. We focused on streamlining the registration process by taking advantage of the tracking email that e-commerce buyers already receive from Endicia when they order a package through one of Endicia's partners. The tracking email Endicia currently sends to e-commerce buyers as shown in Figure 20 is our initial entry point for a new user. The email offers tracking for a specific package and a compelling reason for a user to register.


Tracking Email Window

A package was shipped to you on 09/20/2016 via U.S. Postal Service Priority Mail to the following address:

Alexi Khechfe
828 Embarcadero Del Mar # K
Goleta, CA 93117-4722

The following optional services were used: None

The package's USPS Tracking ID is 420931179405510200881152047060

To check the delivery status of your package at any time please visit:
https://www.endicia.com/Status/?PIC=9405510200881152047060
or the U.S. Postal Service's web-site:
http://www.usps.com/shipping/trackandconfirm.htm

Front Desk
Endicia

Figure 20 - The tracking email currently shipped to recepients of a package from one of Endicia's partnered e-commerce merchants.

Figure 21 shows our application's proposed path taken by a user opening the tracking email on a desktop. Immediately after selecting the link, the system brings the user to the welcome page showcasing the BuyerLink product. From the welcome page, users can decide to login, register, or not proceed with BuyerLink. If a user decides to continue, we prompt them to login or register with their credentials. After a successful login or registration, the user enters the normal flow of application use.

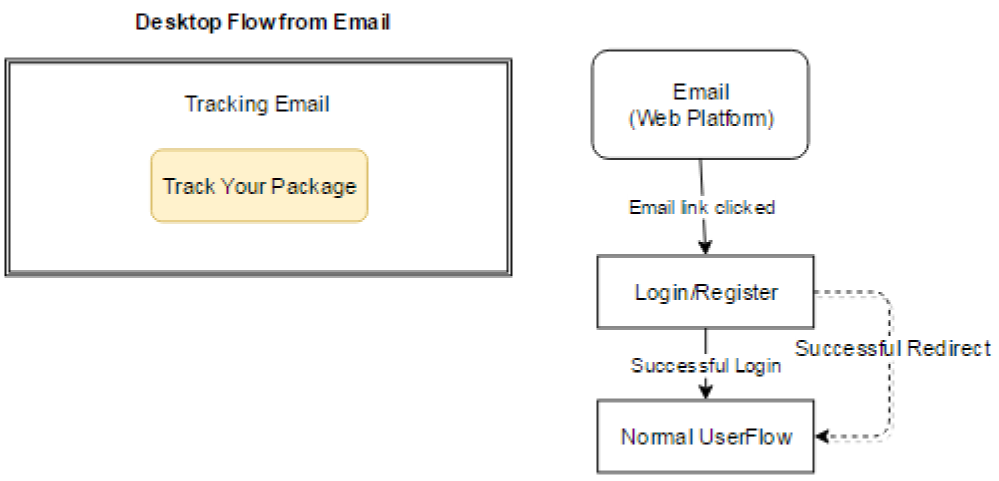


Figure 21 - Login flow for a user on a desktop computer

Figure 22 shows the flow of a user opening a tracking email from a mobile device. The link embedded in the email is a deeplink, meaning when selected on a mobile device, if an application is present, the system opens the mobile application. This allows us to direct users to the app from the tracking link if they already have the app installed, skipping any need to enter a mobile browser. Once the mobile application is open, if the app recognizes the user as logged in, the system brings the user directly to the package details page. If this is a first time the user opens the corresponding package, the package is automatically added to the users account. On the other hand, if the app does not recognize the user as logged in, the system brings the user to a modified package details screen with a button to redirect the user to the login page. If the application is not installed, the user's phone opens the mobile friendly website in a default web browser.

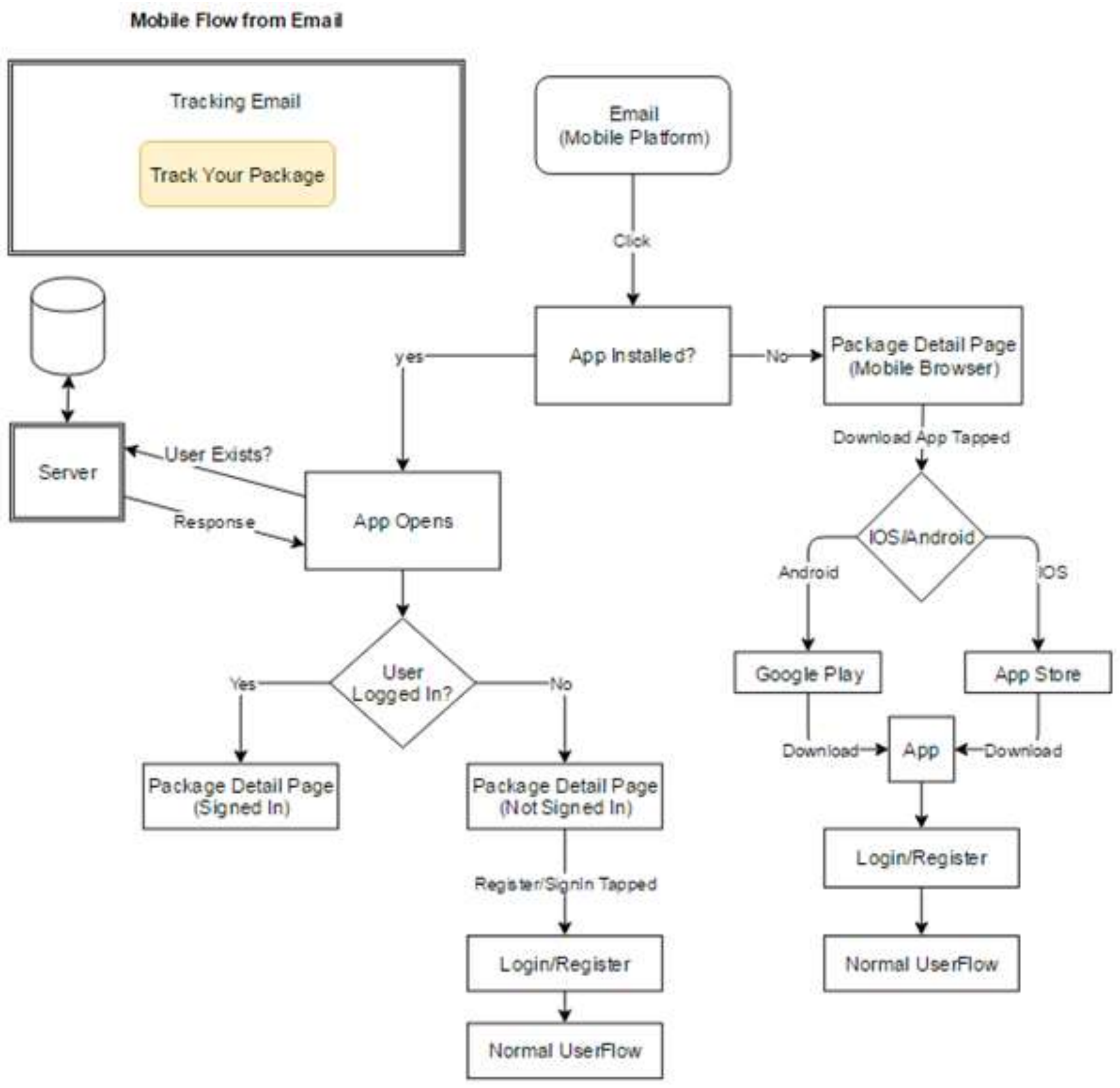


Figure 22 - Flow of a user opening the tracking email on a mobile device.

With the registration workflows outline, we can focus on how users interact with the main body of the application. Before starting from scratch to design a plan for application navigation, it is helpful instead to examine established structural paradigms which are suitable for web and app development. One model of particular use for this project is the *multi-*

dimensional hierarchy where the system gives the user many ways to browse the same content (Hunt, 2006). In Figure 23, boxes denote pages or views of the application, and shading shows the range of pages which users can navigate to from the page at the top of the shading. This structure is highly appropriate for an application like ours because multiple methods of navigation grant a user a fluid user experience and easy access to features and information. A good example of a web site that uses this structure is Amazon, which lets you browse your order history or to navigate to one page, such as a specific product's detail page, in many different ways.

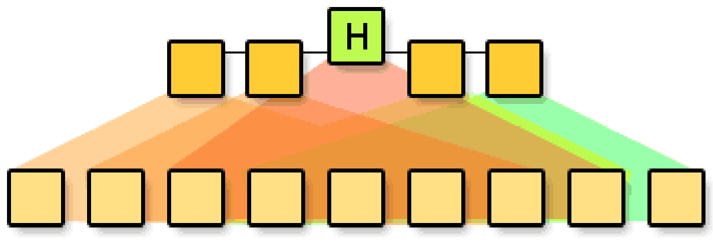


Figure 23 - Multi-dimensional hierarchy (Hunt, 2006)

Figure 24 illustrates the application structure for the mobile application. The login and registration pages are the initial gateway for the user, after which the system brings the user is to the application home. Home consists of three tabs, dashboard, package-list, and shipping. While in any of the tabs of Home, users can access the side menu, which contains access to account details, settings, help, and 'Opt-In to AutoTrack'. AutoTrack allows users to automatically track all packages sent to their address after verifying their identity and address. This application is a multi-dimensional hierarchy, with pages generally accessible through a parent page, though the user can access the print label page through both the package-details page and the dashboard page.

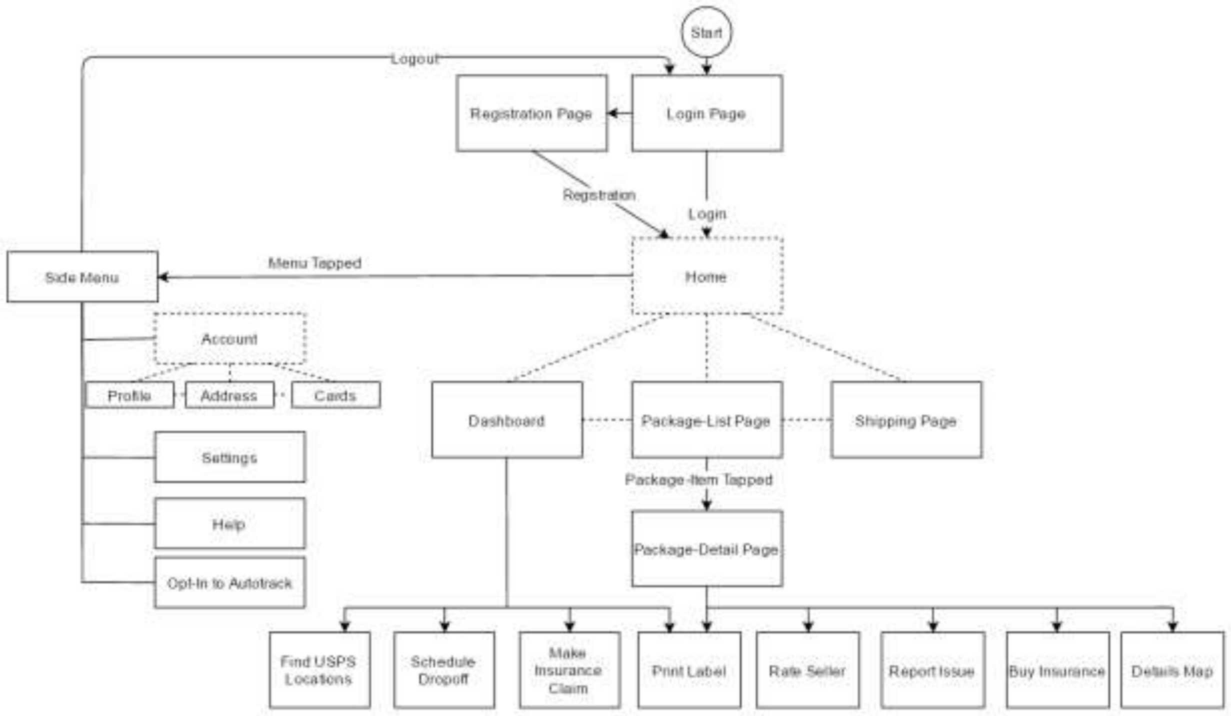


Figure 24 - Pages of the mobile application. Dotted line items (Home and Account) denote pages composed of tabs.

Figure 25 shows the navigation and structure of the web application. The web application has significantly fewer features than the mobile application. It focuses only on users who received tracking emails from Endicia. The application brings these users to either login or registration and after to the package details page corresponding to the email. Notably, we ease the registration process by auto-populating data such as the name and address from the package. From the package details page a user can also navigate to the ‘My Packages’ page which displays a list of his or her packages.

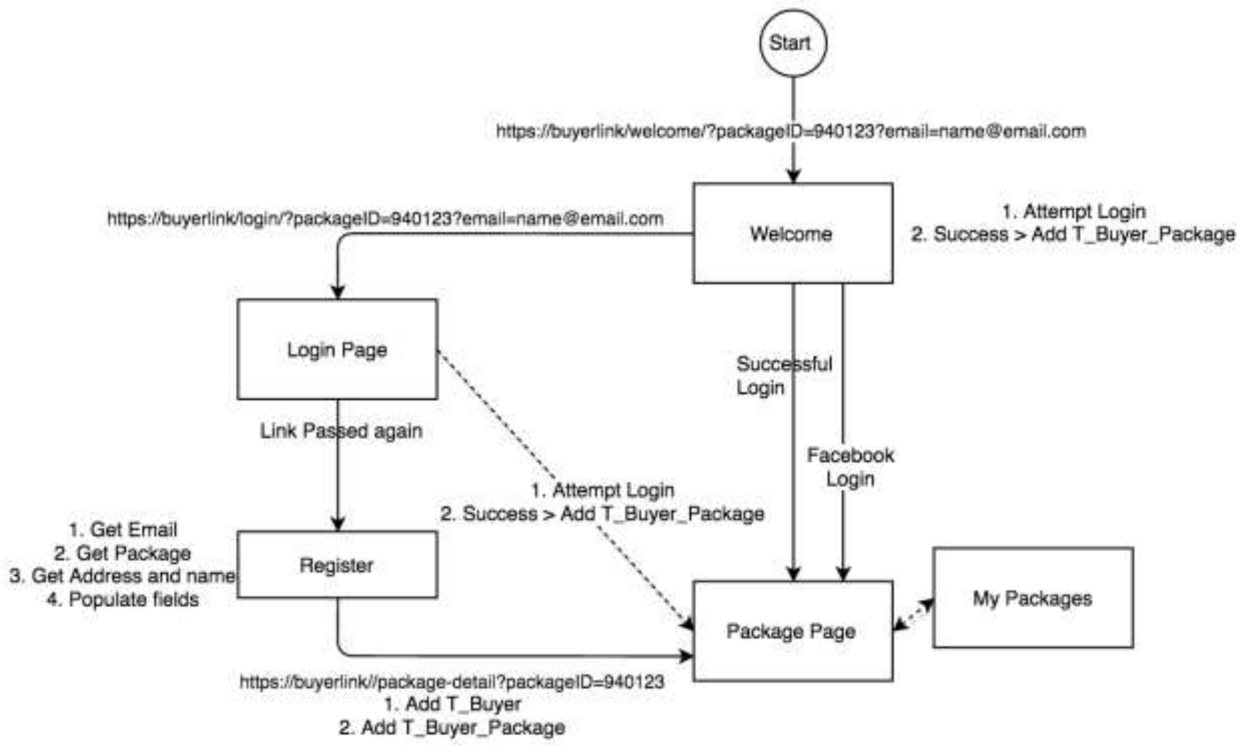

Start
https://buyerlink/welcome/?packageID=940123?email=name@email.com
Welcome
1. Attempt Login
2. Success > Add T_Buyer_Package
https://buyerlink/login/?packageID=940123?email=name@email.com
Login Page
Successful Login
Facebook Login
Link Passed again
1. Attempt Login
2. Success > Add T_Buyer_Package
1. Get Email
2. Get Package
3. Get Address and name
4. Populate fields
Register
Package Page
My Packages
https://buyerlink//package-detail?packageID=940123
1. Add T_Buyer
2. Add T_Buyer_Package

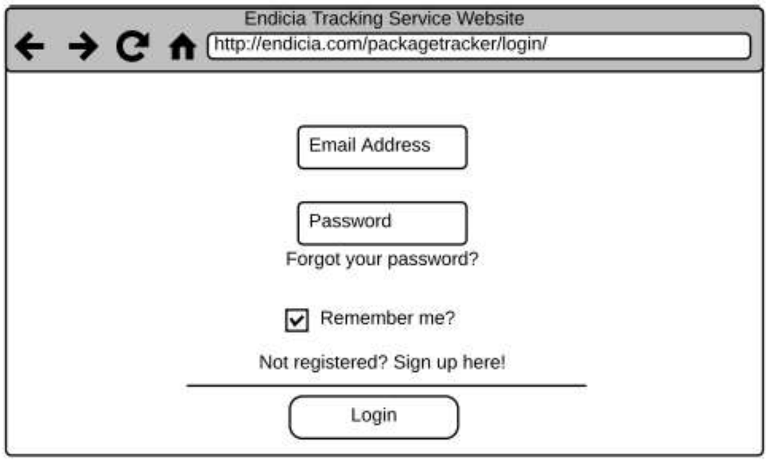
Figure 25 - Pages and navigation for the web application

4.3. Wireframes

Users desire a product that is usable, accessible, and responsive. To empower the user, we focused on encompassing these aspects of the user experience by creating intuitive pages with information and features located in logical places. To provide the basis for the user interface, we developed a series of wireframes. Wireframes specify the layout and design of a user interface with a focus on element placement and positioning. By designing effective wireframes we can ensure that pages include the proper UI elements to accommodate the user stories outlined during requirements. We developed wireframes for both mobile and web to illustrate account registration, logins, and a user dashboard.

4.3.1. Account Registration Wireframes

The first wireframes we created were for logins and account registration for the web application. The UI for Logins on the web as shown in Figure 26 is a simple form, asking for user credentials, and providing a login button. Additional UI elements are present for users to retrieve a forgotten password or create an account if they are not registered.



Endicia Tracking Service Website
http://endicia.com/packagetracker/login/
Email Address
Password
Forgot your password?
Remember me?
Not registered? Sign up here!
Login

Figure 26 - Initial web login

Figure 27 is the wire frame for user registration on the web. This asks users to enter account details, email, password, first name and last name. The user is also asked for shipping information so that we could support future services requiring a user's address.

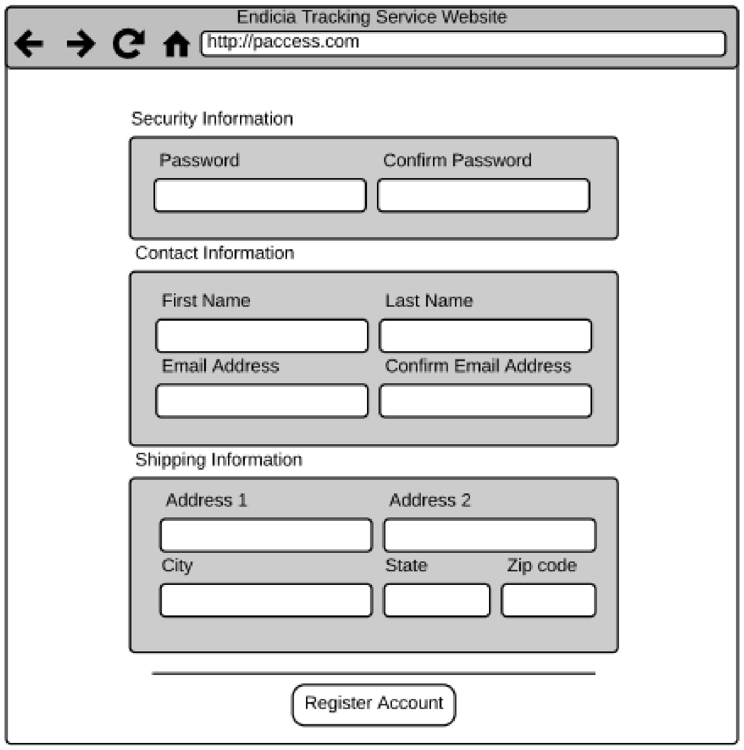
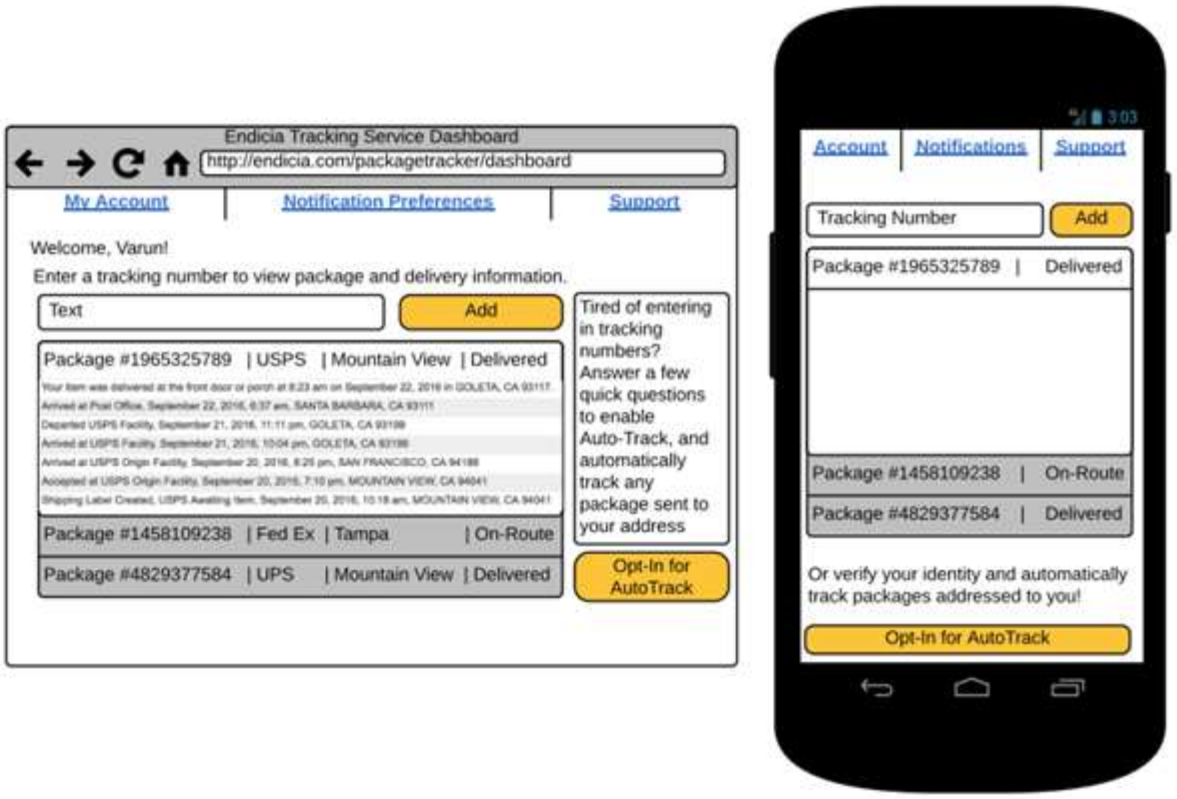


Figure 27 - Initial web registration page

4.3.2. Dashboard and Package Tracking Wireframes

Designing the dashboard required multiple iterations, and had impacts on redesigning application work flow as well. We planned the for the dashboard to hold almost all information relevant to a user, including the list of packages, package details, the ability to add packages, and the option to Opt-In for AutoTrack. Figure 28 shows as these components as well as links to pages for My Account, Notification Preferences, and Support.



Endicia Tracking Service Dashboard

http://endicia.com/packagetracker/dashboard

My Account | Notification Preferences | Support

Welcome, Varun!

Enter a tracking number to view package and delivery information.

Text | Add

Package #1965325789 | USPS | Mountain View | Delivered

Your item was delivered at the front door or porch at 8:23 am on September 22, 2016 in GOLETA, CA 93117.
Arrived at Post Office, September 22, 2016, 6:37 am, SANTA BARBARA, CA 93111
Departed USPS Facility, September 21, 2016, 11:11 pm, GOLETA, CA 93199
Arrived at USPS Facility, September 21, 2016, 10:04 pm, GOLETA, CA 93199
Arrived at USPS Origin Facility, September 20, 2016, 8:25 pm, SAN FRANCISCO, CA 94188
Accepted at USPS Origin Facility, September 20, 2016, 7:10 pm, MOUNTAIN VIEW, CA 94041
Shipping Label Created, USPS Awaiting Item, September 20, 2016, 10:18 am, MOUNTAIN VIEW, CA 94041

Package #1458109238 | Fed Ex | Tampa | On-Route

Package #4829377584 | UPS | Mountain View | Delivered

Tired of entering in tracking numbers? Answer a few quick questions to enable Auto-Track, and automatically track any package sent to your address

Opt-In for AutoTrack

Account | Notifications | Support

Tracking Number | Add

Package #1965325789 | Delivered

Package #1458109238 | On-Route

Package #4829377584 | Delivered

Or verify your identity and automatically track packages addressed to you!

Opt-In for AutoTrack

Figure 28 - Initial wireframe for web (left) and mobile dashboard (right)

Review of the initial dashboard revealed that our application needed to showcase more services than just package tracking. Without additional features, our application would be poor competition for package tracking applications that already exist. To adjust for the second iteration of wireframes, we incorporated recent activity and a group of services into the dashboard. We moved much of the initial dashboard functionality into new package list and package detail pages. We also began exploring options for a dashboard that could present package information in interesting ways. One possible dashboard display, as shown in Figure 29, was a calendar where the system could mark dates with package arrivals and tracking events.

The services displayed on this dashboard iteration include opting into auto-track, adding a package to the account, printing a shipping label and returning a package.

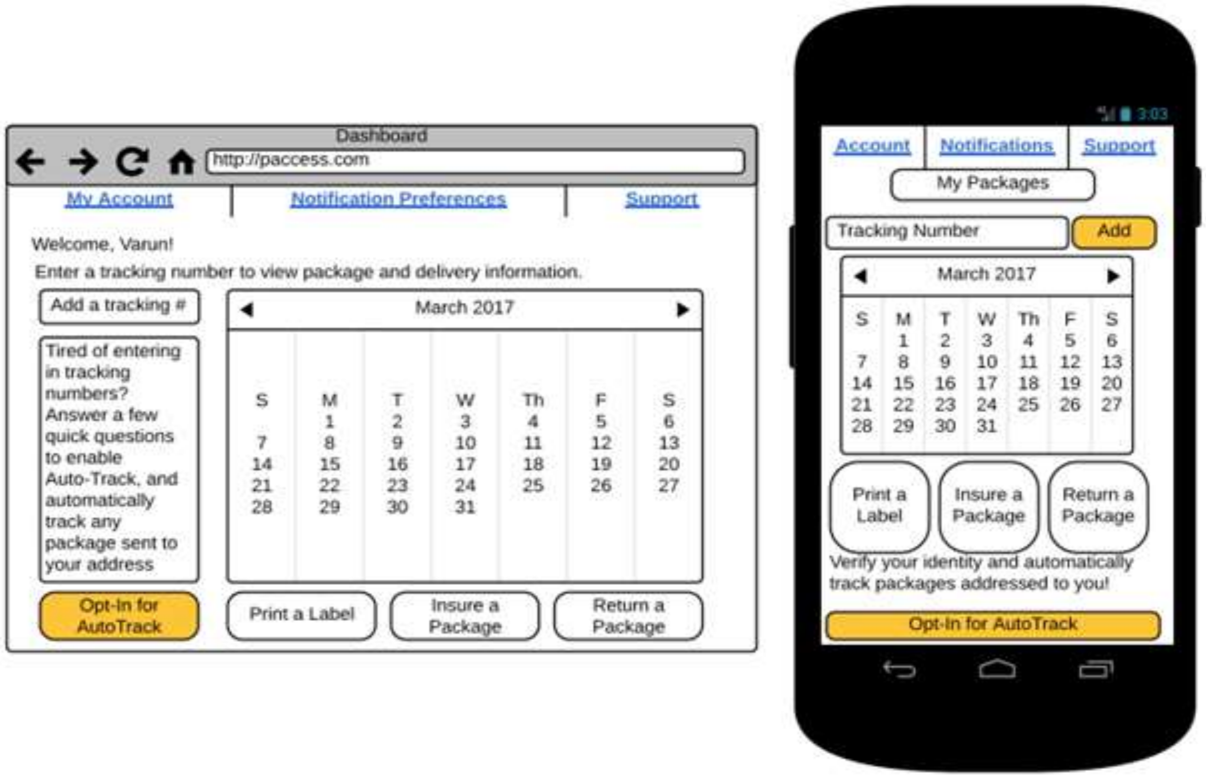


Figure 29 - Second iteration of dashboard design for web and mobile

Figure 30 shows the wireframes for the new package-list page, since the dashboard no longer included this information. The package list wireframes display all of a user's packages. On this page the system offers the user the ability to add packages to their account by entering a tracking number. The page displays each package linked with the account including its status and tracking number. The web version makes use of the additional space available to show the logos of the merchants who sold each package.

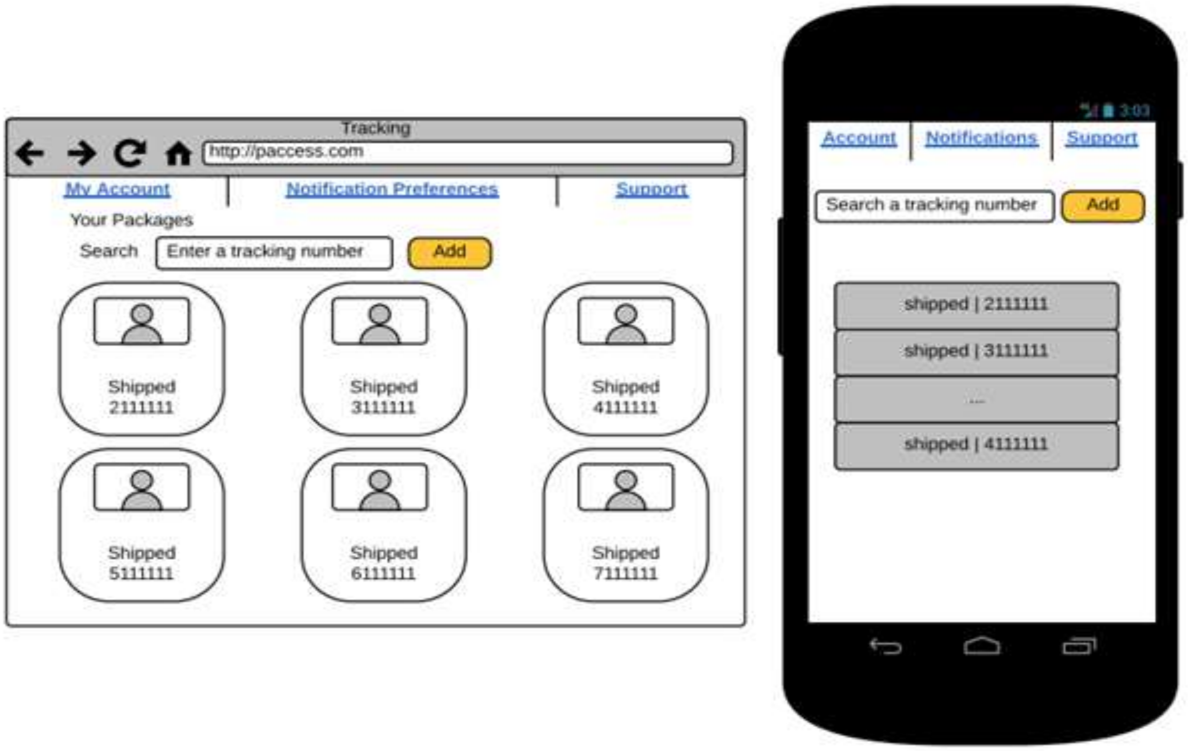
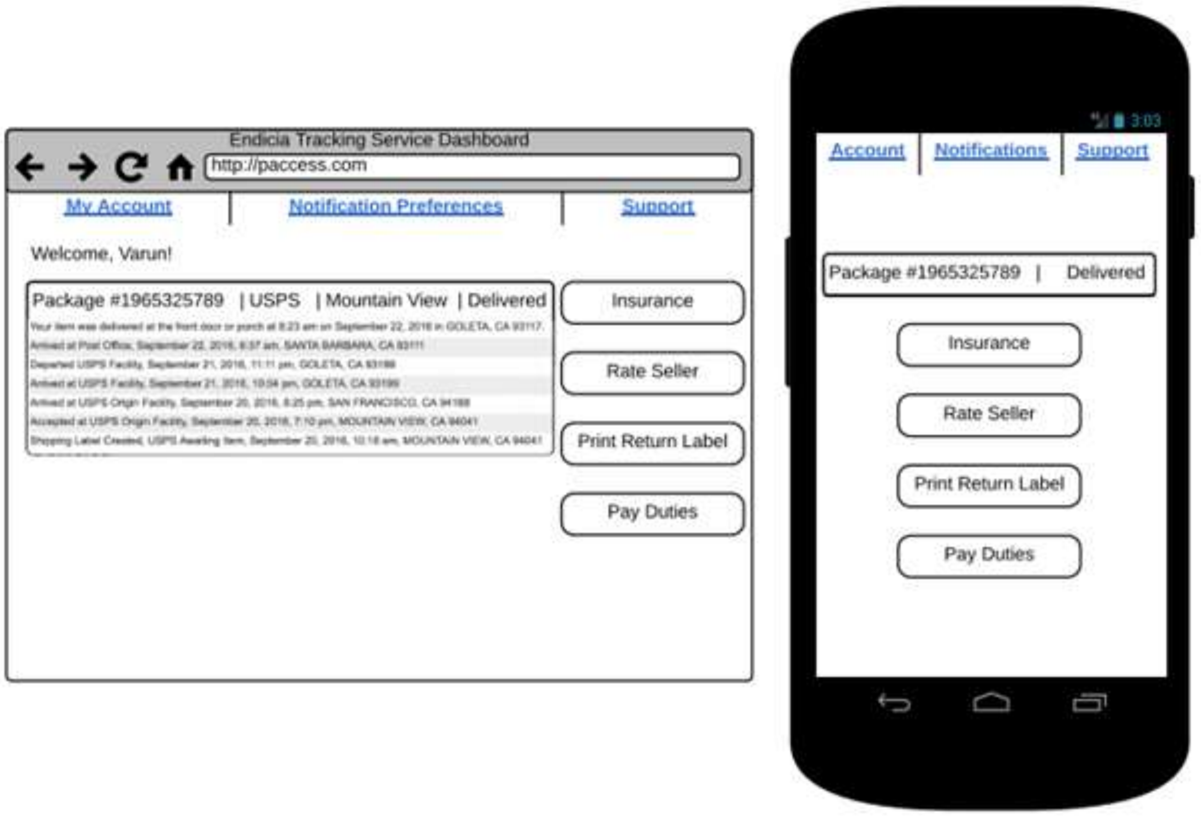


Figure 30 - Initial wireframes for a package list on web and mobile

The page is quite basic, but including a logo of the merchant would strengthen the relationship between the e-commerce buyer and seller. If a user clicks a package, the system brings the user to the package detail page as shown in Figure 31.



Your item was delivered at the front door or porch at 8:23 am on September 22, 2016 in GOLETA, CA 93117.
Arrived at Post Office, September 22, 2016, 8:37 am, SANTA BARBARA, CA 93111
Departed USPS Facility, September 21, 2016, 11:11 pm, GOLETA, CA 93199
Arrived at USPS Facility, September 21, 2016, 10:04 pm, GOLETA, CA 93199
Arrived at USPS Origin Facility, September 20, 2016, 8:25 pm, SAN FRANCISCO, CA 94188
Accepted at USPS Origin Facility, September 20, 2016, 7:10 pm, MOUNTAIN VIEW, CA 94041
Shipping Label Created, USPS Awaiting Item, September 20, 2016, 10:18 am, MOUNTAIN VIEW, CA 94041

Figure 31 - Package detail wireframes

The package detail page provides the user with an overview of a specific package, including its status and a detailed list of each of the tracking events for the package. Additionally, the page offers services to the user that include, buying insurance, rating the seller, printing a return label or paying any duties or fees for international shipments. We deemed the wireframes created for phase two sufficient for the minimal web application, but the mobile wireframes needed additional refinement.

4.3.3. Mobile Wireframes

In phase 3 we revisited the mobile wireframes with significantly greater attention to detail. We wanted to improve on the ideas from our phase 2 wireframes and establish a basis for application development. On a mobile device it was especially important that our wireframes looked native and made use of common mobile elements. By using common UI components such as menus, lists, and cards provided by the ionic framework we designed the wireframe shown in Figure 32.

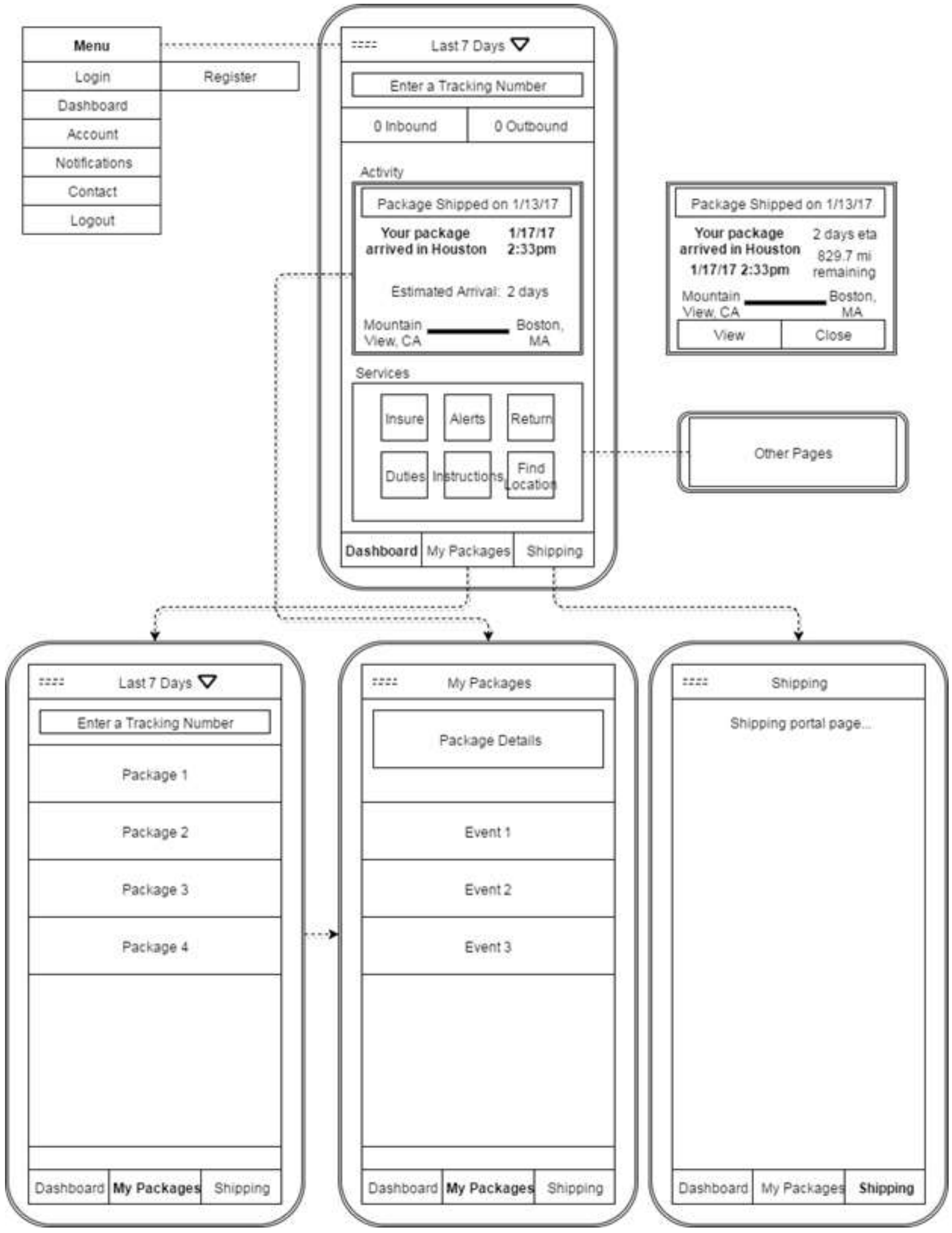


Figure 32 - First mobile wireframe iteration

The initial mobile wireframe shows the dashboard, package list, package detail page and an additional page for shipping as well as the navigation between these pages. On the mobile device the screen is narrow and there is limited space so it is important for navigation and content to be intuitive and accessible. To reflect this we adopted a consistent layout and offered many options for user routing. The menu in the top left provides navigation to a user's account and settings. The tab bar at the bottom provides navigation between the dashboard, the package list, the dashboard, and the shipping page. Other navigation events would occur based on user events such as a tap on a specific package or service button in the dashboard.

Figure 33 shows the second iteration of the mobile specific wireframe, which was the basis for initial development of the mobile application. For the second iteration we mainly rearranged components and changed text. We renamed the menu items to more common topics such as 'settings' in place of 'notifications' and 'help' in place of 'contact'. We also swapped the location of services and activity cards on the dashboard, to more eagerly present services to users.

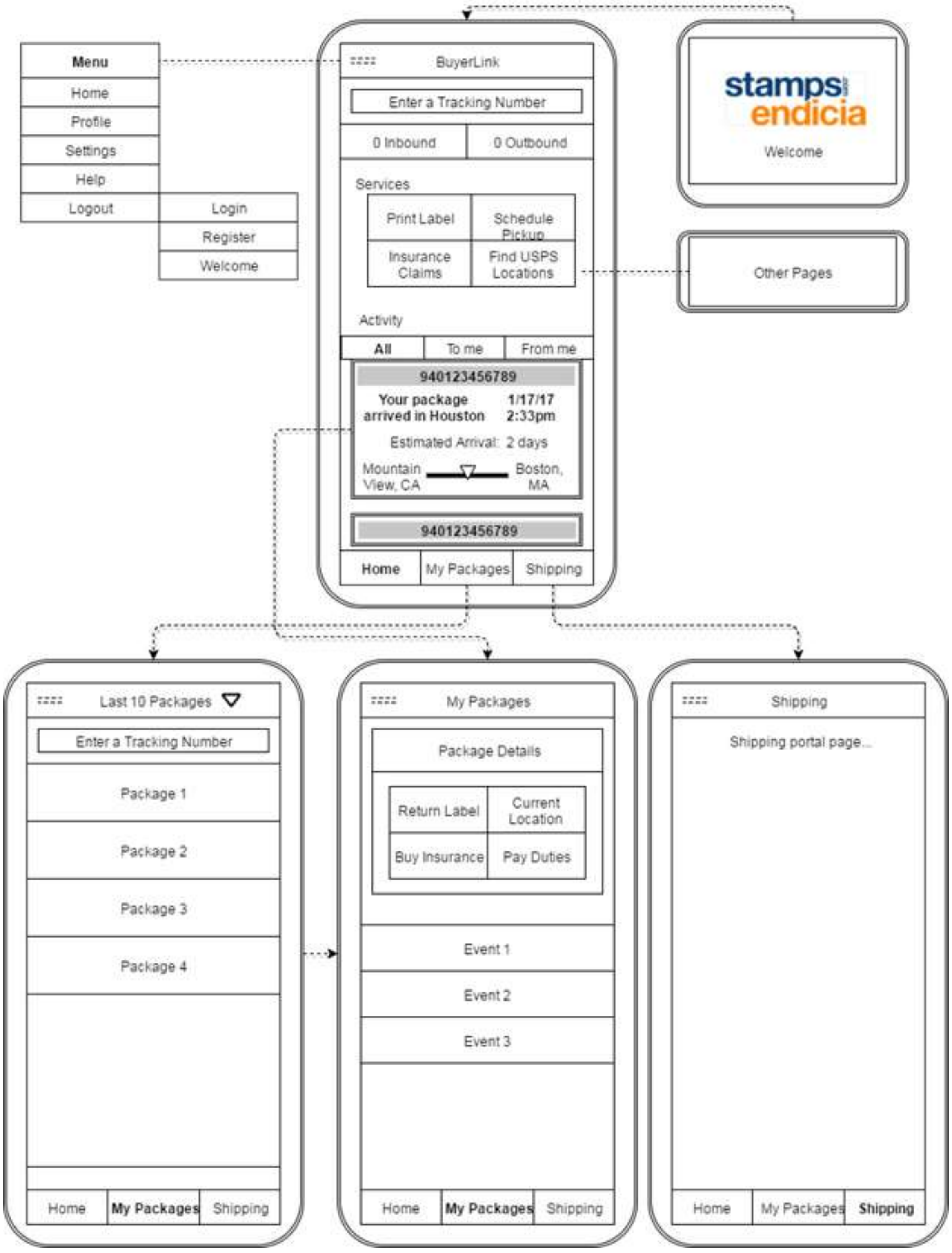
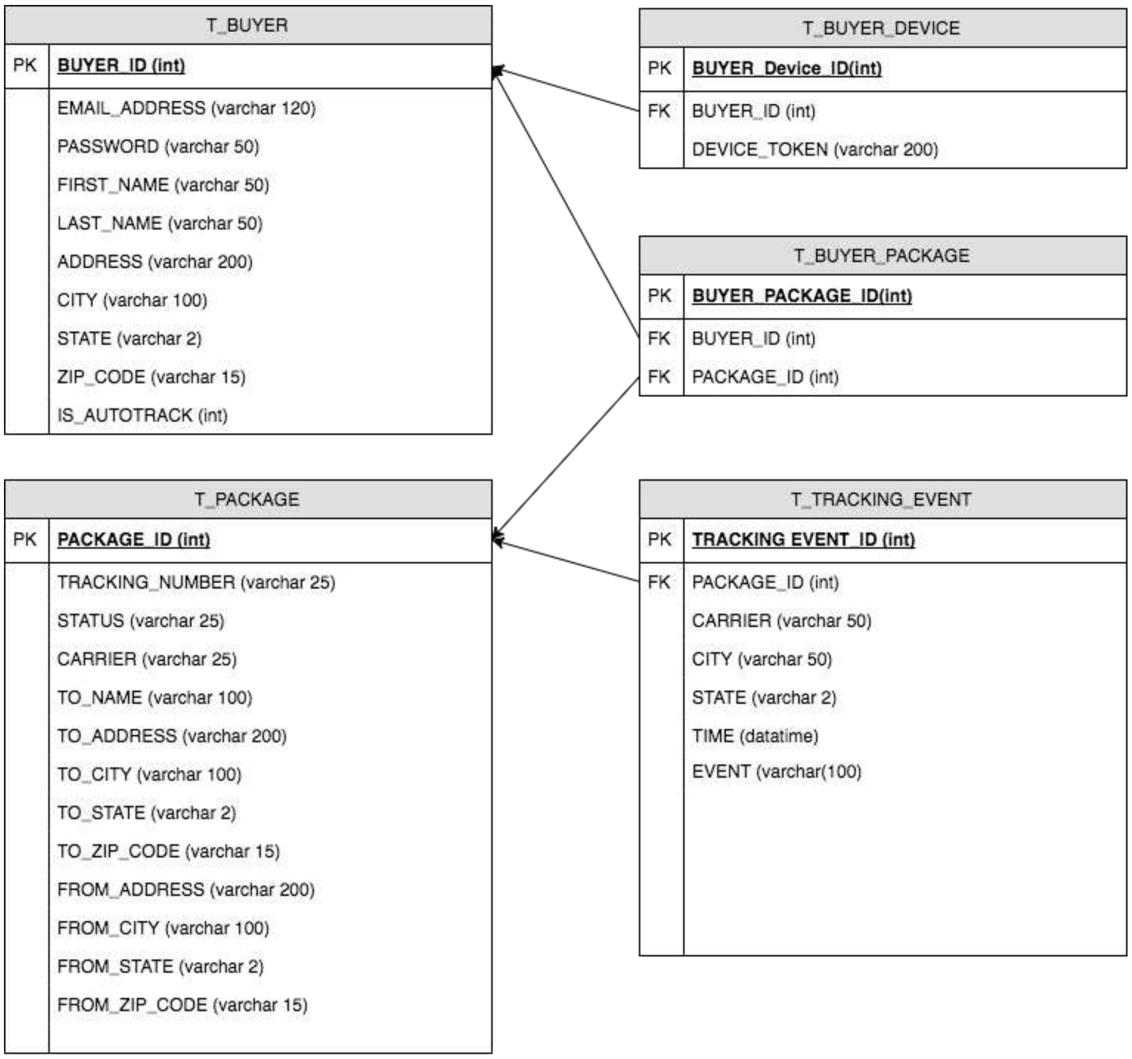


Figure 33 - Second mobile wireframe iteration

4.4. Database Schema

To store data that persists across devices and between user sessions the application requires a database. We created a database schema to ensure the database is well structured. Figure 34 shows the database schema, with each relation represented by a table.



T_BUYER	
PK	**BUYER_ID (int)**
	EMAIL_ADDRESS (varchar 120)
	PASSWORD (varchar 50)
	FIRST_NAME (varchar 50)
	LAST_NAME (varchar 50)
	ADDRESS (varchar 200)
	CITY (varchar 100)
	STATE (varchar 2)
	ZIP_CODE (varchar 15)
	IS_AUTOTRACK (int)

T_BUYER_DEVICE	
PK	**BUYER_Device_ID(int)**
FK	BUYER_ID (int)
	DEVICE_TOKEN (varchar 200)

T_BUYER_PACKAGE	
PK	**BUYER_PACKAGE_ID(int)**
FK	BUYER_ID (int)
FK	PACKAGE_ID (int)

T_PACKAGE	
PK	**PACKAGE_ID (int)**
	TRACKING_NUMBER (varchar 25)
	STATUS (varchar 25)
	CARRIER (varchar 25)
	TO_NAME (varchar 100)
	TO_ADDRESS (varchar 200)
	TO_CITY (varchar 100)
	TO_STATE (varchar 2)
	TO_ZIP_CODE (varchar 15)
	FROM_ADDRESS (varchar 200)
	FROM_CITY (varchar 100)
	FROM_STATE (varchar 2)
	FROM_ZIP_CODE (varchar 15)

T_TRACKING_EVENT	
PK	**TRACKING EVENT_ID (int)**
FK	PACKAGE_ID (int)
	CARRIER (varchar 50)
	CITY (varchar 50)
	STATE (varchar 2)
	TIME (datatime)
	EVENT (varchar(100)

Figure 34 - Showing column names and data types. Lines connect Foreign keys (FK) and Primary keys (PK) across tables.

We chose names for each relation that match the type of data they contain. T_BUYER holds information about an e-commerce buyer. Since e-commerce buyers are the intended users of the application, the T_BUYER relation contains fields such as email address and password which identify users. The T_BUYER_DEVICE relation links users to device tokens which target push notifications. This relation must be separate from T_BUYER since a single user can have multiple devices on which they want to receive notifications. T_BUYER_PACKAGE records which users are tracking which packages. A single user may track many packages and multiple users may track a single package. T_PACKAGE contains destination and source information for a package as well as tracking number and carrier info. T_TRACKING_EVENT contains information for tracking events. Each tracking even contains a foreign key to the package it belongs to.

5. Implementation

For development we used an adapted form of the Scrum software development approach. This approach consisted of daily team check-ins to review progress, outline goals for the day, and resolve any major blocking issues. Each week we met with the project sponsor Amine to readjust the project direction to better suit company needs and vision. By operating this way the team was able to move quickly and build the application by implementing features one at a time. Due to time constraints, we developed some features as front-end mock-ups to show possible features and services without fully implementing backend support for the feature. Table 2 lists the set of implemented features in the mobile prototype and web prototype.

Table 2 - A full set of features implemented during development.

Mobile Features
Login
Registration
Dashboard
Package List
Package Details
Package Map
AutoTrack
Push Notifications
Account Details
DeepLinking
Mobile Frontend Features
Insurance Claim
Notification Settings
Buy Insurance
Find Nearby Locations
Print Label (and Return Label)
Report Issue
Rate Seller
Web Prototype
Marketing Page
Login
Registration
Package Detail
Package List

5.1. Mobile Application

We developed our mobile application in Ionic 2, using Cordova plugins to access native device features. We built each page of the mobile application with a corresponding Angular 2 component and HTML template comprised mainly of Ionic 2 UI components. Page components inject services which handle forming and receiving HTTP requests to the Web API, and to Firebase for authentication. The mobile application has both fully implemented features, and front end implemented features. Fully implemented features include calls to the backend or to an external API, whereas front-end features are exclusively designed as a UI to showcase possible services and features that could exist in a production application.

5.1.1. Full Stack Features

This section covers features that we implemented across the full stack, or features that only require front end implementation in order to be complete features. These full stack features represent the core functionalities that we aimed to implement. For each of these features we will discuss the client facing page and the code that enables it to function. We did not include all code written for the project in this section, instead, we have selected key segments of code that explain how a feature works.

Login (Mobile)

Figure 35 shows the login page of the application, produced by the template login.page.html, here we can see buttons for social logins through Facebook, Twitter and Google. In a test environment Facebook logins work, however, developers must acquire an application key for the login to work on an actual device. The Twitter and Google logins are in place as a concept and as a suggestion for future implementation. The login screen also includes email and password fields for a user to input credentials, as well as a login button to send those credentials. Lastly the page includes a button which can direct users to the registration page.

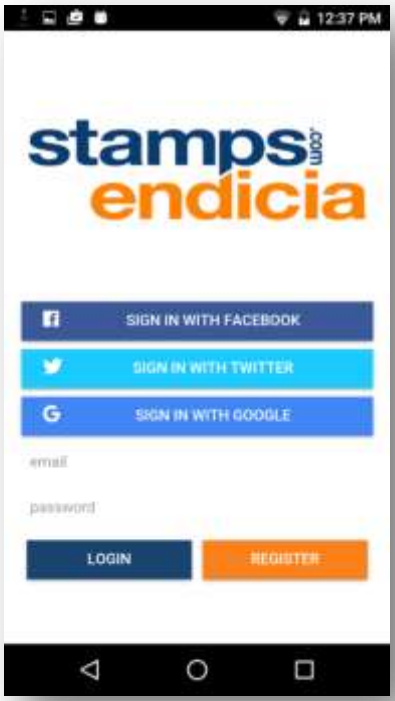
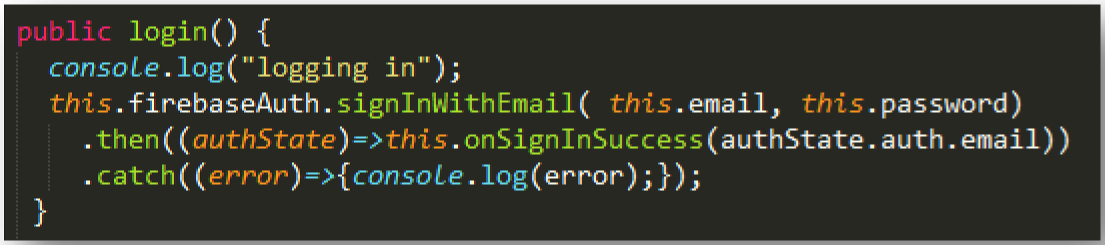


Figure 35 – Mobile Login Page

Figure 36 shows the code for the login method which the system calls when a user presses the login button. In this method, the system passes the user's credentials through a FirebaseAuthentication object, which sends the credentials in a request to the firebase server. The signInWithEmail method returns a promise of an authState, which a developer can use to complete the login process on client side by calling onSignInSuccess. The onSignInSuccess method stores the user in the application and sets the home page as the root navigation page. After a successful login, the user credentials and account are additionally verified in our ASP.net API and database.



```
public login() {
  console.log("logging in");
  this.firebaseAuth.signInWithEmail( this.email, this.password)
    .then((authState)=>this.onSignInSuccess(authState.auth.email))
    .catch((error)=>{console.log(error);});
 }
```

Figure 36 - Login Code Snippet

Registration (Mobile)

The registration page asks for four pieces of user information in order to keep the process lightweight and encourage user signup. Figure 37 shows the registration page composed of four input fields and a button to complete registration.

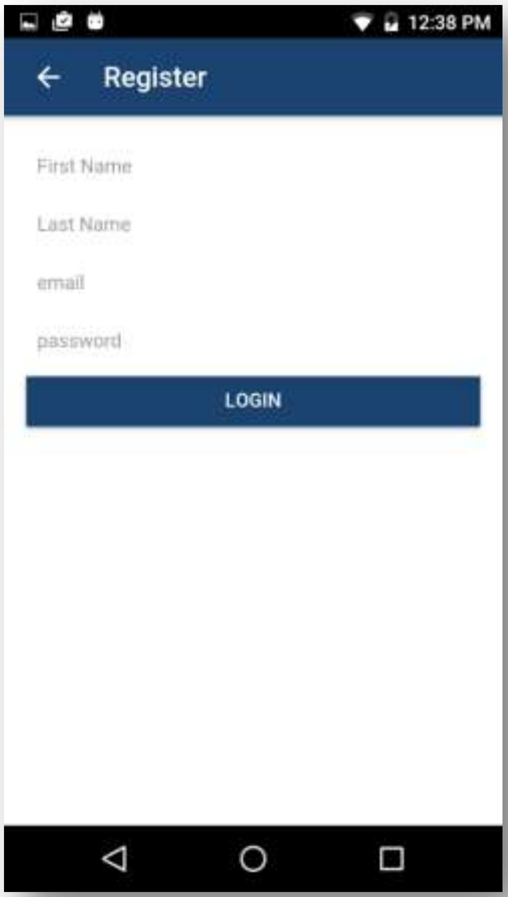
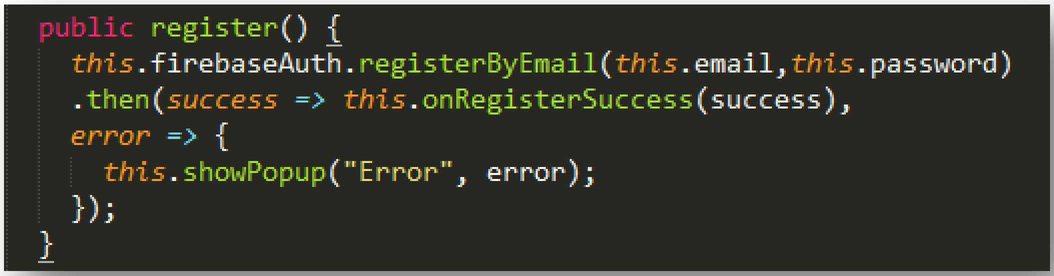


Figure 37 - Mobile Registration page

Figure 38 shows the register method, which gets called when the user submits their registration information. This method is very similar to the login process, however the system calls the registerByEmail method of the FirebaseAuthentication object instead of the signInWithEmail method. The registration process gives users feedback by presenting a popup if registration encounters an error.



```
public register() {
  this.firebaseAuth.registerByEmail(this.email,this.password)
  .then(success => this.onRegisterSuccess(success),
  error => {
    this.showPopup("Error", error);
  });
}
```

Figure 38 - Registration Code Snippet

Dashboard

The system presents the dashboard page to the user, as shown in Figure 39, immediately after logging in. This is the user's home page, providing a broad overview of all the users' packages in the set of cards near the top of the screen. These cards show the number of delivered, on route, and alert packages, packages which may have some issue with delivery. The fourth card, outbound, we reserved for packages the user ships. The dashboard also allows users to enter a tracking number in the top to begin tracking a package. In the lower section of the screen, the display presents a list of services to the user which include printing a label, filing an insurance claim or viewing nearby USPS locations. The navigation bar at the bottom of the dashboard gives users tab options to the package list or to the shipping portal.

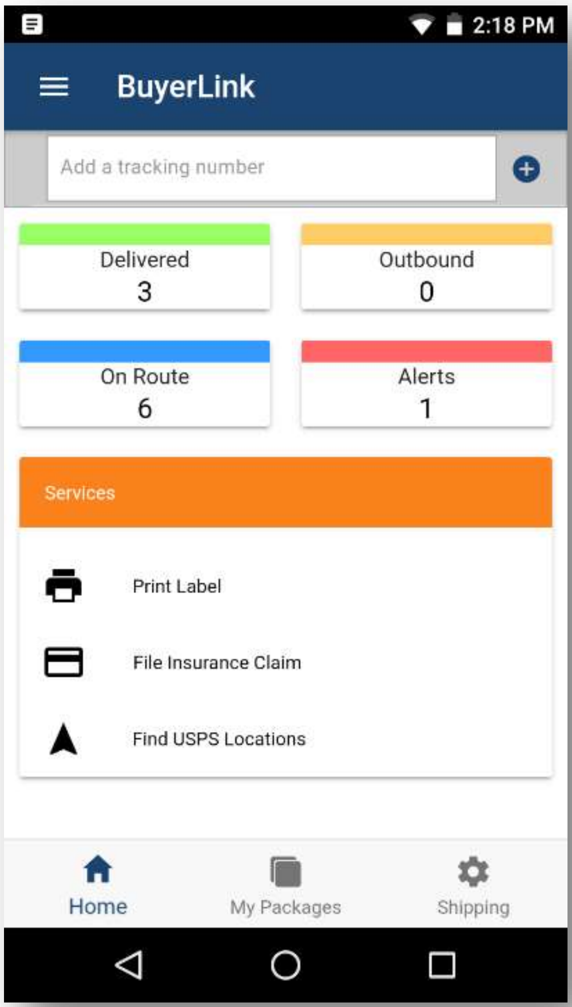
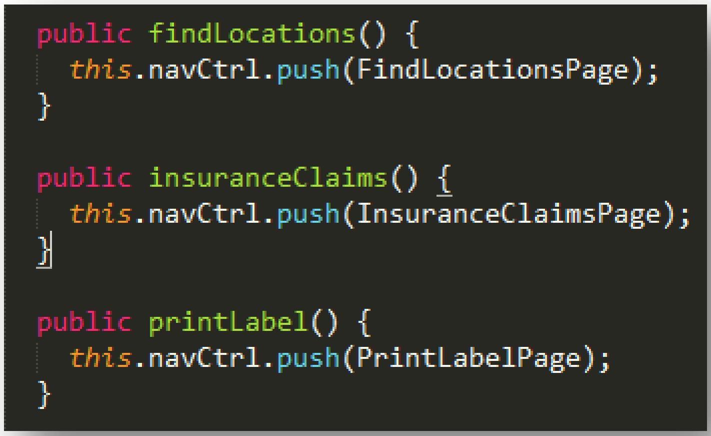


Figure 39 - Mobile Dashboard page

We developed the page with limited functionality, and instead it mostly serves as a navigation hub where users can access different services and pages. Figure 40 shows how the dashboard pushes other pages onto the navigation stack. These functions use the Ionic Frameworks NavigationController to push a component onto the stack. For a developer to push components, they must import the NavigationController into the dashboard component.



```
public findLocations() {
  this.navCtrl.push(FindLocationsPage);
}

public insuranceClaims() {
  this.navCtrl.push(InsuranceClaimsPage);
}

public printLabel() {
  this.navCtrl.push(PrintLabelPage);
}
```

Figure 40 - Code snippet from Dashboard.ts showing how the developer pushes pages onto the navigation stack

Package List

The main page for viewing packages is the Package List page as shown in Figure 41. The left display shows each package with its status, carrier, logo of a seller and the timestamp of the most recent tracking event associated with the package. We can see each type of package which includes delivered packages, on route packages, and a canceled package. In the figure on the right, the display shows the delivered segment selected at the top of the page, and the page only lists delivered packages.

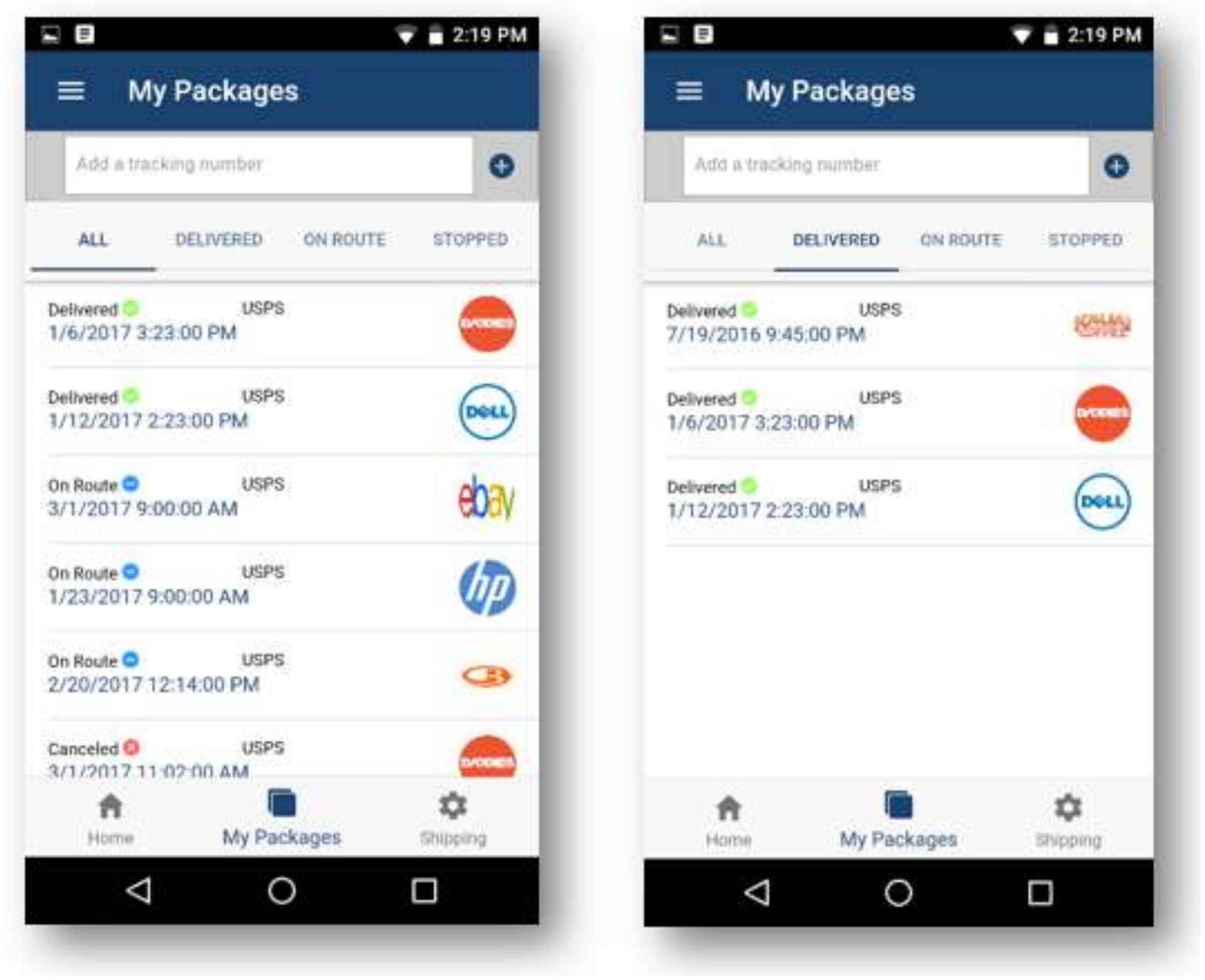
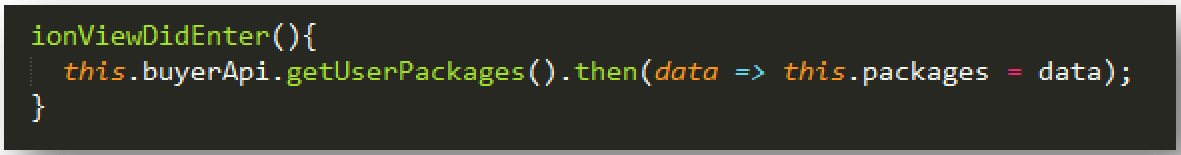

2:19 PM
My Packages
Add a tracking number
ALL
DELIVERED
ON ROUTE
STOPPED
Delivered USPS 1/6/2017 3:23:00 PM
Delivered USPS 1/12/2017 2:23:00 PM
On Route USPS 3/1/2017 9:00:00 AM
On Route USPS 1/23/2017 9:00:00 AM
On Route USPS 2/20/2017 12:14:00 PM
Canceled USPS 3/1/2017 11:02:00 AM
Home
My Packages
Shipping
2:19 PM
My Packages
Add a tracking number
ALL
DELIVERED
ON ROUTE
STOPPED
Delivered USPS 7/19/2016 9:45:00 PM
Delivered USPS 1/6/2017 3:23:00 PM
Delivered USPS 1/12/2017 2:23:00 PM
Home
My Packages
Shipping

Figure 41 – Mobile Package List page: The unfiltered package page (left) and filtered package list (right) showing only delivered items

To display packages, the client app must request the user's packages from the server when the page loads. To do this, the app makes use of "Component lifecycle events", which are functions that get called when certain events occur, like when the user enters or leaves a page.

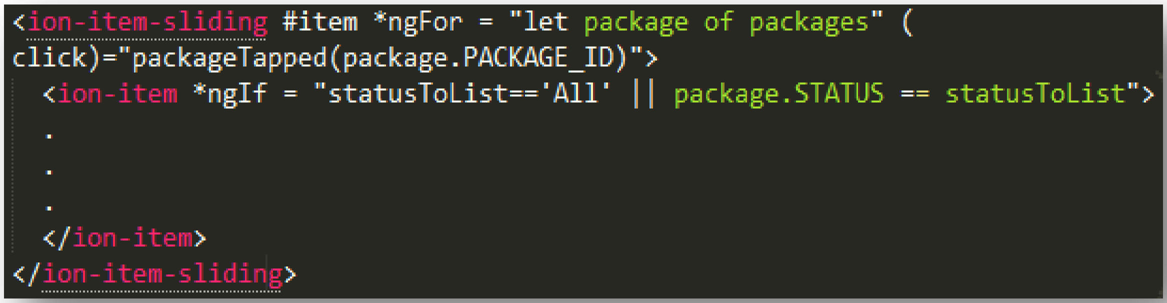
Figure 42 shows the ionViewDidEnter function, which is automatically called when the user enters the view. Here, the ionViewDidEnter function uses a reference to the Buyer Api service to get the user packages. The Buyer Api Service includes functions that send HTTP requests to our backend Web API. This service returns a Promise of package data and when the Promise resolves, the packages property of the package-list component is set to the data retrieved from the API.



```
ionViewDidEnter(){
  this.buyerApi.getUserPackages().then(data => this.packages = data);
}
```

Figure 42 - Lifecycle event ionViewDidEnter populating the package list by requesting data from the Web Api through the buyerApi service.

To display these packages, the HTML template uses the angular directive *ngFor to display items for each package as shown in Figure 43. This iteratively creates an ion item for each package of the packages array from the package-list component. Additionally, the system only creates items for the packages that match the current filter, which we expressed by the *ngIf statement. In the *ngIf, if the current filter is all packages, the system makes all packages items, but if statusToList changes, the display only shows those packages which match the new statusToList.

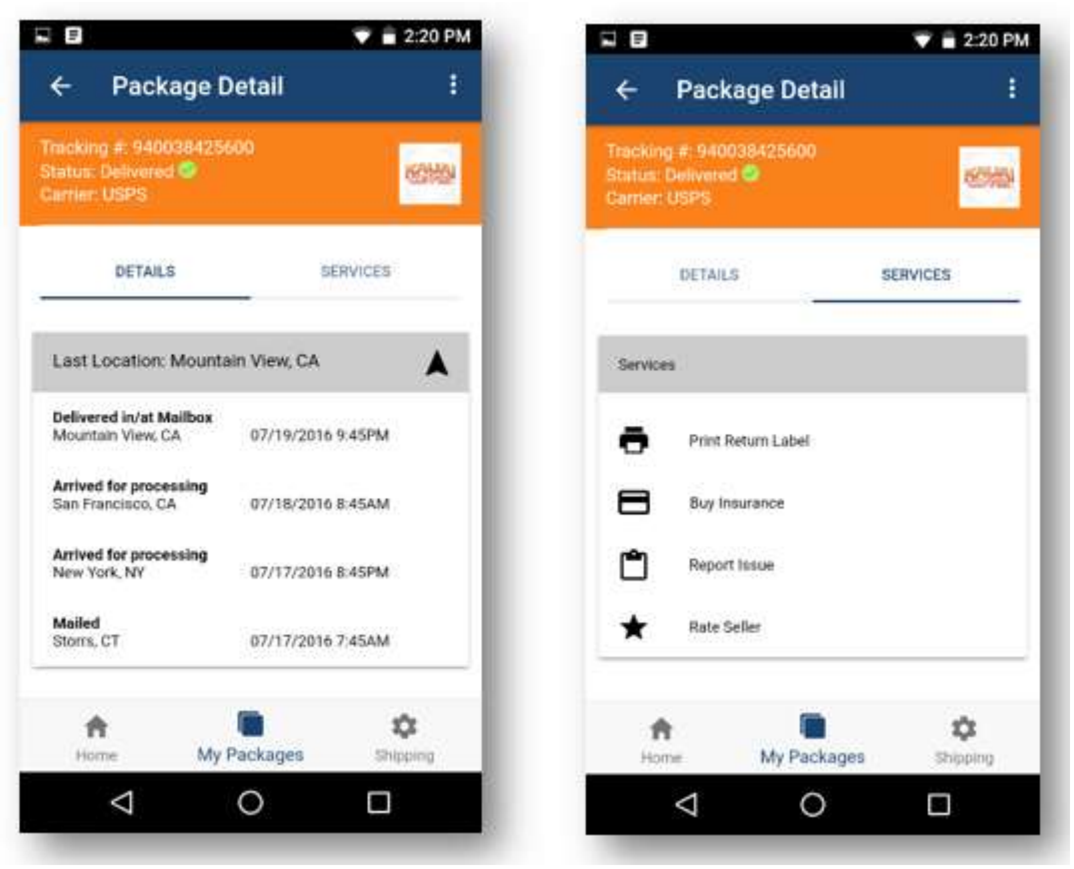


```
<ion-item-sliding #item *ngFor = "let package of packages" (
click)="packageTapped(package.PACKAGE_ID)">
  <ion-item *ngIf = "statusToList=='All' || package.STATUS == statusToList">
  .
  .
  .
  </ion-item>
</ion-item-sliding>
```

Figure 43 - HTML in package-list.page.html with some code omitted for readability.

Package Details

The Package details page displays information about package tracking events and presents the user with services to enact on that specific package. In Figure 44, on the left, the Package details page presents tracking information in a list showing the most recent event at the top of the list. On the page to the right, it shows the list of services offered for a given package, including printing a return label, buying insurance, reporting an issue or rating the seller of the package. When a user enters the package detail page from the package list page, the system passes the package to the PackageDetail component as a navigation parameter. Navigation parameters allow developers to pass data between pages during navigation. Like the Dashboard page, the PackageDetail page acts mainly as a source of information for the user and as a navigation hub to additional service and features.



Tracking #: 940038425600
Status: Delivered
Carrier: USPS

Last Location: Mountain View, CA

Delivered in/at Mailbox
Mountain View, CA 07/19/2016 9:45PM

Arrived for processing
San Francisco, CA 07/18/2016 8:45AM

Arrived for processing
New York, NY 07/17/2016 8:45PM

Mailed
Storrs, CT 07/17/2016 7:45AM

Services

Print Return Label

Buy Insurance

Report Issue

Rate Seller

Figure 44 – Mobile Package Detail page: package status and tracking events (left) and package specific services offered (right)

Package Map

The PackageMap page, as shown in Figure 45, shows the user a map to illustrate both the current and destination locations of the package. This page uses a combination of native features and external APIs to provide this feature.

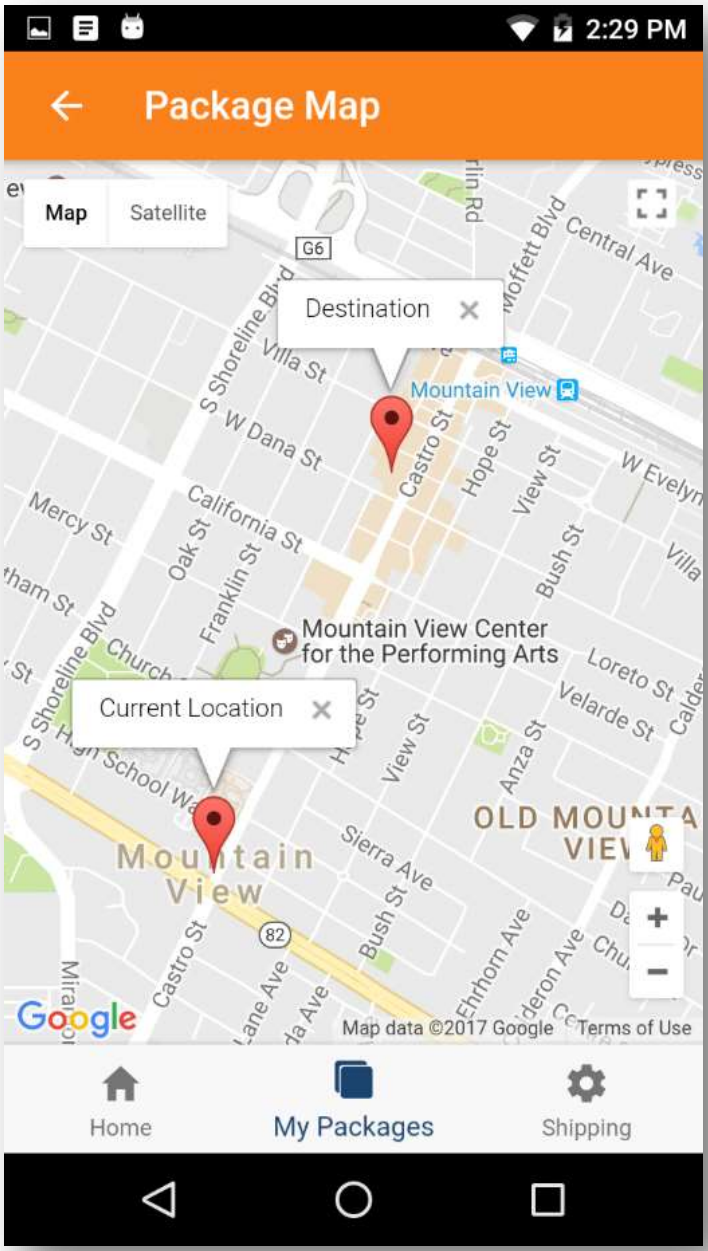
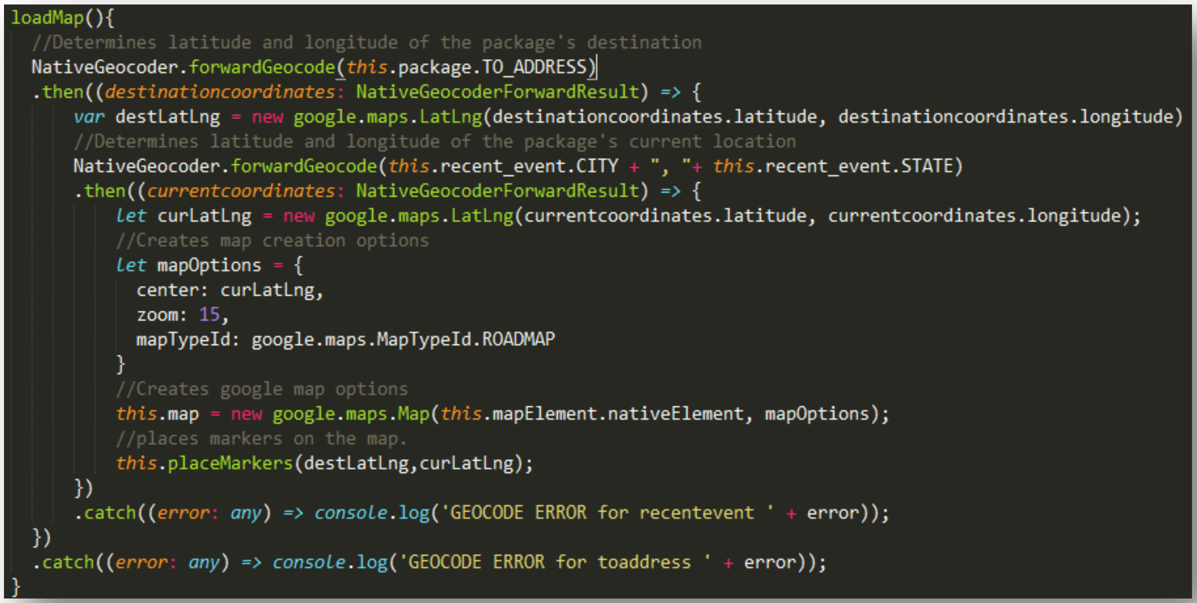


Figure 45 – Mobile Map page: shows package current location and destination

Figure 46 shows how the native Geocoding feature first produces the markers latitudes and longitudes, then the system initializes the map using a Google maps API. Lastly, we added markers to the map at the latitudes and longitudes of the current and destination addresses. One limitation of this feature is the precision of the current location, since tracking events only include the city of the location, the marker is not placed directly on the facility where the package is. An improvement to this feature might include providing data in tracking events to accurately locate the package process facilities a package travels through.

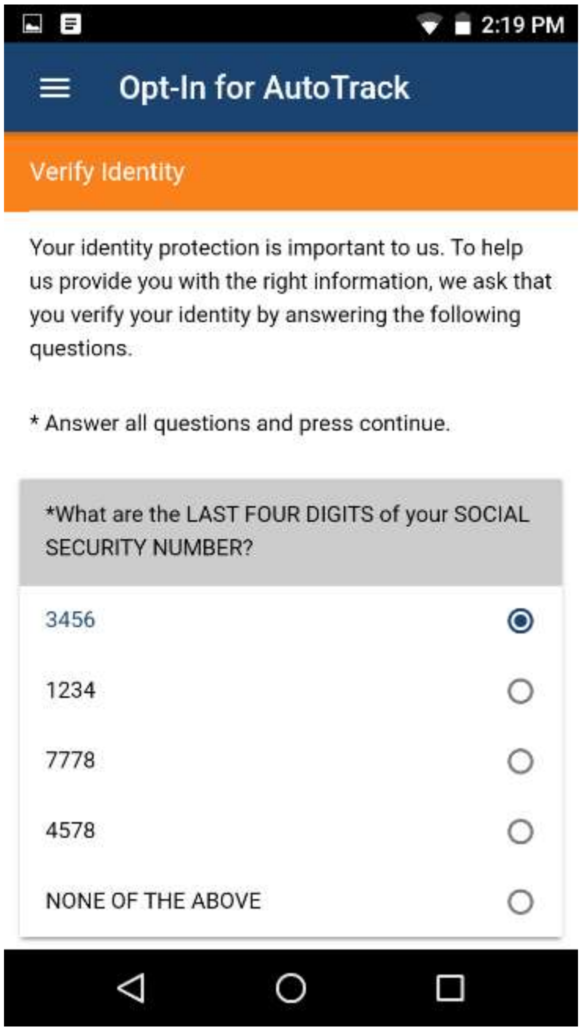


```
loadMap(){
  //Determines latitude and longitude of the package's destination
  NativeGeocoder.forwardGeocode(this.package.TO_ADDRESS)
  .then((destinationcoordinates: NativeGeocoderForwardResult) => {
      var destLatLng = new google.maps.LatLng(destinationcoordinates.latitude, destinationcoordinates.longitude)
      //Determines latitude and longitude of the package's current location
      NativeGeocoder.forwardGeocode(this.recent_event.CITY + ", "+ this.recent_event.STATE)
      .then((currentcoordinates: NativeGeocoderForwardResult) => {
          let curLatLng = new google.maps.LatLng(currentcoordinates.latitude, currentcoordinates.longitude);
          //Creates map creation options
          let mapOptions = {
            center: curLatLng,
            zoom: 15,
            mapTypeId: google.maps.MapTypeId.ROADMAP
          }
          //Creates google map options
          this.map = new google.maps.Map(this.mapElement.nativeElement, mapOptions);
          //places markers on the map.
          this.placeMarkers(destLatLng,curLatLng);
      })
      .catch((error: any) => console.log('GEOCODE ERROR for recentevent ' + error));
  })
  .catch((error: any) => console.log('GEOCODE ERROR for toaddress ' + error));
}
```

Figure 46 - The load map method preparing the map for display

Auto Track

AutoTrack is one of the most significant features we implemented. This feature was mostly implemented in the backend, but required a front end page to enable it for a given user. With AutoTrack enabled, a user automatically tracks a package that enters the backend of our system if the package address and name matches the user’s information. If a developer implemented this feature without thought to security, a malicious user could take advantage of this feature to receive information about another person’s packages. To avoid this, we included a mock up of an identity verification system. There is an industry standard for identity verification, and there are API’s which can provide questions about an individual to which only the individual would know the answers. These questions, known as “Out of wallet” questions, often ask about topics such as the user’s social security number, recent places the user has lived, names of family members or recent phone numbers. Figure 47 shows an example of this question.



2:19 PM

Opt-In for AutoTrack

Verify Identity

Your identity protection is important to us. To help us provide you with the right information, we ask that you verify your identity by answering the following questions.

* Answer all questions and press continue.

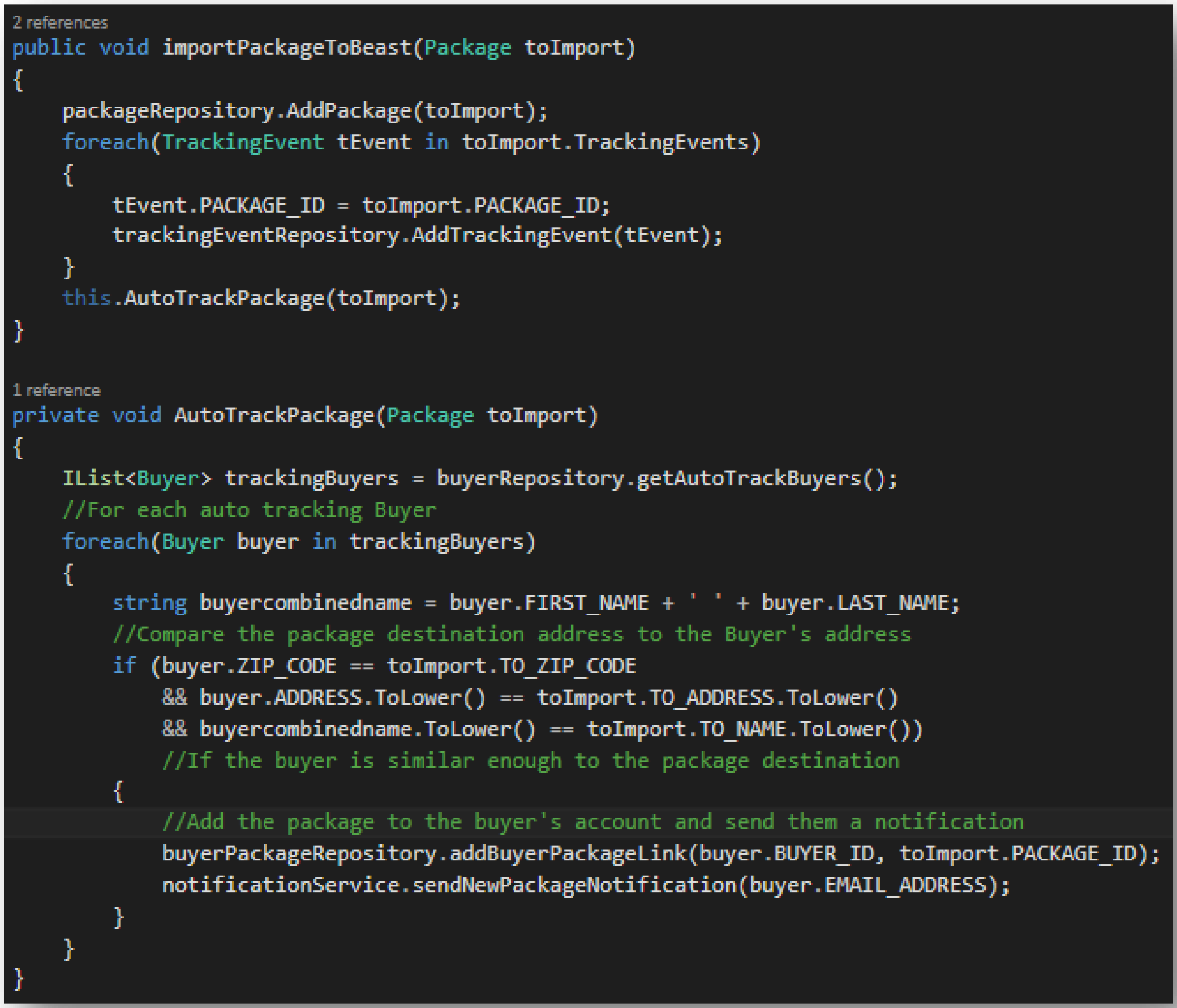
*What are the LAST FOUR DIGITS of your SOCIAL SECURITY NUMBER?

[✓ 3456]
1234
7778
4578
NONE OF THE ABOVE

Figure 47 – Mobile Opt-In for AutoTrack page with sample identity verification questions

Once a system verifies the user's identity, we consider them opted-in for Auto Track. In the Beast database, the system additionally confirms them as a verified user. When new packages enter the backend API and into Beast, they are then AutoTracked as shown in Figure 48. When the system AutoTracks a package, it compares the package to all users that are currently opted in for auto track. The system then adds the package to any user's account with a matching address.

Currently, the system only adds packages to a user's account if the name, zip code, and address are an exact match, ignoring the case of the text. In a future implementation, this algorithm could be more tolerant of variations like nicknames or mistyped addresses. Once the system links the buyer and a package, the system sends a notification to the user.

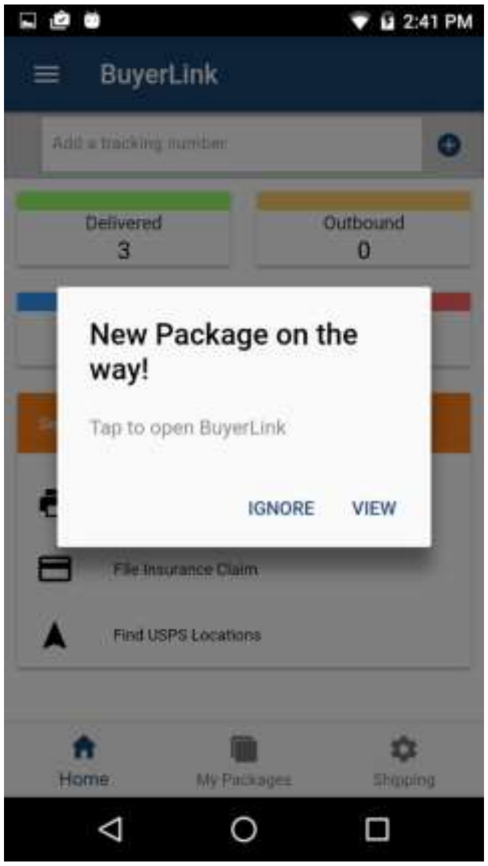


```
2 references
public void importPackageToBeast(Package toImport)
{
    packageRepository.AddPackage(toImport);
    foreach(TrackingEvent tEvent in toImport.TrackingEvents)
    {
        tEvent.PACKAGE_ID = toImport.PACKAGE_ID;
        trackingEventRepository.AddTrackingEvent(tEvent);
    }
    this.AutoTrackPackage(toImport);
}

1 reference
private void AutoTrackPackage(Package toImport)
{
    IList<Buyer> trackingBuyers = buyerRepository.getAutoTrackBuyers();
    //For each auto tracking Buyer
    foreach(Buyer buyer in trackingBuyers)
    {
        string buyercombinedname = buyer.FIRST_NAME + ' ' + buyer.LAST_NAME;
        //Compare the package destination address to the Buyer's address
        if (buyer.ZIP_CODE == toImport.TO_ZIP_CODE
            && buyer.ADDRESS.ToLower() == toImport.TO_ADDRESS.ToLower()
            && buyercombinedname.ToLower() == toImport.TO_NAME.ToLower())
            //If the buyer is similar enough to the package destination
        {
            //Add the package to the buyer's account and send them a notification
            buyerPackageRepository.addBuyerPackageLink(buyer.BUYER_ID, toImport.PACKAGE_ID);
            notificationService.sendNewPackageNotification(buyer.EMAIL_ADDRESS);
        }
    }
}
```

Figure 48 - The importPackageToBeast() and AutoTrackPackage() functions in backend Web API

Push Notifications

Push notifications are a valuable feature for informing users about new packages and tracking events that may interest them. For this project we explore push notifications as a proof of concept, and only implemented notification for new packages added to a user's account via AutoTrack. Figure 49 shows what the user sees when a new push notification arrives. While the client facing component of this feature is fairly simple, the backend to support this feature is in no way trivial. In order for this notification to appear, a user takes a number of steps.



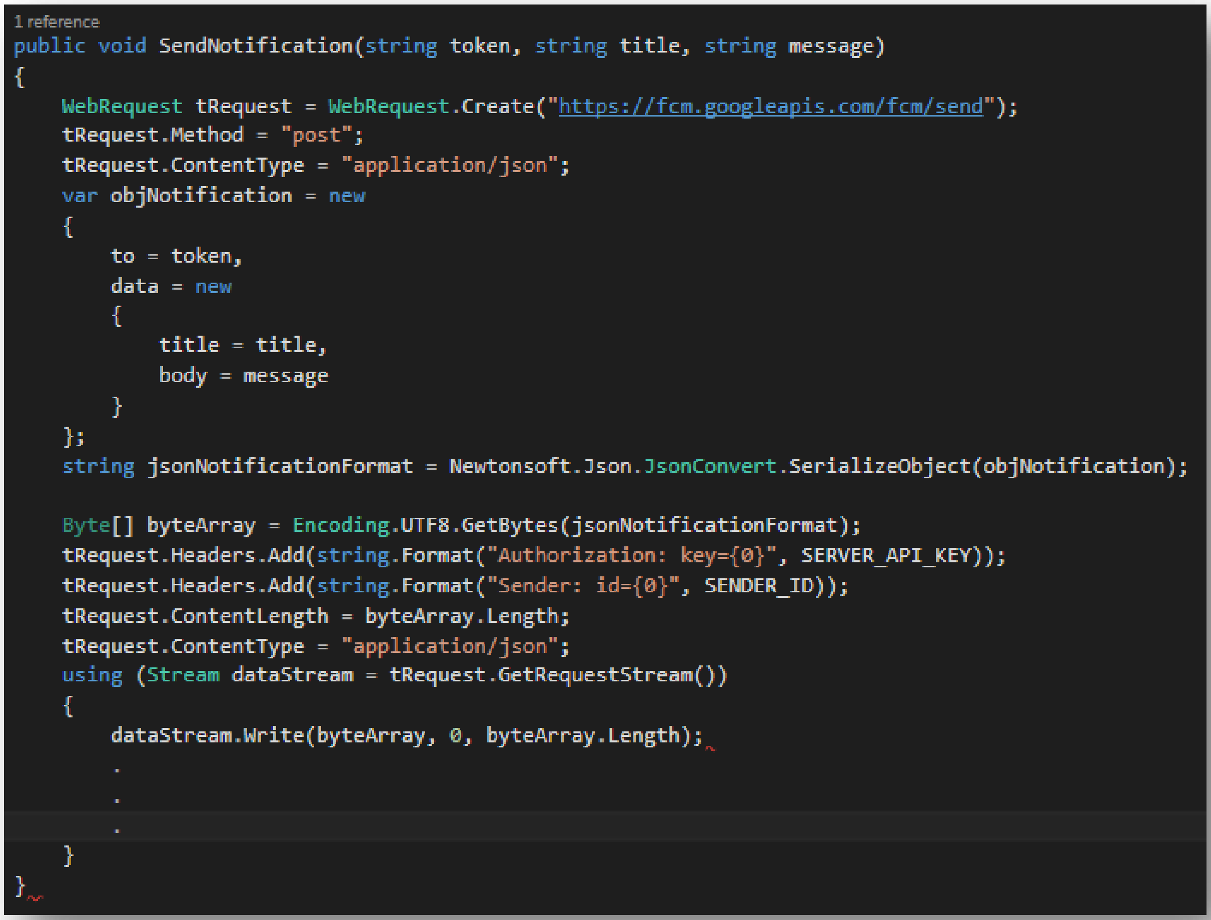
New Package on the way!

Tap to open BuyerLink

IGNORE VIEW

Figure 49 - Push notification on new package tracked via AutoTrack.

First the user’s application sends an HTTP post containing a device registration token to the Web API, which stores that token in the database. When the system adds a new package to a user via AutoTrack, the system must additionally send a push notification to the user. A function in the notification service of the web API prepares a request and sends it to the Firebase Cloud Messaging (FCM) server. Figure 50 shows the function which prepares a push notification with a title, message and user’s device registration token, and sends that notification to FCM. The FCM server acts as an intermediary for notifications between our API and users’ devices. FCM implements complex logic for sending push notifications to registered devices, ensuring that the notification eventually arrives on the user’s device, regardless of whether the user has their device turned off or not connected to the internet for a period of time.

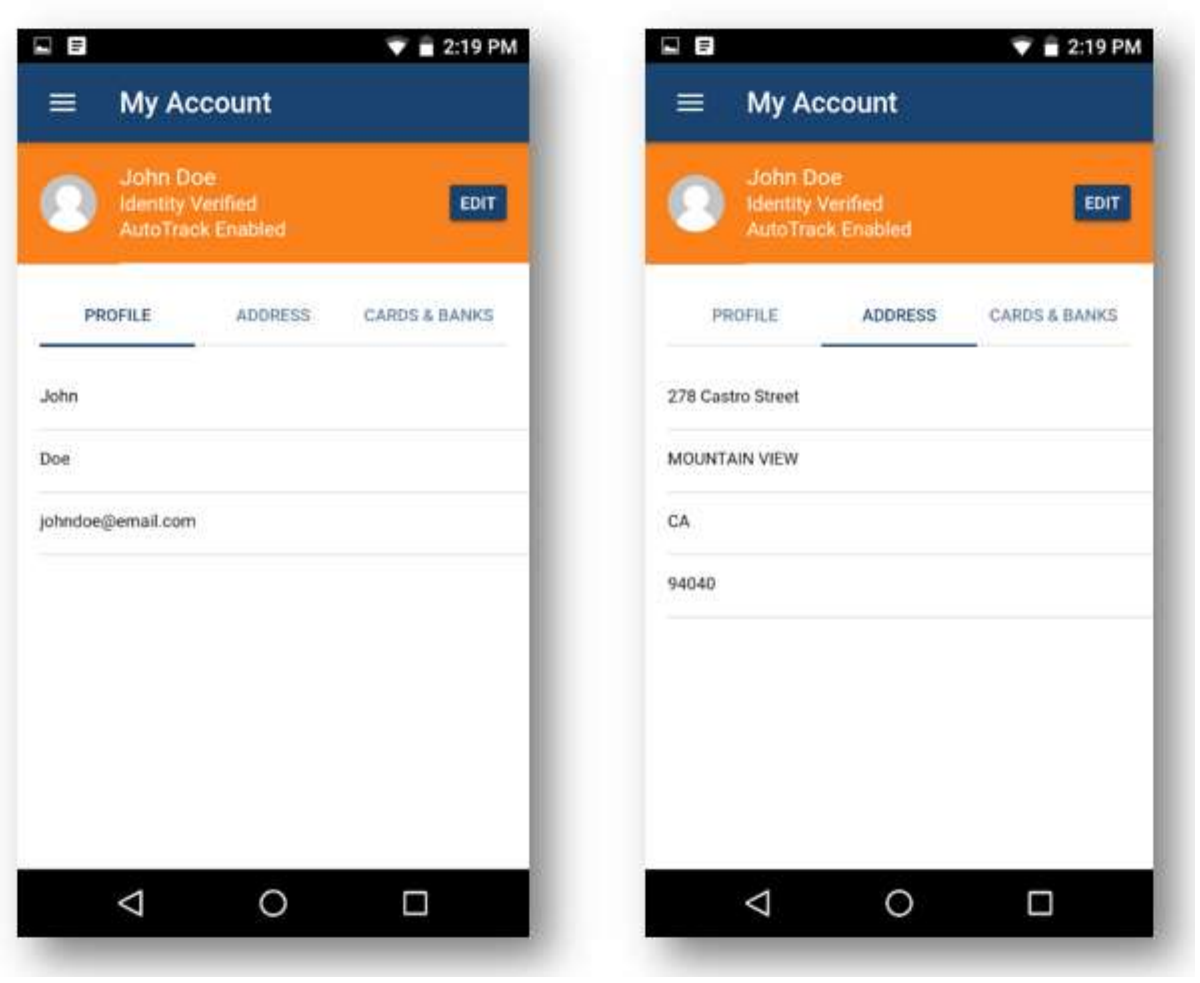


```
1 reference
public void SendNotification(string token, string title, string message)
{
    WebRequest tRequest = WebRequest.Create("https://fcm.googleapis.com/fcm/send");
    tRequest.Method = "post";
    tRequest.ContentType = "application/json";
    var objNotification = new
    {
        to = token,
        data = new
        {
            title = title,
            body = message
        }
    };
    string jsonNotificationFormat = Newtonsoft.Json.JsonConvert.SerializeObject(objNotification);

    Byte[] byteArray = Encoding.UTF8.GetBytes(jsonNotificationFormat);
    tRequest.Headers.Add(string.Format("Authorization: key={0}", SERVER_API_KEY));
    tRequest.Headers.Add(string.Format("Sender: id={0}", SENDER_ID));
    tRequest.ContentLength = byteArray.Length;
    tRequest.ContentType = "application/json";
    using (Stream dataStream = tRequest.GetRequestStream())
    {
        dataStream.Write(byteArray, 0, byteArray.Length);
        .
        .
        .
    }
}
```

Figure 50 – SendNotification() function consuming a device registration token, title and message to send a push notification to FCM. FCM response handling code omitted for readability

Account Details

While establishing a link between the buyer and Endicia, one important area was creating an account system. To do this we included a profile page in our application. Currently the profile page includes the user’s name, email, and address information, as well as a mocked up page for payment information. Future profile pages could support more customization to further solidify the link between the buyer and the application. Figure 51 illustrates the Account page.



John

Doe

johndoe@email.com

278 Castro Street

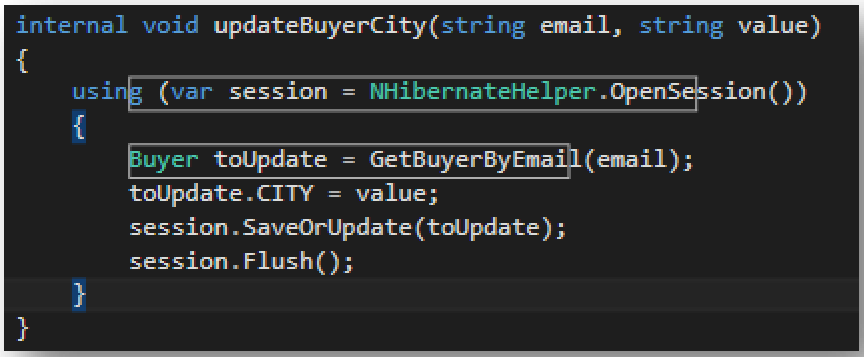
MOUNTAIN VIEW

CA

94040

Figure 51 – Mobile Account page on profile segment (left) and Account page on address segment (right)

All of these features are editable, allowing users to select edit, and then modify fields. This page takes advantage of Angular 2's two way binding, so that when a user makes changes in the input, the system simultaneously makes them in the model of the user. When the user finishes editing and hits save (not pictured), the system sends updates to the backend so that the database can reflect these changes. To send this information, the ProfilePage uses our BuyerAPI service to send an HTTP Put request with the updated user information to the Web API. Once the API receives the Put request, the system processes the request, then hands it to the Buyer repository to adjust the value of the buyer in the database. Figure 52 shows an example of the function that changes the City for a buyer with a given email.



```
internal void updateBuyerCity(string email, string value)
{
    using (var session = NHibernateHelper.OpenSession())
    {
        Buyer toUpdate = GetBuyerByEmail(email);
        toUpdate.CITY = value;
        session.SaveOrUpdate(toUpdate);
        session.Flush();
    }
}
```

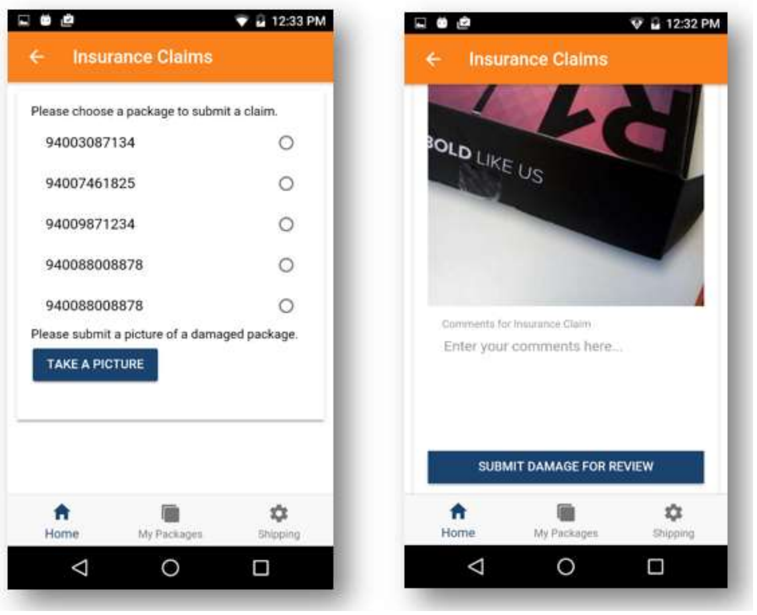
Figure 52 - The updateBuyerFunction() in the Buyer Repository getting the user by email, updating the city value, and saving the changes

5.1.2. Front End Features

The front-end features include pages that we did not fully implement yet. The purpose of making these was to provide a complete user experience for demonstrations. By utilizing native device features such as the camera and geolocation in addition to auto-populating data we found that we could successfully provide an idea of what these services might look like.

Insurance Claim

The insurance-claim page, as shown below in Figure 53, allows users to file an insurance claim.



Please choose a package to submit a claim.

94003087134

94007461825

94009871234

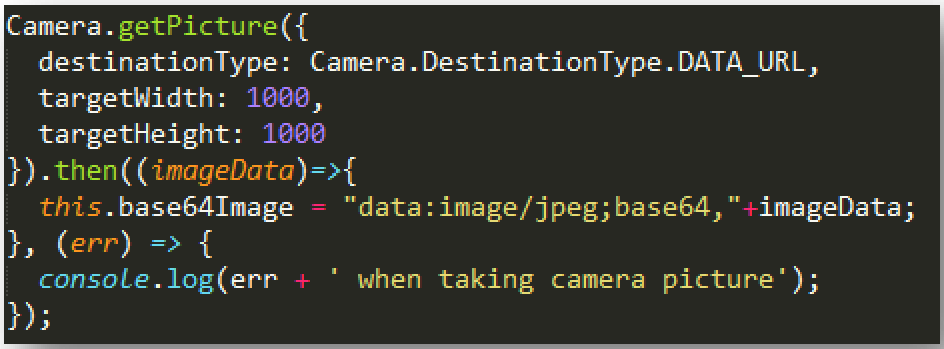
940088008878

940088008878

Please submit a picture of a damaged package.

Figure 53 – Mobile Insurance Claim page, choosing a package for claim submission (left) and an image taken and comment box (right)

In this feature we access the Camera through Ionic Native. The code to add the camera does not require developers to write native code and instead developers can create in TypeScript as shown in Figure 54.



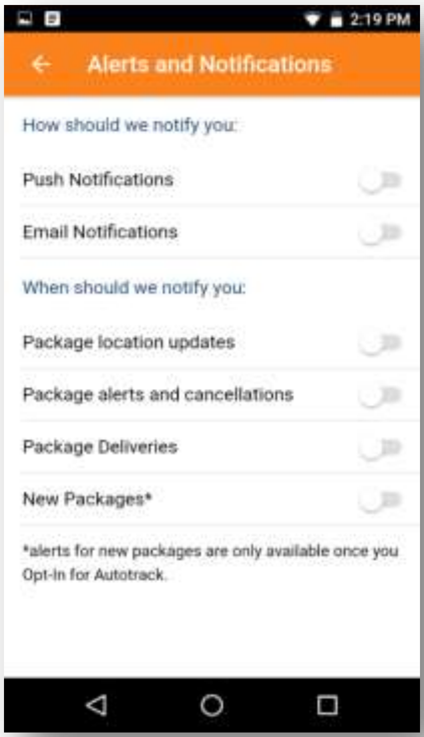
```
Camera.getPicture({
  destinationType: Camera.DestinationType.DATA_URL,
  targetWidth: 1000,
  targetHeight: 1000
}).then((imageData)=>{
  this.base64Image = "data:image/jpeg;base64,"+imageData;
}, (err) => {
  console.log(err + ' when taking camera picture');
});
```

Figure 54 - Code snippet of Ionic Native Camera plugin

To take a picture, the developer calls the Camera plugin's getPicture({}) member. The getPicture({}) method allows developers to customize the quality, set the encoding, and returns a promise a developer can act on. The development of this feature is proof that developers can add native device features that easily enhance the user experience without sacrificing much time or effort.

Notification Settings

The notifications page, as shown if Figure 55, is navigable from the side menu settings. This provides an idea of what it might look like for a user to change their notifications.



How should we notify you:

Push Notifications

Email Notifications

When should we notify you:

Package location updates

Package alerts and cancellations

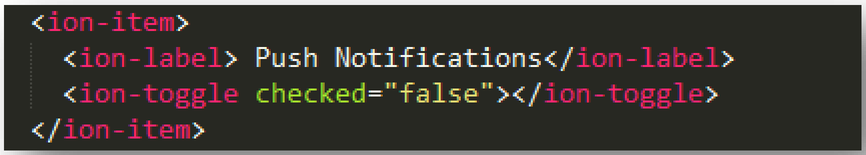
Package Deliveries

New Packages*

*alerts for new packages are only available once you Opt-In for Autotrack.

Figure 55 – Mobile Notification Settings page

In the figure we imagined that users would like to set preferences for notifications they receive. To implement this view we used a list containing items with ion-toggles as shown in Figure 56.

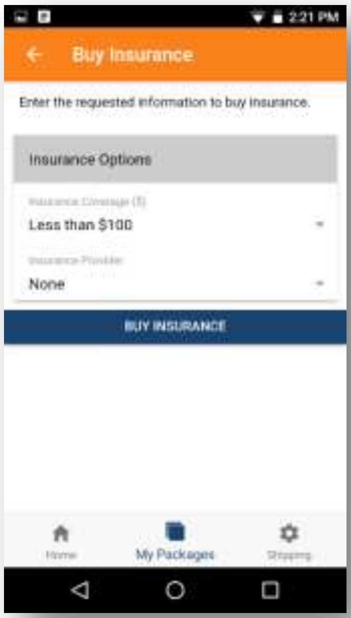


```html
<ion-item>
  <ion-label> Push Notifications</ion-label>
  <ion-toggle checked="false"></ion-toggle>
</ion-item>
```

Figure 56 - Example of implementing a toggle button in HTML

Buy Insurance

The insurance page, as shown in Figure 57, is available only from the package detail page. This shows an idea of the process involved for a user to buy insurance.



Enter the requested information to buy insurance.

Insurance Options

Insurance Coverage ($)
Less than $100

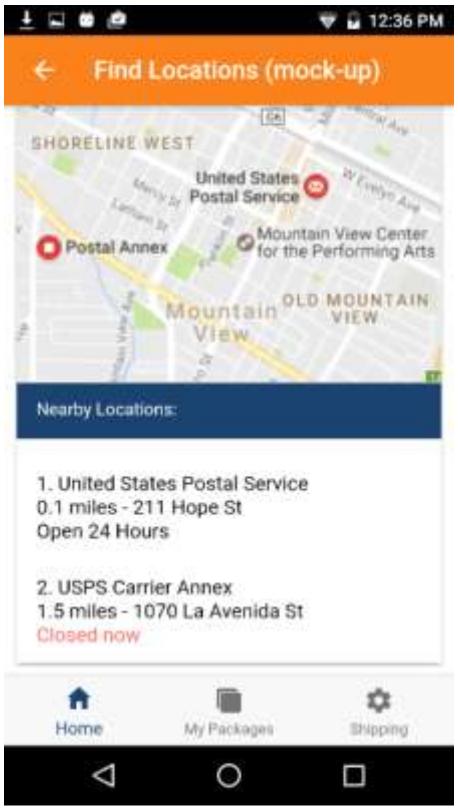
Insurance Provider
None

Figure 57 – Mobile Insurance page

On this page a user selects the amount of insurance they would like to purchase and the insurance provider they would like to use. This feature is important for user's that would like to make sure that an insurance provider can cover damage to important packages.

Find Nearby Locations

The Find Nearby Locations, as shown in Figure 58, is a mock-up of what the future might look like.



Nearby Locations:

1. United States Postal Service
0.1 miles - 211 Hope St
Open 24 Hours

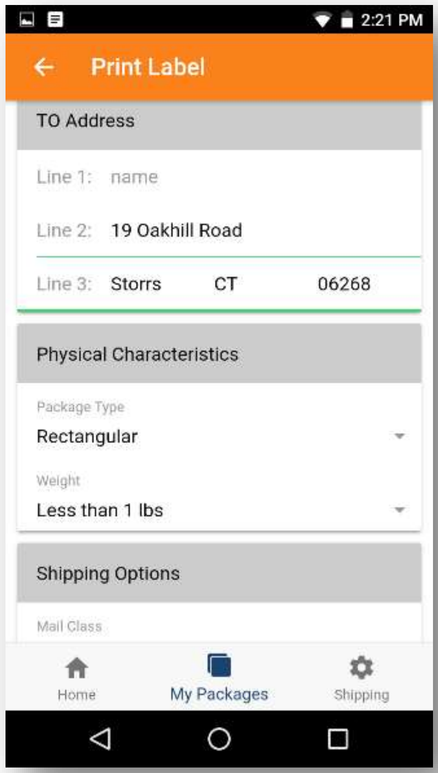
2. USPS Carrier Annex
1.5 miles - 1070 La Avenida St
Closed now

Figure 58 - Mobile Find Nearby Locations page

In the feature a static image of a map and a list of the closest locations appears at the bottom of the screen. This is an important feature to offer for users printing labels.

Print Label (and Return Label)

The print-label page, as shown in Figure 59, includes all the information needed to print a shipping label or a print a return label.



Line 1: name

Line 2: 19 Oakhill Road

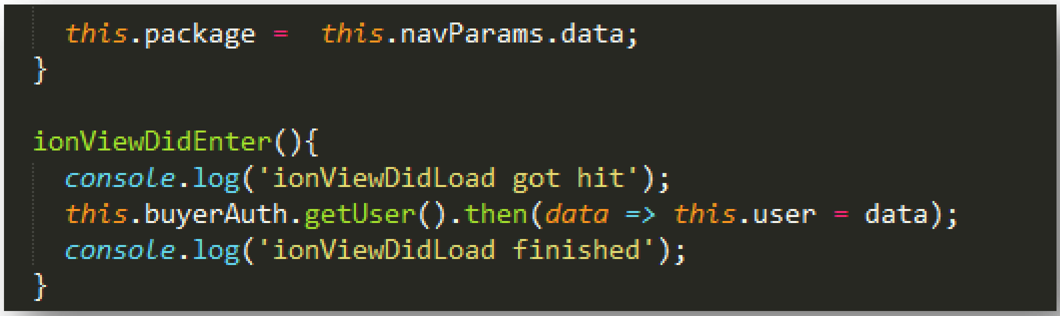
Line 3: Storrs CT 06268

Package Type: Rectangular

Weight: Less than 1 lbs

Figure 59 – Mobile Print Label (and Print Return Label) page

In our prototype, this page auto-populates the From Address with the address associated with the current user. Furthermore, if a user wants to return a package we auto-fill the To Address of the seller as well. We accomplish this by using the current user's address and the package data respectively as shown in Figure 60.

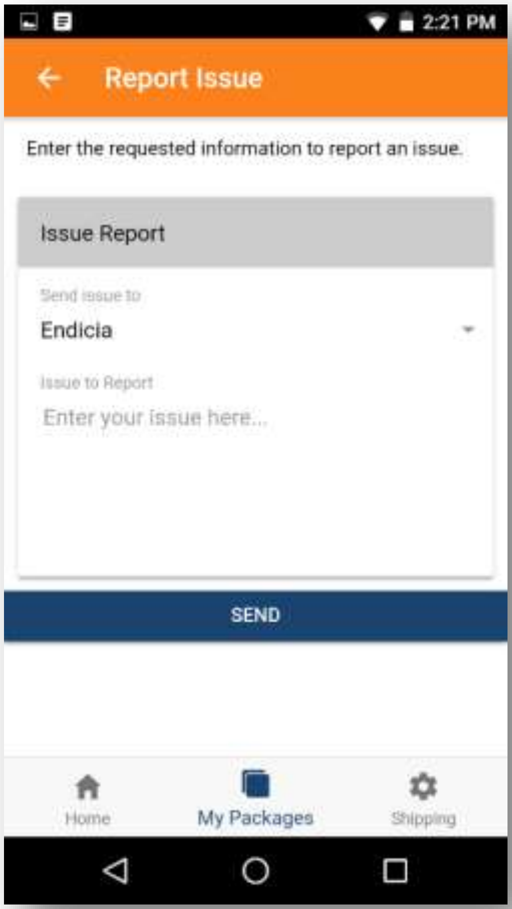


```
  this.package =  this.navParams.data;
}

ionViewDidEnter(){
  console.log('ionViewDidLoad got hit');
  this.buyerAuth.getUser().then(data => this.user = data);
  console.log('ionViewDidLoad finished');
}
```

Figure 60 - Example of BuyerAuthentication retrieving the user

Report Issue

The report issue feature as shown in Figure 61, allows users to report any issues with packages.



2:21 PM

Report Issue

Enter the requested information to report an issue.

Issue Report

Send issue to

Endicia

Issue to Report

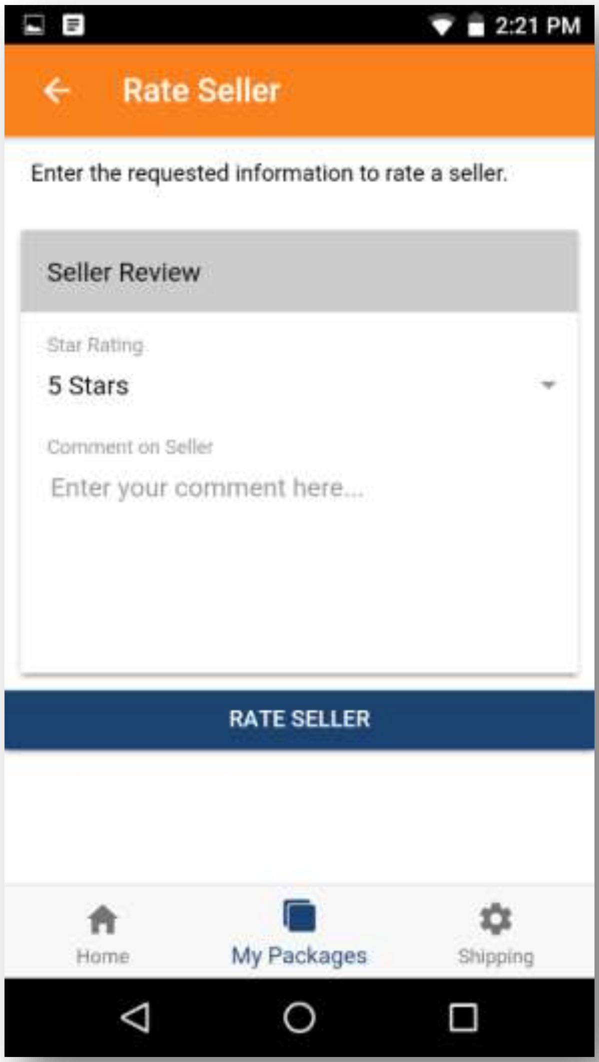
Enter your issue here...

Figure 61 – Mobile Report Issue page

This page involves a simple card that a user can fill out and send to Endicia or the Seller so that they may receive feedback on their business. This is important because customer feedback is useful when entering a new market.

Rate Seller

Lastly, the rate-seller feature, as shown in Figure 62, is available to users who would like to rank sellers.



Enter the requested information to rate a seller.

Seller Review

Star Rating: 5 Stars

Comment on Seller: Enter your comment here...

Figure 62 – Mobile Rate Seller page

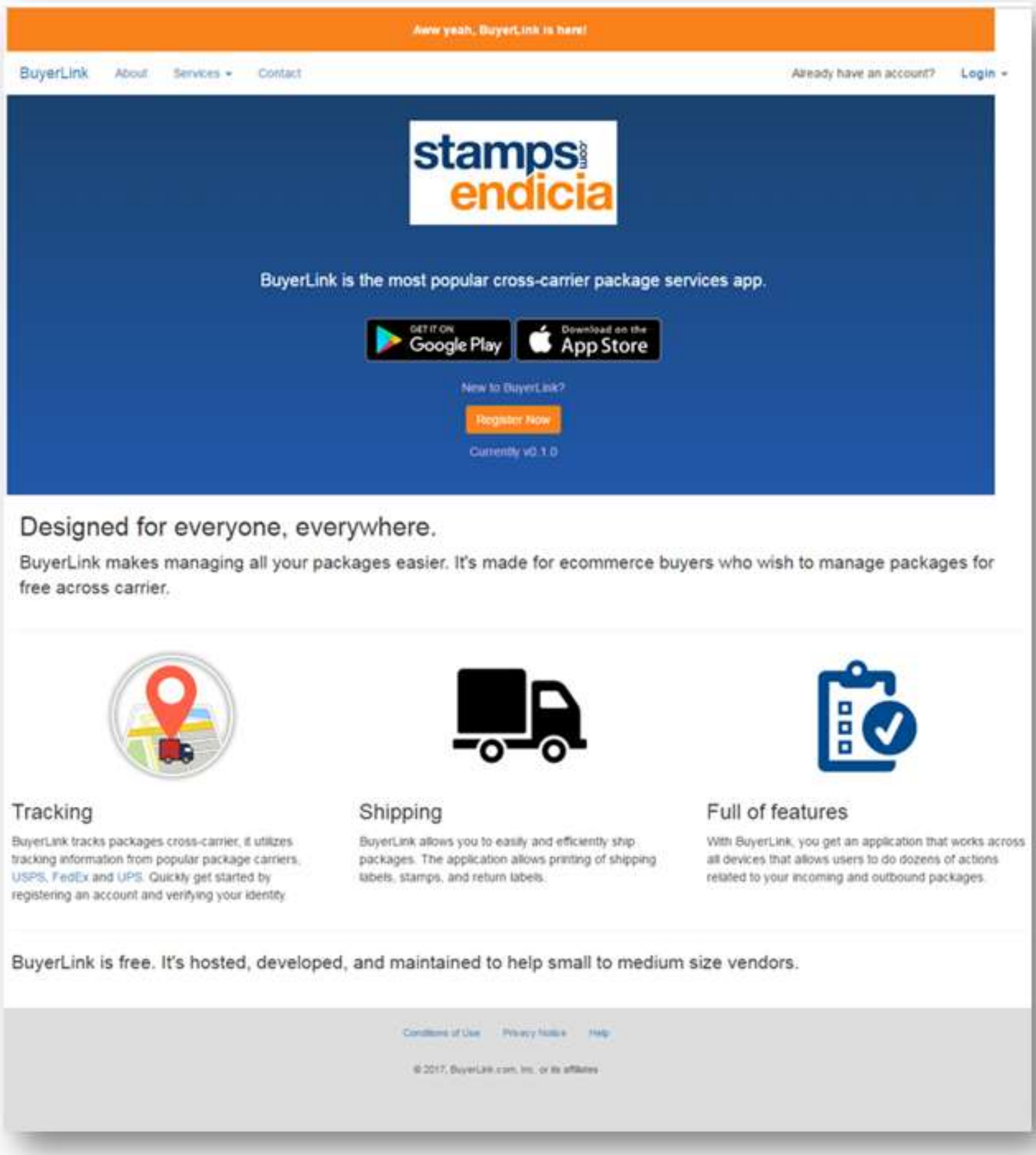
This page could be an important part in helping establish a stronger relationship between sellers and buyers. Over time, consumers will rate more sellers. Once there is an accumulation of ratings the buyers may feel more confident with sellers of higher ratings. The addition of adding seller company logos may complement this relationship too. A complete production version for mobile should consider implementing many of these features.

5.2. Web Prototype

The web prototype contains the registration workflow. We designed the web prototype to capture new buyers receiving Endicia's tracking email. To implement the web prototype we used similar technologies to the mobile prototype with an addition of Bootstrap in place of Cordova and Ionic. In addition to an Angular 2 and Bootstrap front-end, we utilized the existing ASP.net backend to communicate with our database system. Overall, the web prototype consists of five pages: marketing, login, registration, package detail, and package list which we will cover in this section.

5.2.1. Marketing Page

The marketing page, as shown in Figure 63, is the first impression new buyer's will make of BuyerLink, making it an important contributor to the success of the project.



BuyerLink is the most popular cross-carrier package services app.

Designed for everyone, everywhere.

BuyerLink makes managing all your packages easier. It's made for ecommerce buyers who wish to manage packages for free across carrier.

Tracking

BuyerLink tracks packages cross-carrier, it utilizes tracking information from popular package carriers, USPS, FedEx and UPS. Quickly get started by registering an account and verifying your identity

Shipping

BuyerLink allows you to easily and efficiently ship packages. The application allows printing of shipping labels, stamps, and return labels.

Full of features

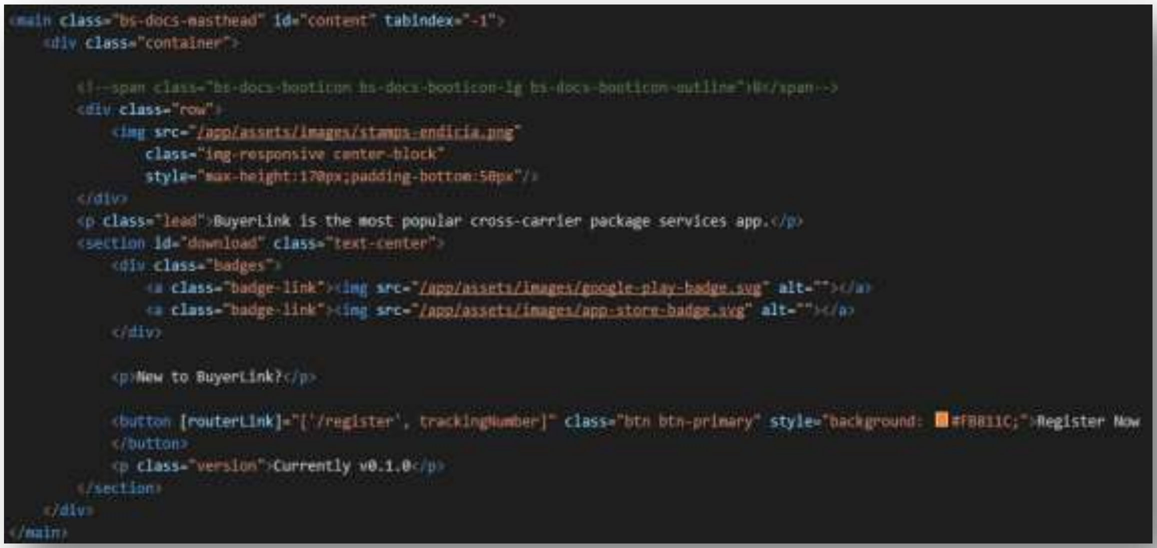
With BuyerLink, you get an application that works across all devices that allows users to do dozens of actions related to your incoming and outbound packages.

BuyerLink is free. It's hosted, developed, and maintained to help small to medium size vendors.

Figure 63 - Web Marketing page

This page is merely a prototype, but is extremely important as it needs to capture the user’s attention and sell them on BuyerLink. In the future, marketing inputs could make this page more compelling to users. The end goal is to make new users want to register an account with Endicia for BuyerLink. On this page, we explain why we developed the application, why users should sign up, and how user’s can get started with the product. We constructed this page with HTML components and CSS styling that specifically utilized Bootstrap style classes and custom style classes.

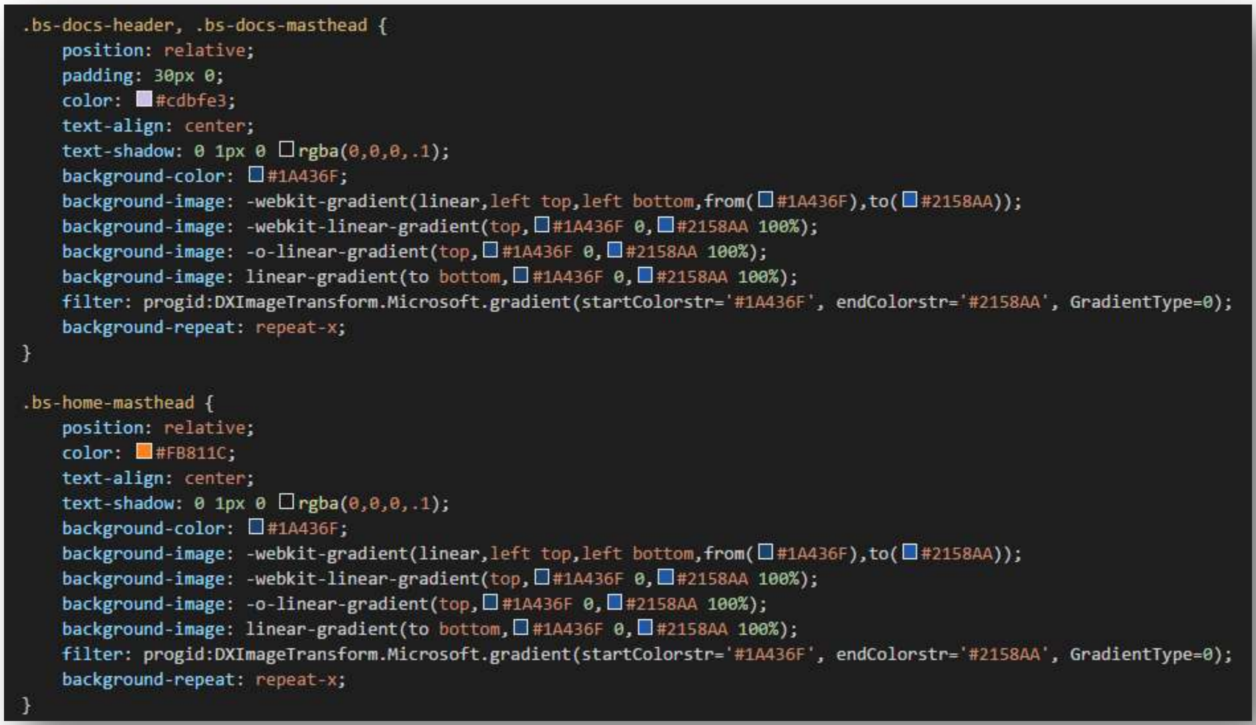
The blue gradient section in the center of the marketing page is a good example of how we used the combination of Bootstrap and custom styling to design the user interface of the web prototype. We begin by specifying the components such as <main>, <div>, <section> with the desired text in HTML as shown in Figure 64.



```html
<main class="bs-docs-masthead" id="content" tabindex="-1">
    <div class="container">

        <!--span class="bs-docs-booticon bs-docs-booticon-lg bs-docs-booticon-outline">B</span-->
        <div class="row">
            <img src="/app/assets/images/stamps-endicia.png"
                class="img-responsive center-block"
                style="max-height:170px;padding-bottom:50px"/>
        </div>
        <p class="lead">BuyerLink is the most popular cross-carrier package services app.</p>
        <section id="download" class="text-center">
            <div class="badges">
                <a class="badge-link"><img src="/app/assets/images/google-play-badge.svg" alt=""></a>
                <a class="badge-link"><img src="/app/assets/images/app-store-badge.svg" alt=""></a>
            </div>

            <p>New to BuyerLink?</p>

            <button [routerLink]="['/register', trackingNumber]" class="btn btn-primary" style="background: #FB811C;">Register Now
            </button>
            <p class="version">Currently v0.1.0</p>
        </section>
    </div>
</main>
```

Figure 64 - Code snippet of HTML defining blue gradient section

In the HTML section above we use bootstrap classes such as class=”row”, or class=”text-center”, or class=”btn btn-primary” to style and organize components on the screen. The classes offered by Bootstrap help developers drastically reduce the amount of time spent designing web pages, however, this styling doesn’t cover branding. In order to successfully craft a professional web page, we required custom styling. Figure 65 gives an example of our custom styling in our main class=”bs-docs-masthead”. The Bootstrap website inspired this styling.



```css
.bs-docs-header, .bs-docs-masthead {
    position: relative;
    padding: 30px 0;
    color: #cdbfe3;
    text-align: center;
    text-shadow: 0 1px 0 rgba(0,0,0,.1);
    background-color: #1A436F;
    background-image: -webkit-gradient(linear,left top,left bottom,from(#1A436F),to(#2158AA));
    background-image: -webkit-linear-gradient(top,#1A436F 0,#2158AA 100%);
    background-image: -o-linear-gradient(top,#1A436F 0,#2158AA 100%);
    background-image: linear-gradient(to bottom,#1A436F 0,#2158AA 100%);
    filter: progid:DXImageTransform.Microsoft.gradient(startColorstr='#1A436F', endColorstr='#2158AA', GradientType=0);
    background-repeat: repeat-x;
}

.bs-home-masthead {
    position: relative;
    color: #FB811C;
    text-align: center;
    text-shadow: 0 1px 0 rgba(0,0,0,.1);
    background-color: #1A436F;
    background-image: -webkit-gradient(linear,left top,left bottom,from(#1A436F),to(#2158AA));
    background-image: -webkit-linear-gradient(top,#1A436F 0,#2158AA 100%);
    background-image: -o-linear-gradient(top,#1A436F 0,#2158AA 100%);
    background-image: linear-gradient(to bottom,#1A436F 0,#2158AA 100%);
    filter: progid:DXImageTransform.Microsoft.gradient(startColorstr='#1A436F', endColorstr='#2158AA', GradientType=0);
    background-repeat: repeat-x;
}
```

Figure 65 - CSS styling for bs-docs-masthead class

This section of CSS styling shows that we give the main component of the marketing page a ‘relative’ position, 30px of padding, centered text, and fill the background from top to bottom to make a clean gradient background. This styling cascades on top of any existing styling for these components and therefore enhances the view to our style preferences. Throughout the web site development process we used similar custom style classes to establish our BuyerLink brand.

5.2.2. Login (Web)

The Login page as shown in Figure 66, is the point of entry for users that are already registered. On this page the user can login with their email and password credentials or, in the future, any of the social login buttons. We strived to make this process lightweight for users by offering several options for login.

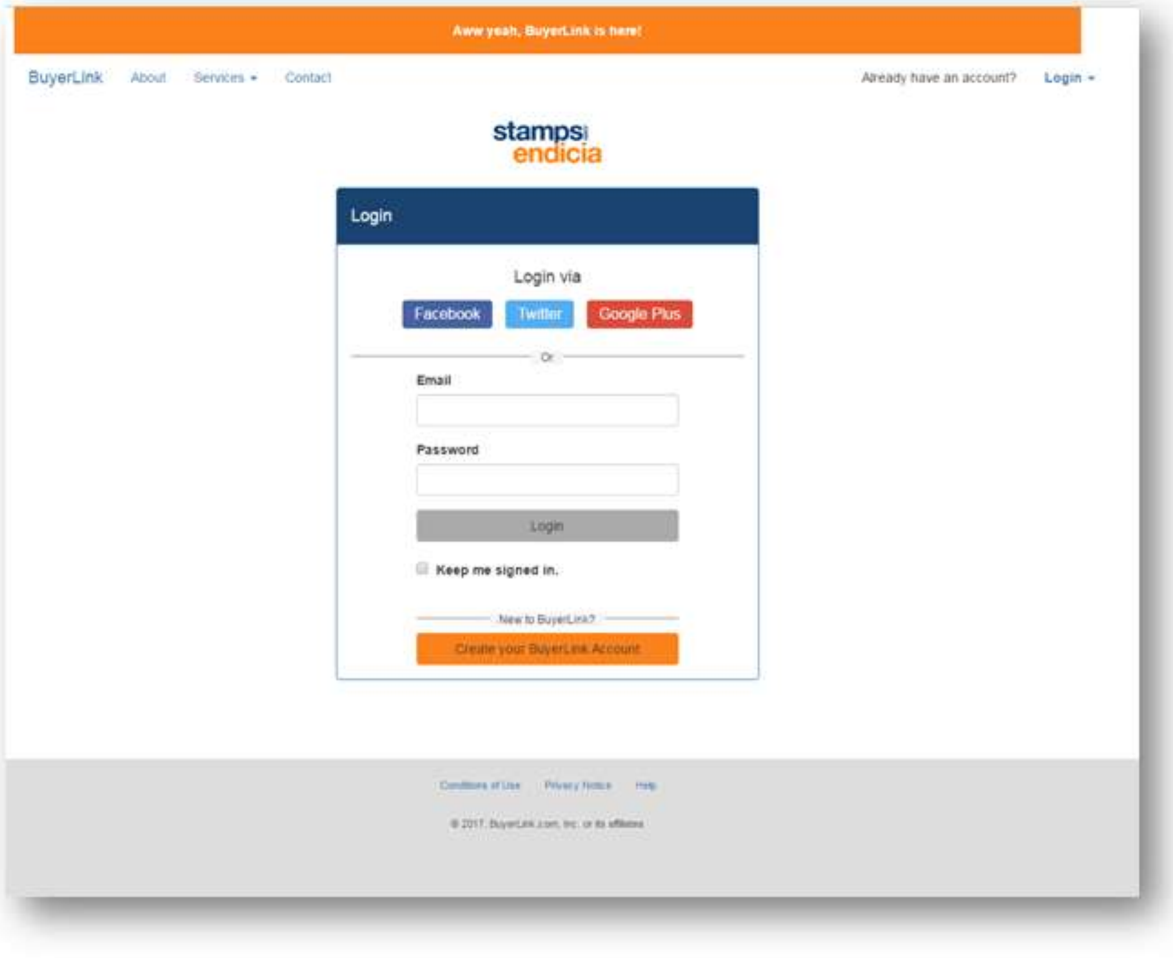
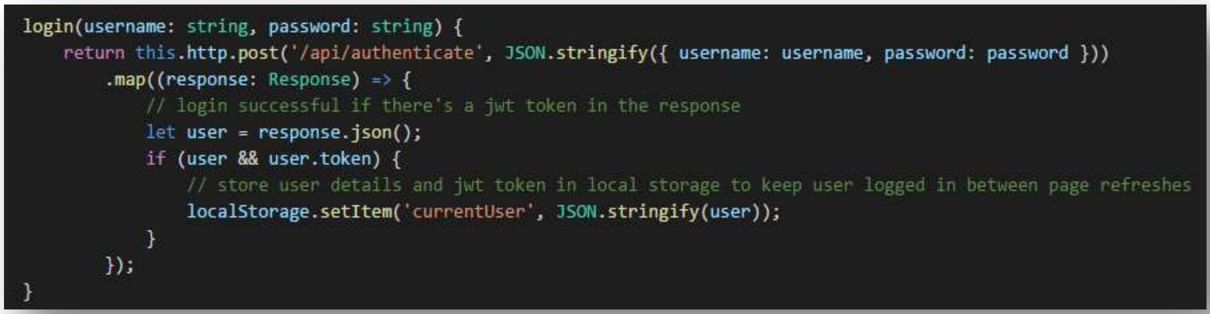


Figure 66 - Web Login page

At this point, we have not implemented social login buttons, and the regular login does not provide proper authorization. To make a complete system we would need to replace our existing web login() method as shown in Figure 67 with the methods used for logins and authentication on mobile.



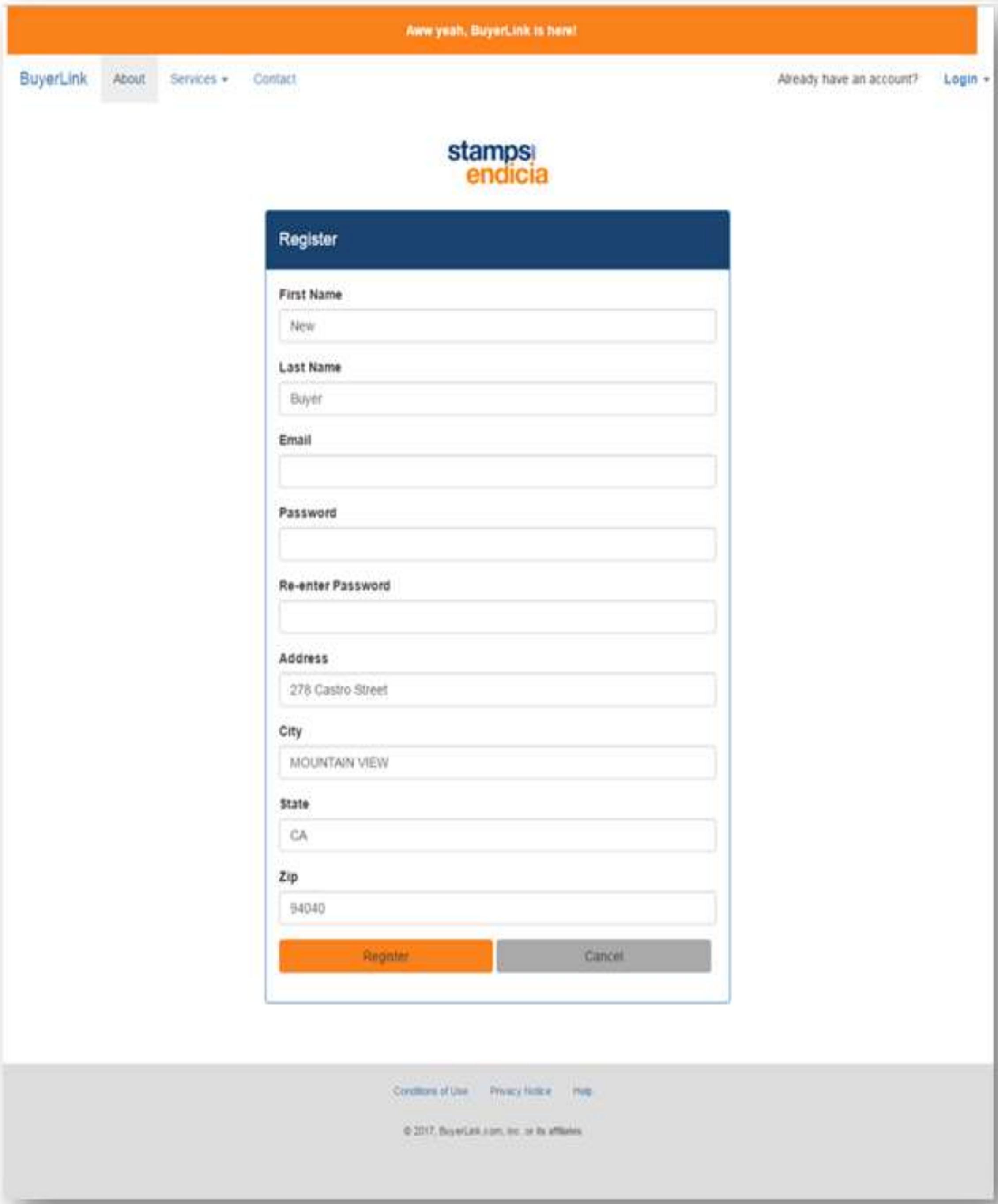
```
login(username: string, password: string) {
    return this.http.post('/api/authenticate', JSON.stringify({ username: username, password: password }))
        .map((response: Response) => {
            // login successful if there's a jwt token in the response
            let user = response.json();
            if (user && user.token) {
                // store user details and jwt token in local storage to keep user logged in between page refreshes
                localStorage.setItem('currentUser', JSON.stringify(user));
            }
        });
}
```

Figure 67 - Web login() method

For now, if the user exists on our fake backend, we log them in. And if not, we alert the user to try again. If a user does not exist they need to register an account in the system.

5.2.3. Registration (Web)

The Registration page as shown in Figure 68, provides a styled form interface to allow users to enter our system.


Aww yeah, BuyerLink is here!

BuyerLink About Services Contact Already have an account? Login

Register

First Name	New
Last Name	Buyer
Email	
Password	
Re-enter Password	
Address	278 Castro Street
City	MOUNTAIN VIEW
State	CA
Zip	94040

Register Cancel

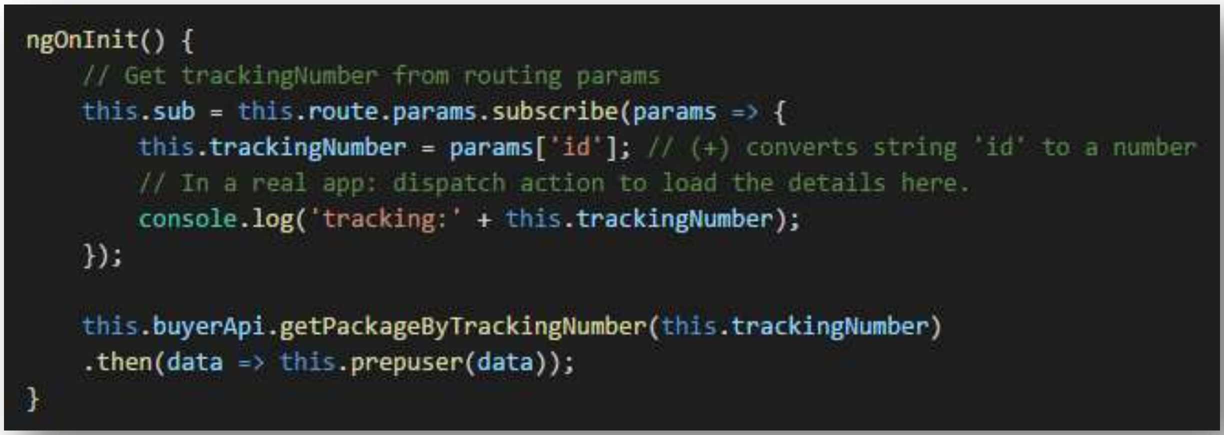
Conditions of Use Privacy Notice Help

© 2017, BuyerLink.com, Inc. or its affiliates

Figure 68 - Web Registration page

For new buyers who received tracking emails, we offer the convenience of auto-populating the name and address fields. This is important because we want the registration process to be as easy as possible. By helping new users register we can make the process feel light-weight.

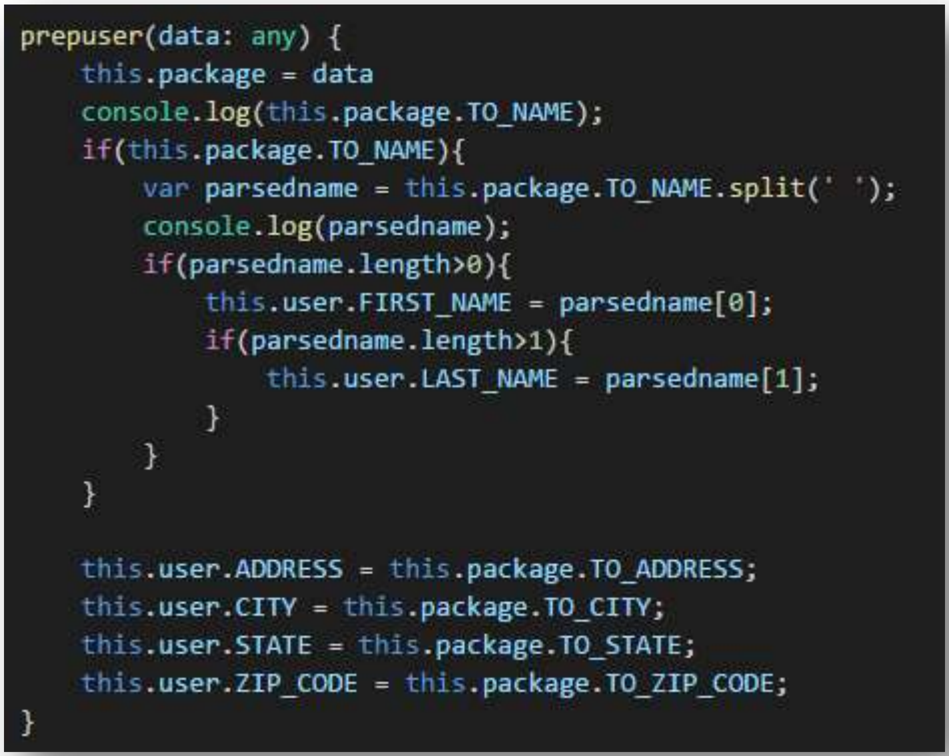
We auto-populate this data by getting the package associated with the tracking number passed from the email. This process takes two steps. The first step, as shown in Figure 69, is having our registration-page component retrieves the associated package with a call to our buyerApi Angular service through ngOnInit(). This example shows in ngOnInit() we retrieve the trackingNumber from the url with the first statement. Later, we then retrieve the package in the bottom statement from our backend with the trackingNumber in our buyerApi Angular service.



```
ngOnInit() {
    // Get trackingNumber from routing params
    this.sub = this.route.params.subscribe(params => {
        this.trackingNumber = params['id']; // (+) converts string 'id' to a number
        // In a real app: dispatch action to load the details here.
        console.log('tracking:' + this.trackingNumber);
    });

    this.buyerApi.getPackageByTrackingNumber(this.trackingNumber)
    .then(data => this.prepuser(data));
}
```

Figure 69 - ngOnInit() retrieving package by tracking number from buyerApi

The second step is once we have the package, we prepare the system to display the package information. We do this with our prepuser(data: any) function as shown in Figure 70.



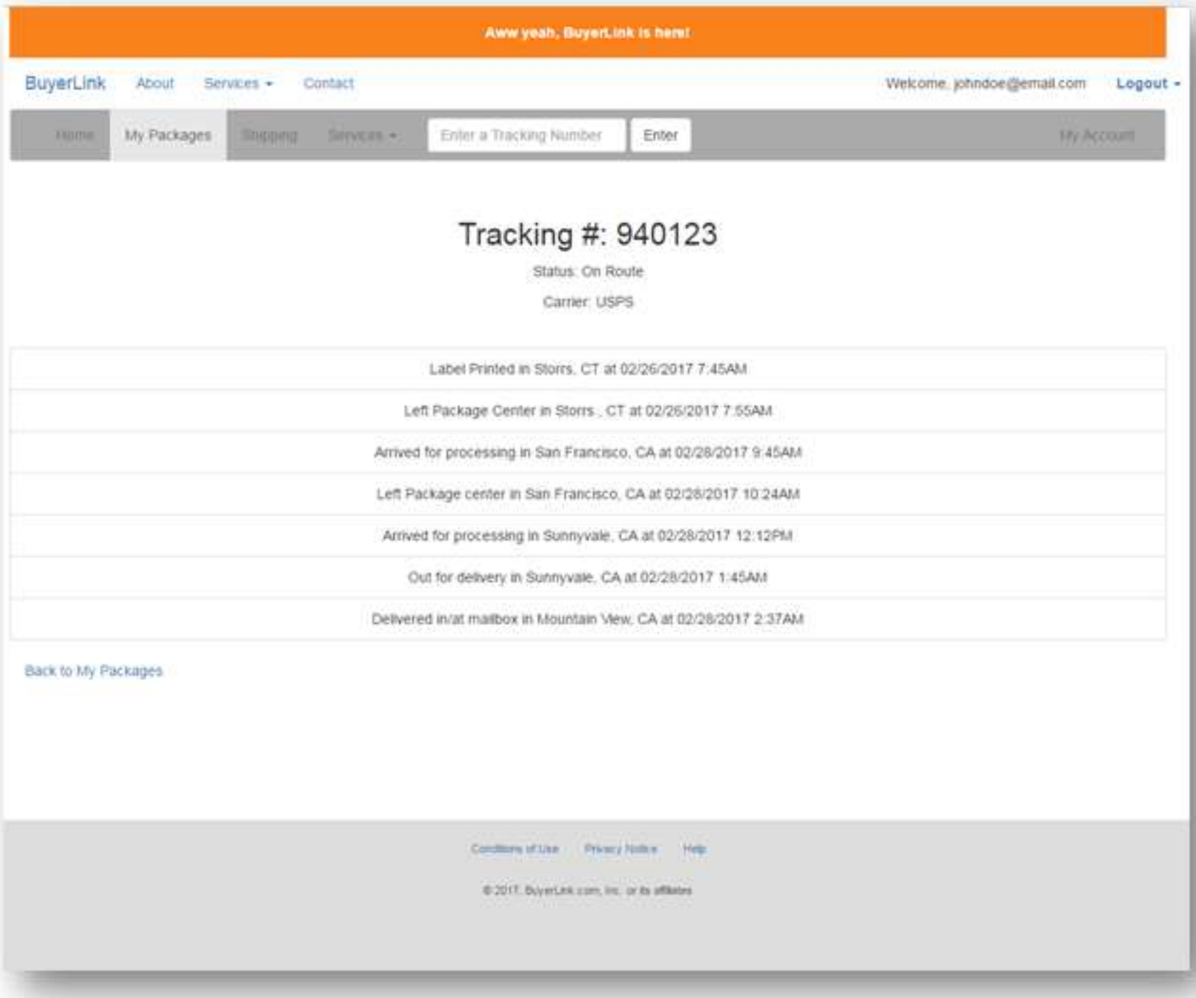
```
prepuser(data: any) {
    this.package = data
    console.log(this.package.TO_NAME);
    if(this.package.TO_NAME){
        var parsedname = this.package.TO_NAME.split(' ');
        console.log(parsedname);
        if(parsedname.length>0){
            this.user.FIRST_NAME = parsedname[0];
            if(parsedname.length>1){
                this.user.LAST_NAME = parsedname[1];
            }
        }
    }

    this.user.ADDRESS = this.package.TO_ADDRESS;
    this.user.CITY = this.package.TO_CITY;
    this.user.STATE = this.package.TO_STATE;
    this.user.ZIP_CODE = this.package.TO_ZIP_CODE;
}
```

Figure 70 - The prepuser() method

The function prepuser(data: any) first sets a package object. It then parses the TO_NAME field of a package into a first and last name. After setting the first and last name, it continues to set the remaining fields of the user object. When the method completes prepping a user object, the two-way binding through Angular allows us to automatically update the display of user fields defined in HTML. Through this, the registration process is almost complete. A user only needs to enter their email, a password, and ‘Register’ to sign up with BuyerLink.

5.2.4. Package Detail

As shown in Figure 71, a user entering the system from a tracking email lands on the package detail page. This screen is the initial reason the user navigated in the email from Endicia. In order to capture the user, the screen must off an initial benefit. This screen is important for these reasons because it will be the first impression of the BuyerLink application.



Tracking #: 940123

Status: On Route

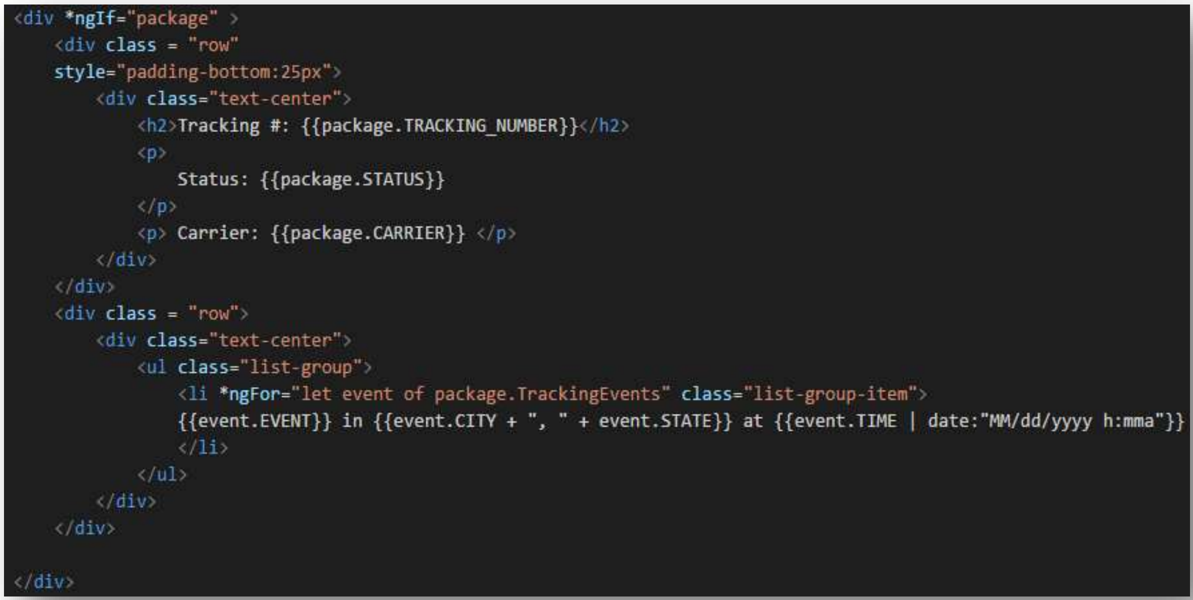
Carrier: USPS

Label Printed in Storrs, CT at 02/26/2017 7:45AM
Left Package Center in Storrs , CT at 02/26/2017 7:55AM
Arrived for processing in San Francisco, CA at 02/28/2017 9:45AM
Left Package center in San Francisco, CA at 02/28/2017 10:24AM
Arrived for processing in Sunnyvale, CA at 02/28/2017 12:12PM
Out for delivery in Sunnyvale, CA at 02/28/2017 1:45AM
Delivered in/at mailbox in Mountain View, CA at 02/28/2017 2:37AM

Figure 71 - Web Package Detail page

On the screen, the user sees that their package information in addition that they logged into the system and can navigate to several other screens such as “Home”, “My Packages”, and “Services”. At this point, the page contains no custom styling so developers could enhance the look and they could be update it to offer services to buyers.

Currently, the package-detail page component populates once the user navigates to the page. This component retrieves the package in the same way the register page component does in ngOnInit() from the server. Once the package is set, the package status and the system displays tracking events through the use of the *NgFor directive as shown in Figure 72.



```
<div *ngIf="package" >
    <div class = "row"
    style="padding-bottom:25px">
        <div class="text-center">
            <h2>Tracking #: {{package.TRACKING_NUMBER}}</h2>
            <p>
                Status: {{package.STATUS}}
            </p>
            <p> Carrier: {{package.CARRIER}} </p>
        </div>
    </div>
    <div class = "row">
        <div class="text-center">
            <ul class="list-group">
                <li *ngFor="let event of package.TrackingEvents" class="list-group-item">
                {{event.EVENT}} in {{event.CITY + ", " + event.STATE}} at {{event.TIME | date:"MM/dd/yyyy h:mma"}}
                </li>
            </ul>
        </div>
    </div>

</div>
```

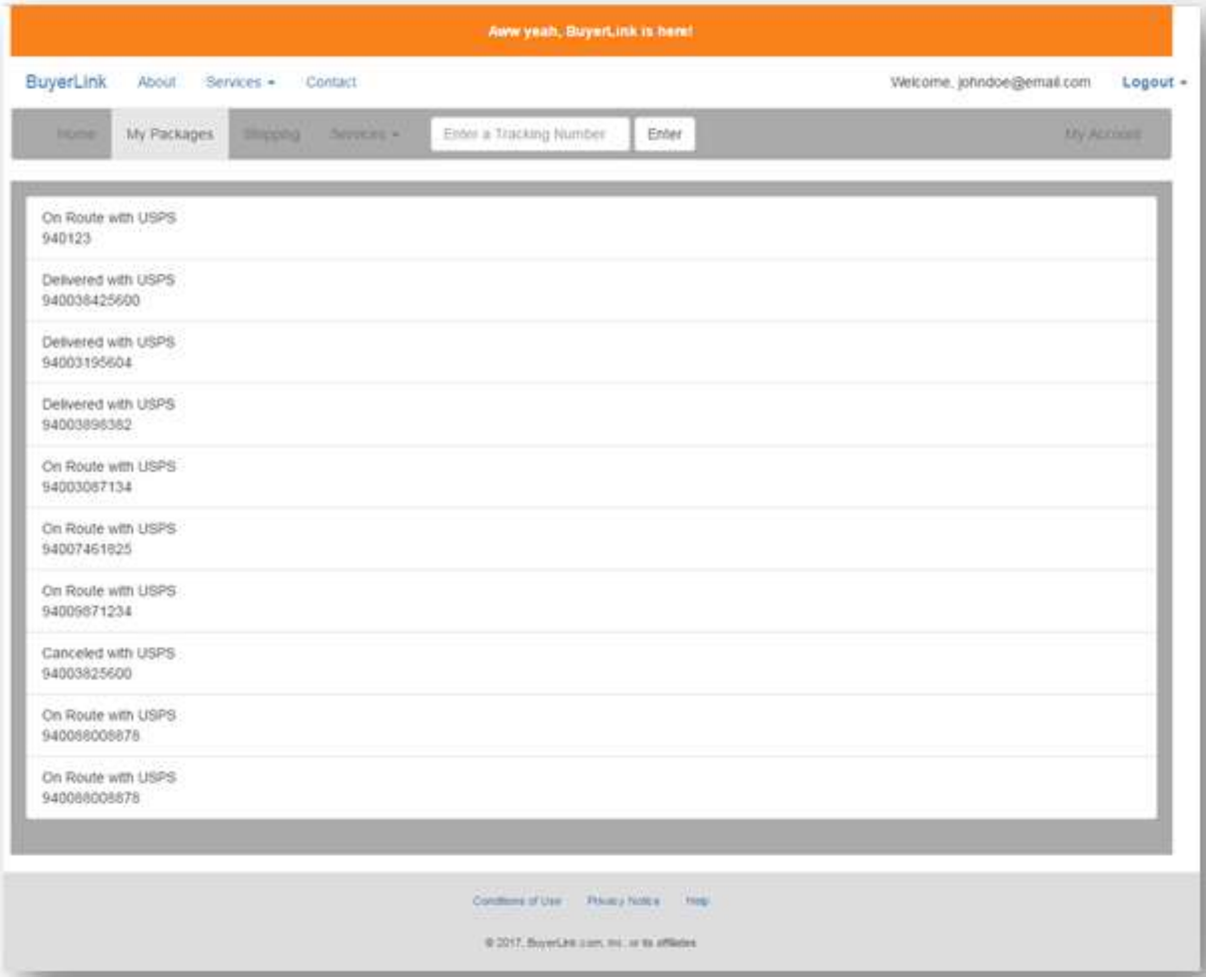
Figure 72 - HTML to display a package and its events

In this code example we iterate through tracking events with *ngFor and filter the date into “MM/dd/yyyy h:mma” format. We additionally utilize Bootstrap styling for the list with class=”list-group” and list items with class=”list-group-item”. The list is presentable but there is clearly room for improvement for the page overall.

From the package-detail page the user can move to the package-list page from the My Packages tab, shown in the navigation bar at the top of the screen, and the link ‘Back to My Packages’, shown at the bottom of the screen.

5.2.5. Package List

The package-list page, as shown in Figure 73, is our last page for the web prototype and it displays a list of the user’s package.



On Route with USPS
940123

Delivered with USPS
940038425600

Delivered with USPS
94003195604

Delivered with USPS
94003898382

On Route with USPS
94003087134

On Route with USPS
94007461825

On Route with USPS
94009871234

Canceled with USPS
94003825600

On Route with USPS
940088008878

On Route with USPS
940088008878

Figure 73 - Web Package List page

For this page, we used similar styling and techniques to the package-detail page component. The system retrieves the user's packages in the same manner and the system displays each package in a list with default Bootstrap styling.

As mentioned before, implementing a complete web prototype was beyond the scope for this project. We focused mainly on providing an interface that users could register with BuyerLink, with an emphasis on registration from the Endicia tracking email. Overall, the web prototype depicts that buyers can registration easily from a tracking email and once registered the system links their package from the tracking email to their account.

6. Conclusions

To establish a link between Endicia and e-commerce buyers we developed a mobile prototype and web prototype. Our prototypes focus on capturing the user experience and satisfying the initial goals set out for the project. By leveraging Endicia’s existing data, our mobile prototype shows we can offer desirable services to users. By adapting the current tracking email to direct users to our web prototype, we created a streamlined registration process that establishes user accounts and gets users viewing package details quickly. By automatically attaching packages to user accounts through our new AutoTrack feature and by utilizing mobile device features like the camera to submit insurance claims, we can grant users greater control of the package recipient experience. Overall, by using traditional web development languages and technologies, we have shown Ionic and Cordova are effective frameworks to develop quality hybrid native applications.

7. Future Steps

This project marks the very first steps in creating a buyer oriented application. As such, there is a great deal of work that developers can complete to take this application and concept forward. We have assembled a set of steps that may move this project forward.

The most direct development that would progress the project would be fully implementing mobile features that we built solely as front-end features or mockups. By actually implementing features like finding nearby USPS locations and printing a return label, the project would be much closer to a production ready application. In addition to continuing feature development on the mobile application, developers might improve the web site to support a similar set of features as the mobile application. Providing users with a consistent experience across the mobile application and the website will build user trust and comfort with both applications.

Another major area for improvement is in security and authentication. As a proof of concept this project had the liberty of labeling many security concerns as out of the project scope. To address this, developers should add security to the API endpoints by requiring authentication tokens from users in order for the system to securely answer requests. Additionally, the seller-facing aspect of the API could require an API key, thus limiting who can add data to the system to just shippers and partners that carry an Endicia approved key. Continuing the security improvements, the identity verification component of Auto Track could be fully implemented by integrating with an API that will generate verification questions. This step is necessary before developers release a production app with an AutoTrack feature.

Lastly, the development team can build our backend web API into a more robust engine. The engine must handle user requests and importing package information efficiently and reliably. This engine could be a multithreaded system, equipped to spawn and dispatch processes to handle individual tasks like sending a push notification, managing database connections, or processing imported data. By building a solid system for this backend, Endicia could increase partner confidence in the viability of the application.

Works Cited

Aggarwal, Ankush (2016, July 16) Push Notification in Ionic 2. (Accessed on 2017, February, 24). Retrieved from https://medium.com/@ankushaggarwal/push-notifications-in-ionic-2-658461108c59#.ajobgimdi

Apache Cordova. (n.d.). Introduction. (Accessed on 2017, March, 2). Retrieved from https://cordova.apache.org/docs/en/latest/guide/overview/index.html

Arango, J., Morville, P., and Rosenfeld, L. (1 October 2015). Information Architecture, 4th Edition. O'Reilly Media Inc.

Bootstrap. (n.d.). Getting Started. Retrieved from http://getbootstrap.com/getting-started/

Bradley, Adam. (2013, October 28). Where does the Ionic Framework fit in?. (Accessed on 2017, February, 24). Retrieved from Ionic Blog: http://blog.ionic.io/where-does-the-ionic-framework-fit-in/?_ga=1.195562988.1766609093.1484761235.

Chapman, C. (2010, October 18). Information Architecture 101: Techniques and Best Practices. (Accessed on 2016, December, 21). Retrieved from Six Revisions: http://sixrevisions.com/usabilityaccessibility/informationarchitecture-101-techniques-and-best-practices/

Covert, Scott B. (n.d.). SPA vs. MVC. (Accessed on 2016, December, 21). Retrieved from https://scottbcovert.github.io/angularjs-meets-salesforce1/#/1

Crosswalk Project. (n.d.). Home page. Retrieved from https://crosswalk-project.org/

Firebase. (2017, February 22). Firebase Authentication. (Accessed on 2017, February, 24). Retrieved from https://firebase.google.com/docs/auth/?gclid=Cj0KEQiA0L_FBRDMmaCTw5nxm-ABEiQABn-VqcjfcUt5VZoSKeSUDj0IMY44QUa53J7NZXrGlWCUaecaAhoB8P8HAQ

Firebase. (2017, February 23). Firebase Cloud Messaging. (Accessed on 2017, February, 24). Retrieved from https://firebase.google.com/docs/cloud-messaging/?gclid=Cj0KEQiA0L_FBRDMmaCTw5nxm-ABEiQABn-Vqab4hD2_9f_GbfAQs9MK_WlpULQsYbSSEro6fz5laQ0aAkQT8P8HAQ

Garcia, Tonya. (3 May 2016). Amazon accounted for 60% of U.S. online sales growth in 2015. MarketWatch. (Accessed on 2016, December, 21). Retrieved from http://www.marketwatch.com/story/amazon-accounted-for-60-of-online-sales-growth-in-2015-2016-05-03

Geldman, Andy. (18 August 2014). 10 Statistics from the Online Marketplace Seller Survey. WebRetailer. (Accessed on 2016, December, 18). Retrieved from http://www.webretailer.com/lean-commerce/statistics-marketplace-seller-survey/#/

Gavruk, Sergey (27 October 2015). Single Page Application Using AngularJS Tutorial. (Accessed on 2017, February, 24). Retrieved from: https://tests4geeks.com/single-page-application-using-angularjs-tutorial/

Google Angular Team. (n.d.). Introduction. (Accessed on 2017, February, 24). Retrieved from https://docs.angularjs.org/guide/introduction

Google Angular Team. (n.d.). Architecture Overview. (Accessed on 2017, February, 24). Retrieved from https://angular.io/docs/ts/latest/guide/architecture.html

Hunt, B. (2006, December 3). Web Site Information Architecture Models. (Accessed on 2016, December, 21). Retrieved from: http://webdesignfromscratch.com/website-architecture/iamodels/

Ionic. (n.d.). v1 Components. (Accessed on 2017, February, 24). Retireved from
http://ionicframework.com/docs/

Ionic. (n.d.). v2 Components. (Accessed on 2017, February, 24). Retireved from
http://ionicframework.com/docs/v2/

Ionic. (n.d.). Pricing. (Accessed on 2017, February, 25). Retrieved from http://ionicframework.com/pricing/index3.html.

Ionic Team. (n.d.). Angular 2 ES6 Promises. (Accessed on 2017, February, 28). Retrieved from http://learnangular2.com/es6/promises.

Lynch, Max. (2017, January 25). Announcing Ionic 2.0.0 Final. (Accessed on 2017, February, 28). Retrieved from Ionic Blog: https://blog.ionic.io/announcing-ionic-2-0-0-final/?_ga=1.27558652.1766609093.1484761235.

Lynch, Max. (2013, October 8). Announcing the Ionic Framework. (Accessed on 2017, February, 28). Retrieved from Ionic Blog: http://blog.ionic.io/announcing-ionic/.

Kearn, Martin. (2015, January 5). Introduction to REST and .net Web API. (Accessed on 2017, February, 28). Retrieved from https://blogs.msdn.microsoft.com/martinkearn/2015/01/05/introduction-to-rest-and-net-web-api/

Morony, Josh. (2016, October 2). A Simple Guide to Navigation in Ionic 2. (Accessed on 2017, February, 28). Retrieved from https://www.joshmorony.com/a-simple-guide-to-navigation-in-ionic-2/.

Rajput, Mehul. (18 March 2016). The pros and cons of choosing AngularJS. (Accessed on 2017, February, 27). Retrieved from: https://jaxenter.com/the-pros-and-cons-of-choosing-angularjs-124850.html

Savkin, Victor. (2016, July 22). Angular: Why Typescript?. (Accessed on 2017, February, 27). Retrieved from https://vsavkin.com/writing-angular-2-in-typescript-1fa77c78d8e8#.ihitfnpyc

Wasson, Mike. (2015, May 28). Getting Started with ASP.NET. (Accessed on 2017, February, 28). Retrieved from https://docs.microsoft.com/en-us/aspnet/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api

W3Schools. (n.d.). Bootstrap 3 tutorial. (Accessed on 2016, December, 21). Retrieved from: http://www.w3schools.com/bootstrap/