Final Report, RepairIT Shop Repair Manager NativeScript App

R. Brian Redd, May 30, 2018

Honors Assignment 3; Multiplatform Mobile App Development with Web Technologies, Coursera; Instructor: Jogesh K. Muppala

Introduction

I've been contacted by a small shop that makes and repairs hand-made one-of-a-kind puppets at Renaissance Festivals. They travel around the country going to various shows, and need a way to manage receipt of puppets to be repaired.

They have asked for an app that can sit on their employee's phones (both Android and iOS) that will allow them to take pictures of puppets when they receive them, detail the necessary repairs required, and attach the customer details so that when the repairs are done, they know exactly who to return them to.

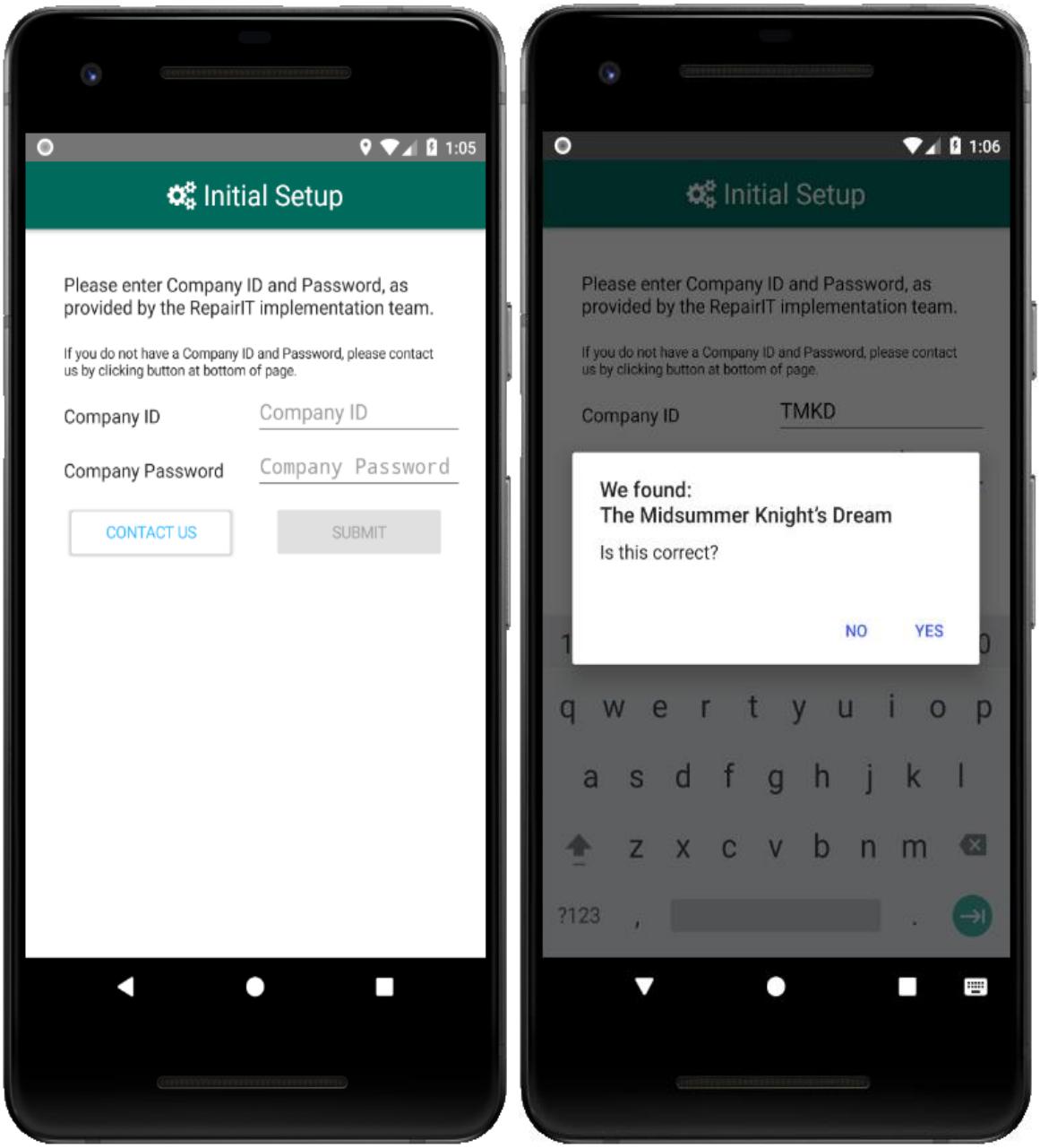
Because of often spotty internet connectivity at these festival sites, the app must be able to function offline, and send the information (either via email and/or upload to a server) when the device has internet access.

Design and Implementation

Initialization of Application

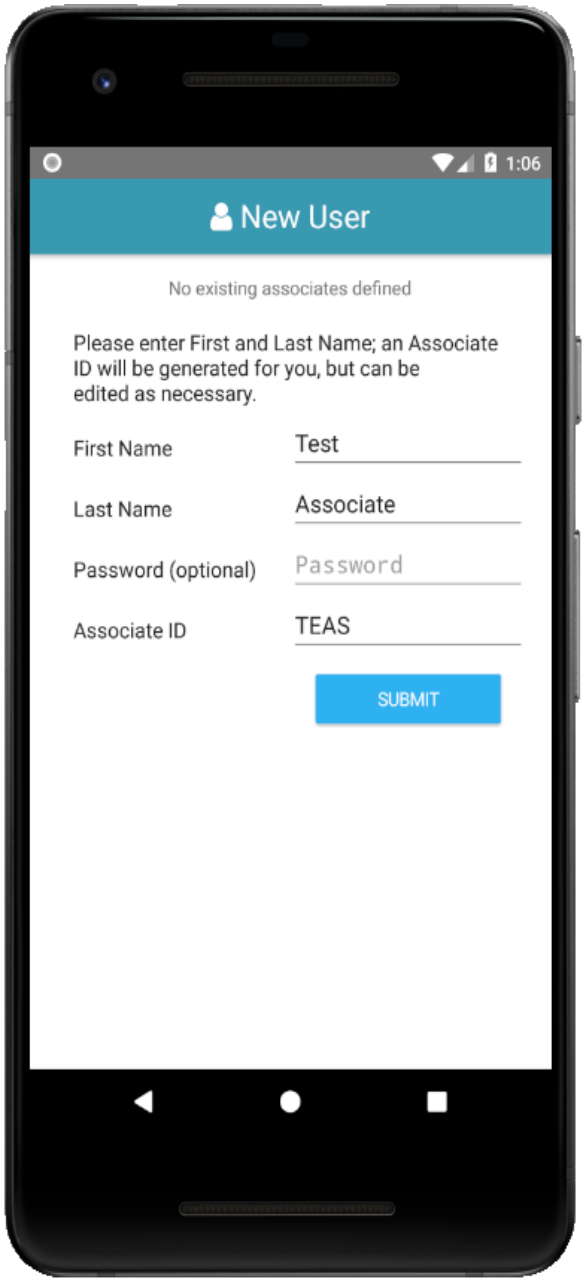
While initially built for a single specific custom, the app is designed to be able to be used by any company with similar needs.

When initially fired, the app will prompt the user for a company ID and password. When entered, the app will make a call to a JSON-server to validate information and prompt the user to verify.



Then the associate will be asked to enter in their name, with an optional password (see Upcoming Features below). From the first and last name, an Associate ID will be generated, and this value will be used in the generated, incremental Order ID.

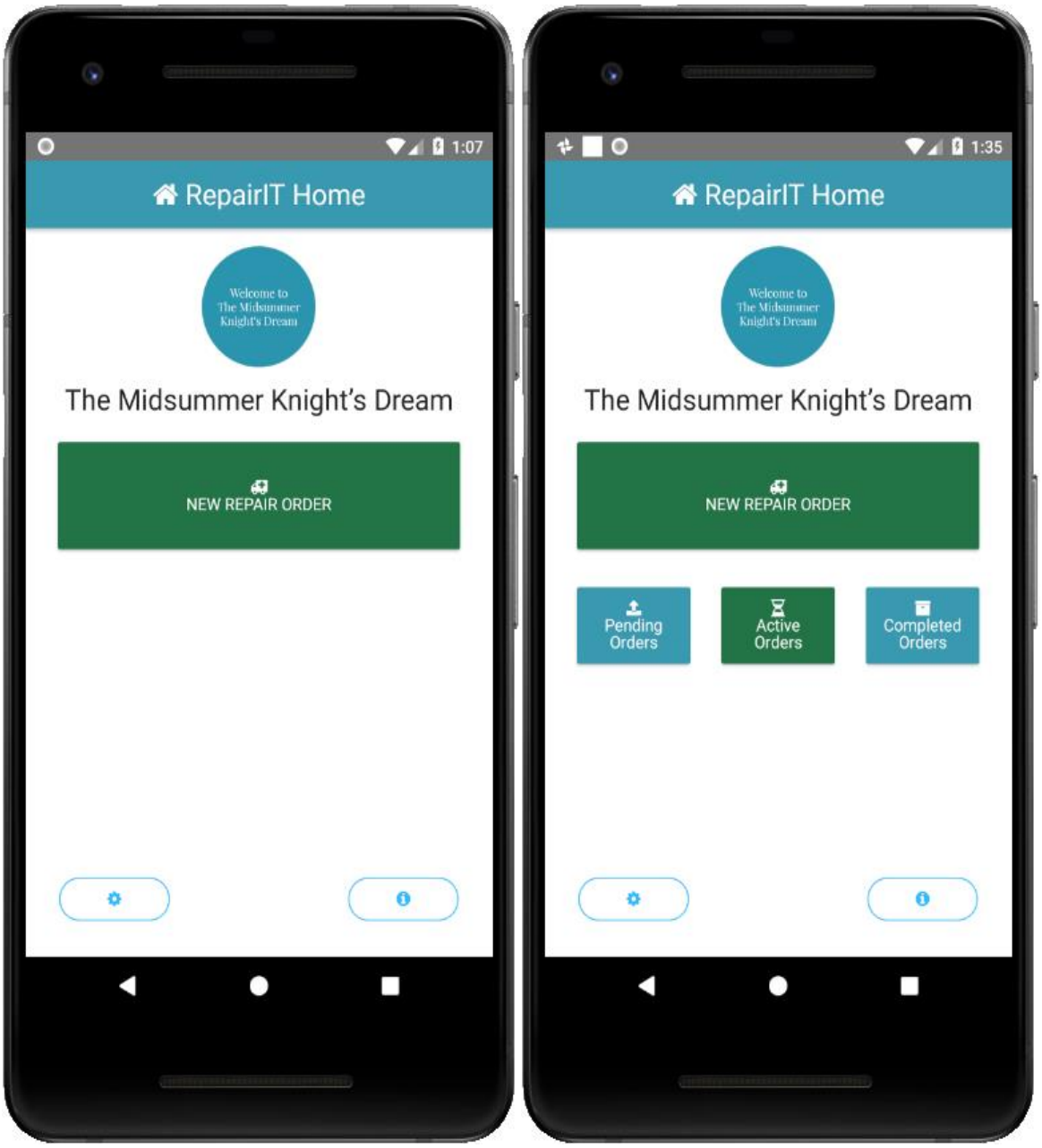
Once entered, the user will be taken to the main screen; this will be the primary screen the app will open to once a user has been stored.



Company details will be grabbed from the JSON-server and stored, for the most part, in application settings, as will the associate information. Additional object- and array-based details will be stored in a Couchbase Lite database.

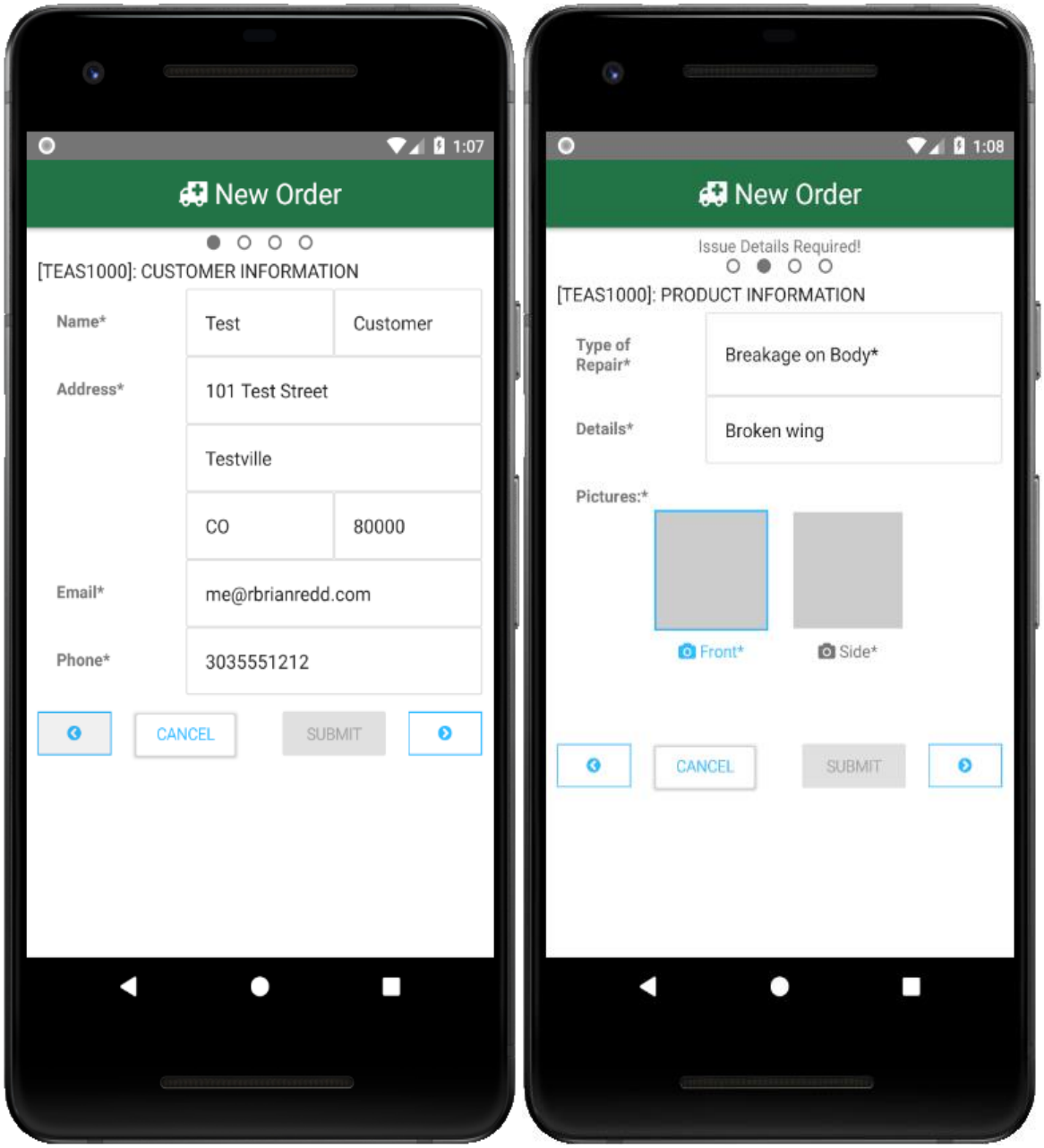
The main screen has four primary buttons along with two secondary buttons: New Repair Orders, Pending Orders, Active Orders, Completed Orders, Settings, and Information. When the associate initializes the app, only the New Repair Orders primary button will be display (as there will be no pending, active, or archived orders).

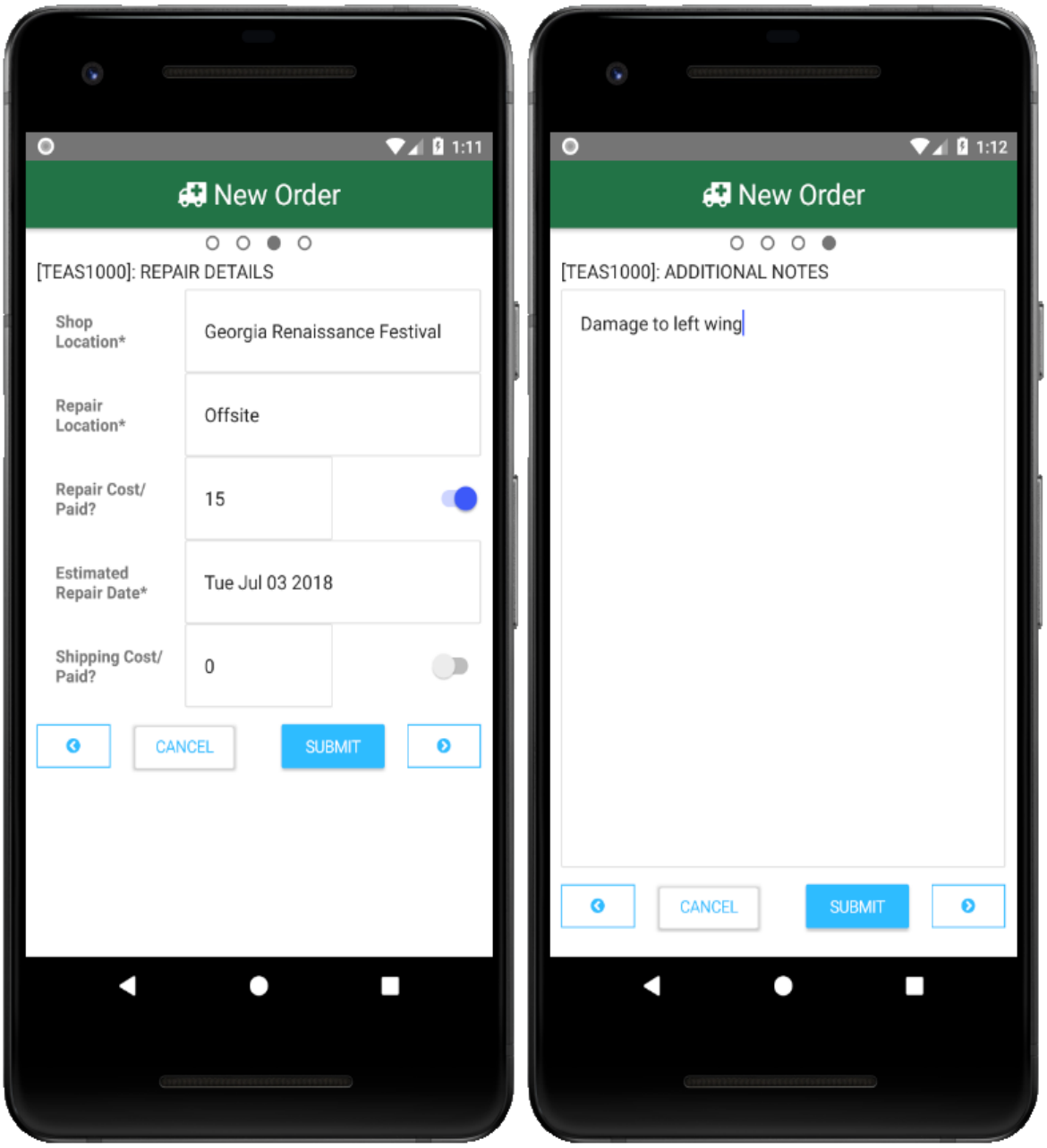
NOTE: Branding such as logo and color scheme are included in the JSON-server company details, and will be stored locally once the app is initialized.



New Order Entry

Tapping the New Repair Order button will take you to the new order page. This page is based on a form and includes four “tabs”. Movement between tabs is available via Next and Previous buttons as well as by left and right swiping.





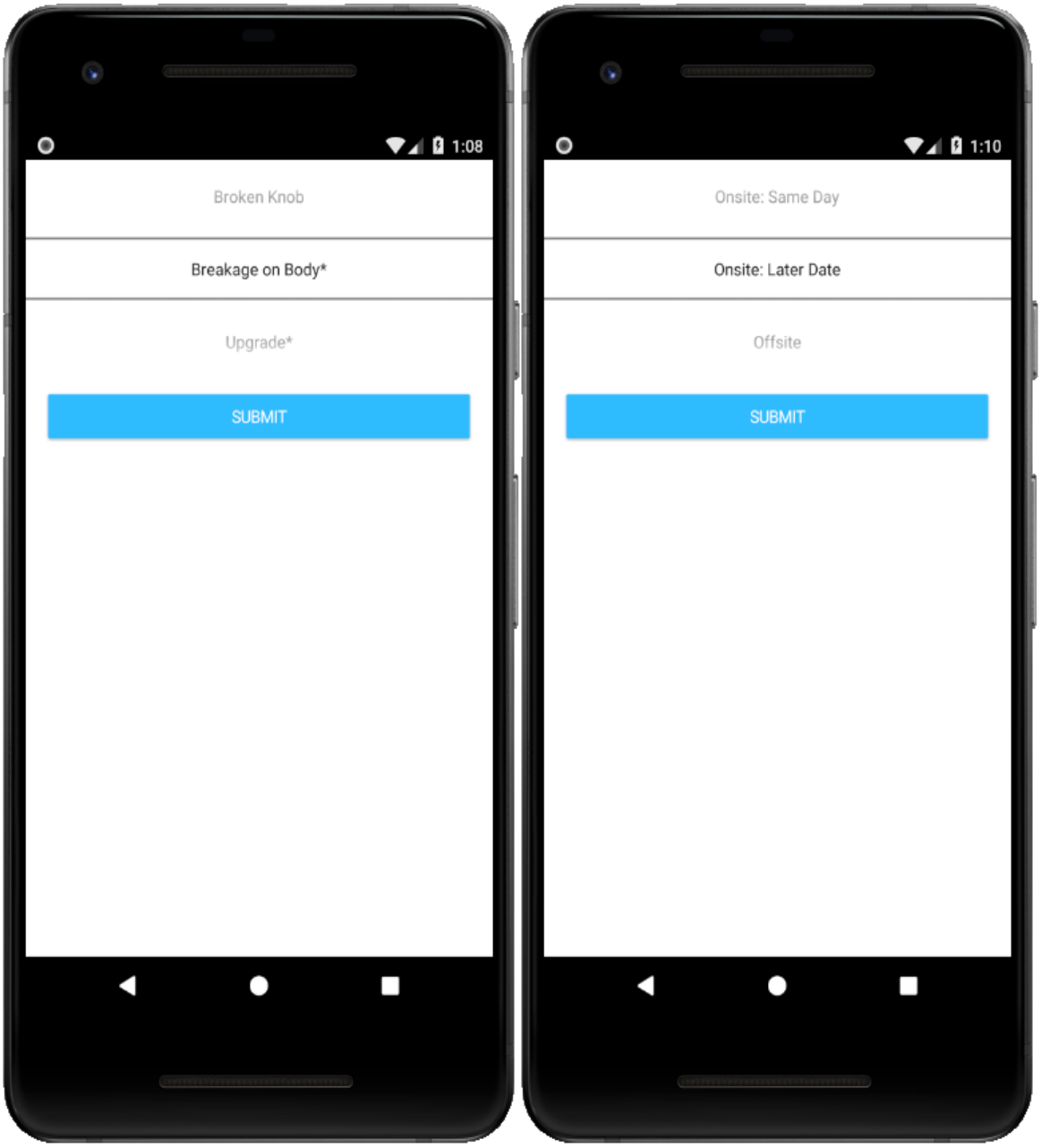
The first tab is for Customer Information. This includes verification of valid email and phone number (the latter strips out unnecessary, non-numeric characters.

The second tab is for Product Information specific to the repairs, including the type of repair, details about the repair, and photos of the object to be repaired.

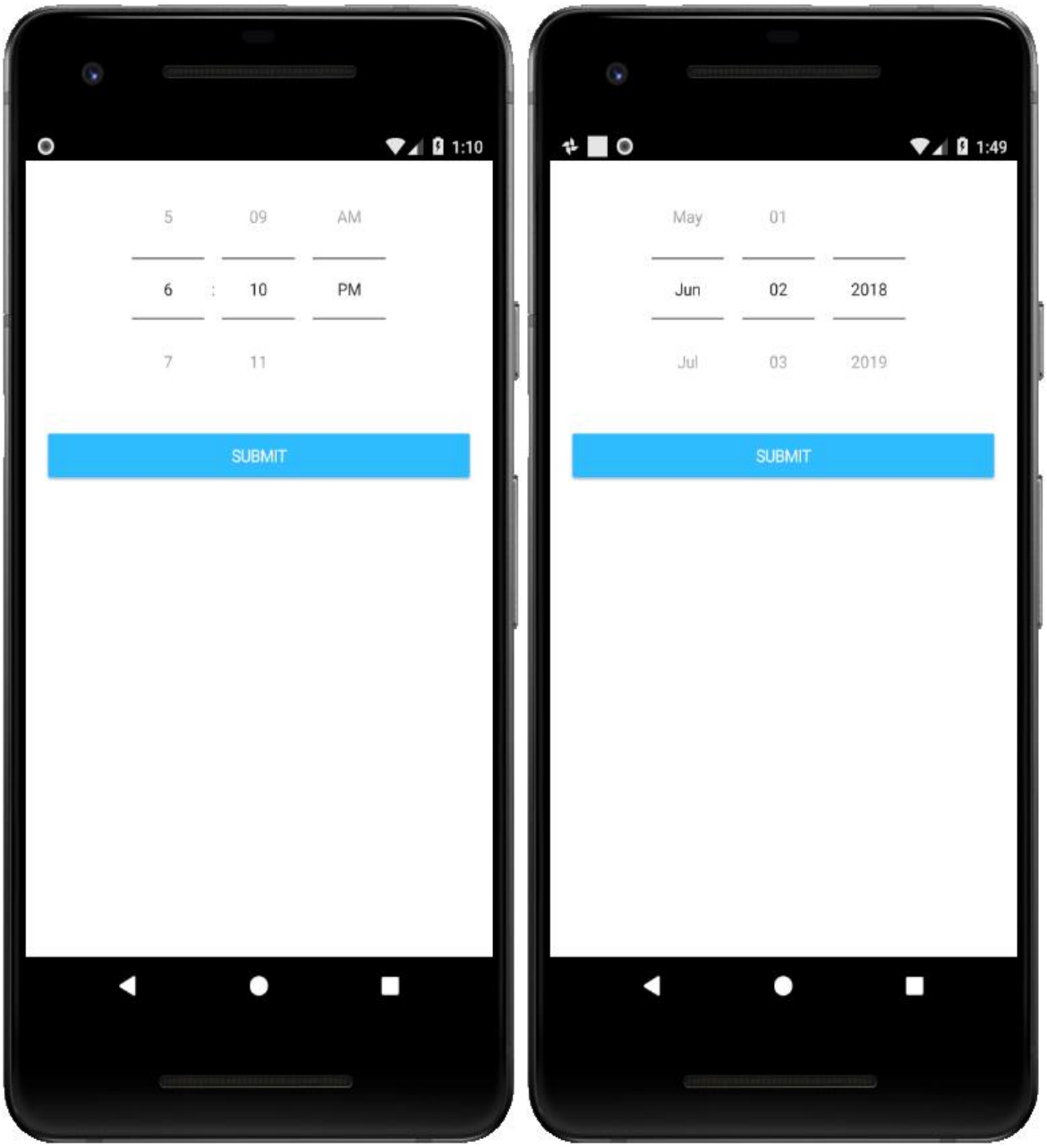
The third tab details the shop location (used to differentiate various renaissance festivals or venues), the location of the repair (will it be performed on-site same day, on-side different day, or will it need to be shipped offsite), costs of repairs and shipping, whether or not those fees have been paid, and estimated time or date of repair completion.

The final tab is for a free-form note field.

There is a modal in place for various fields, such as State, Type of Repair, Location of Repair, Time of estimated repair completion (if repair is to be done on-site same day), or Date of estimated repair.

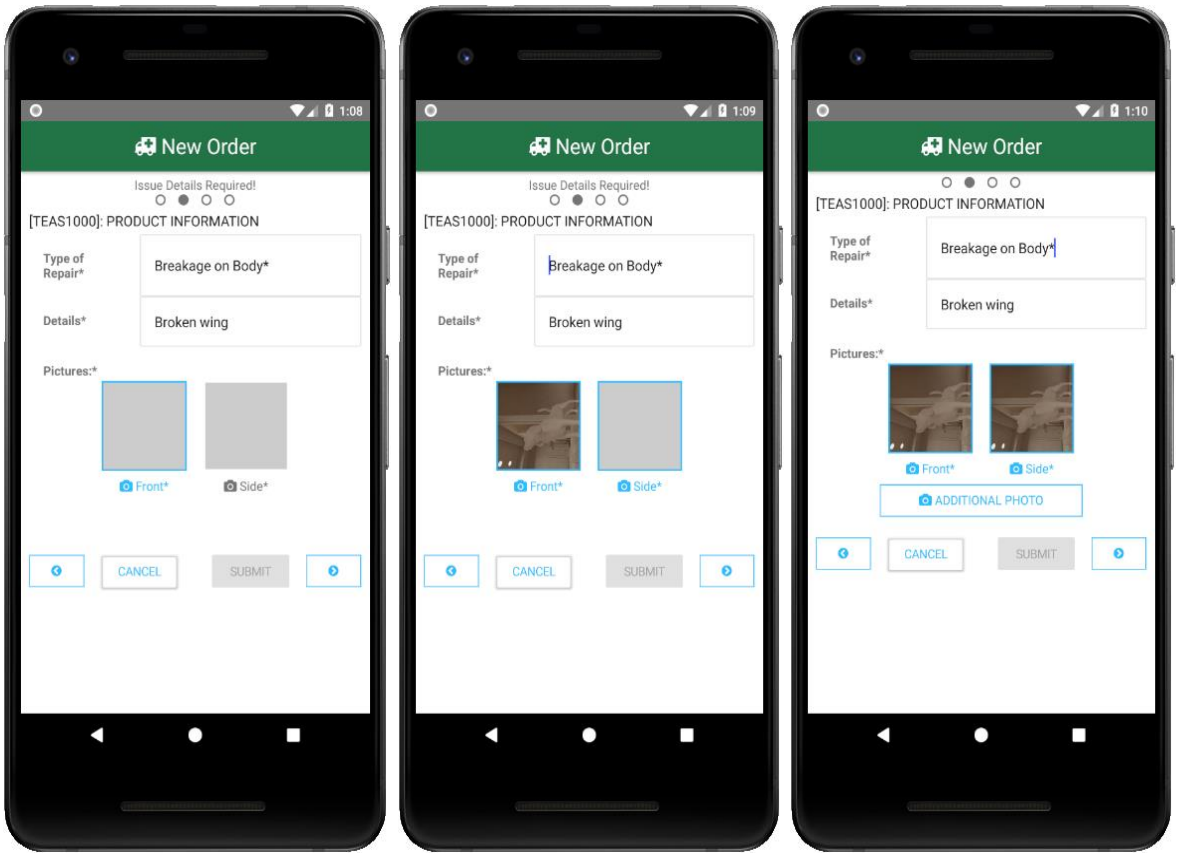


The list of Types of Repairs are included in the company JSON-server capture, so they can be customized per company. In addition, any type of repair with an asterisk at the end requires the Issue Details to be filled out; otherwise that field is optional.



Required as part of the repair order are photographs to be taken of the item being repaired. The number and caption of photos are custom per company. In addition to the number of required photos, there is the option to take any number of additional photos.

Each photo will be taken in turn, and the camera plugin is fired when the highlighted button is tapped. Once a photo is taken, a thumbnail of that will be displayed. Tapping a photo that has already been populated will allow the employee to retake the photo.



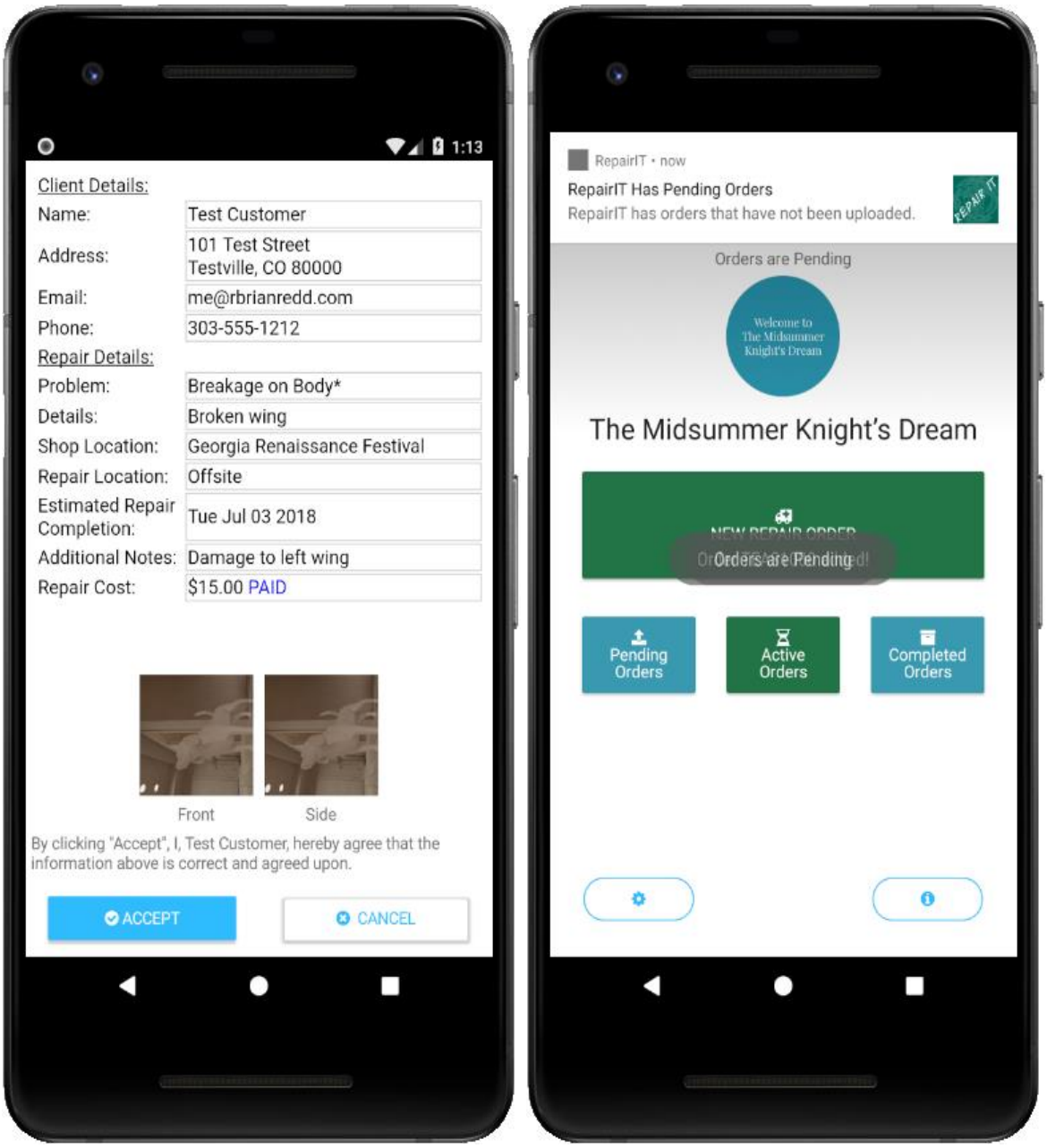
The form can only be submitted when all required fields (designated with an asterisk) are populated, including the required photos.

Canceling will return the employee to the main screen.

When Submitted, a summary of the information will be displayed. Here, the customer can read over the information and either Accept or Cancel. Canceling will return the employee and customer to the form where information can be updated.

Acceptance will cause the form data, including photo information, to be pushed into an Order object which is in turn stored in the Couchbase Lite database.

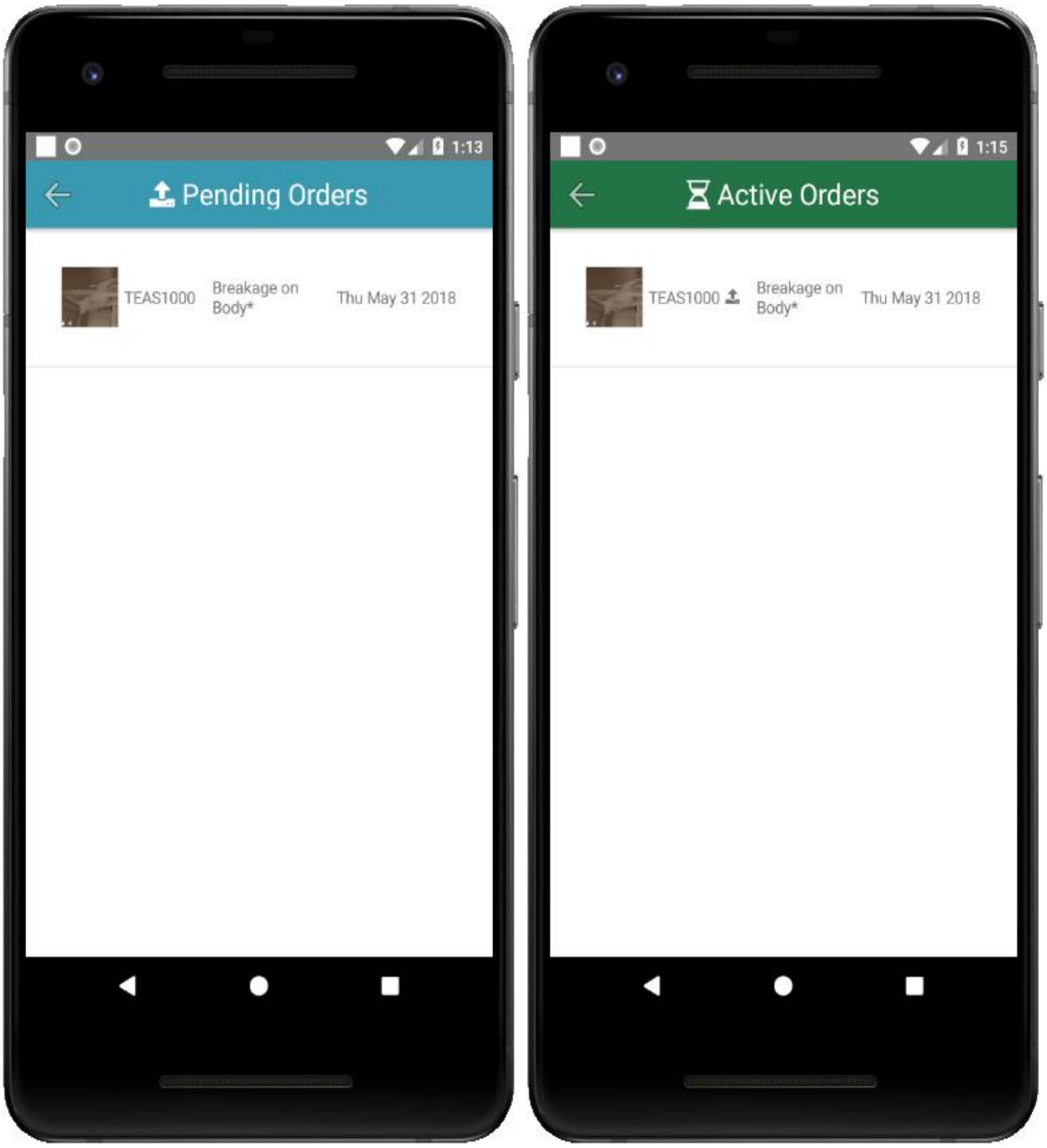
Once the first order is stored, access to “Pending Orders”, “Active Orders”, and “Completed Orders” will be made available.



Order Management and Uploading

A “Pending Order” is any order that has not been “uploaded”. Initial design of the app uses email as the upload method, sending details to the company bookkeeper. Any order that has been “uploaded” but is still active (hasn’t been completed with the product delivered to the customer) is considered “Active”, and is available via the appropriate button.

When there are Pending Orders (orders that haven’t been uploaded), notification will be sent to the phone letting the employee know. This notification will be resent whenever the app is opened while there are Pending Orders.

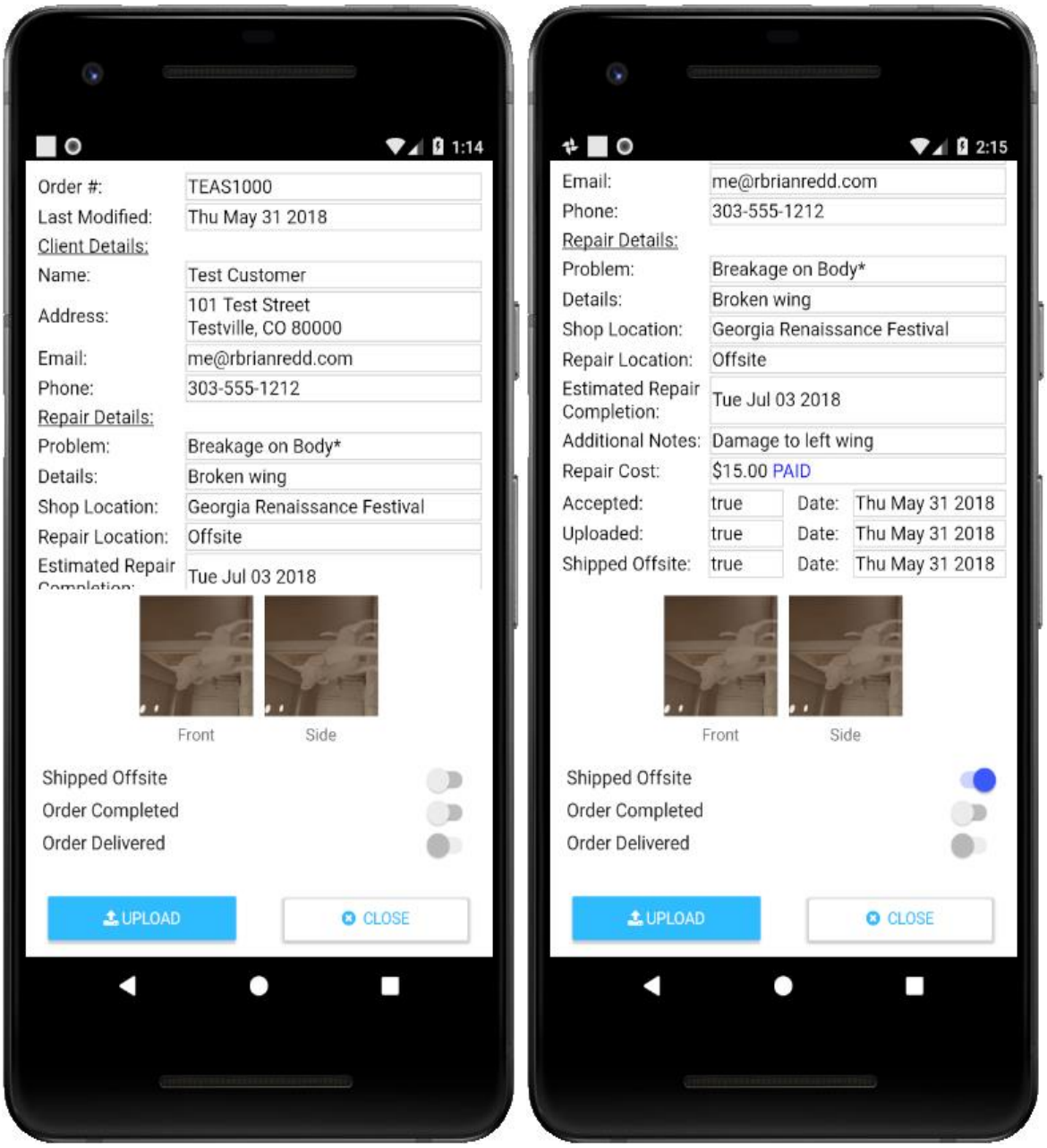


Pending Orders are also available under the Active Orders page, and are designated with an upload icon if they have been uploaded.

Currently the listing of orders are by Order ID (the Associate ID plus an incrementing number that is stored in app settings), the type of repair, the date the repair order was submitted, and a thumbnail of the first image taken.

If there are no Pending Orders (or Active Orders, or Completed Orders), text stating this will be displayed when the appropriate page is opened.

Tapping on an order will open an order summary page.



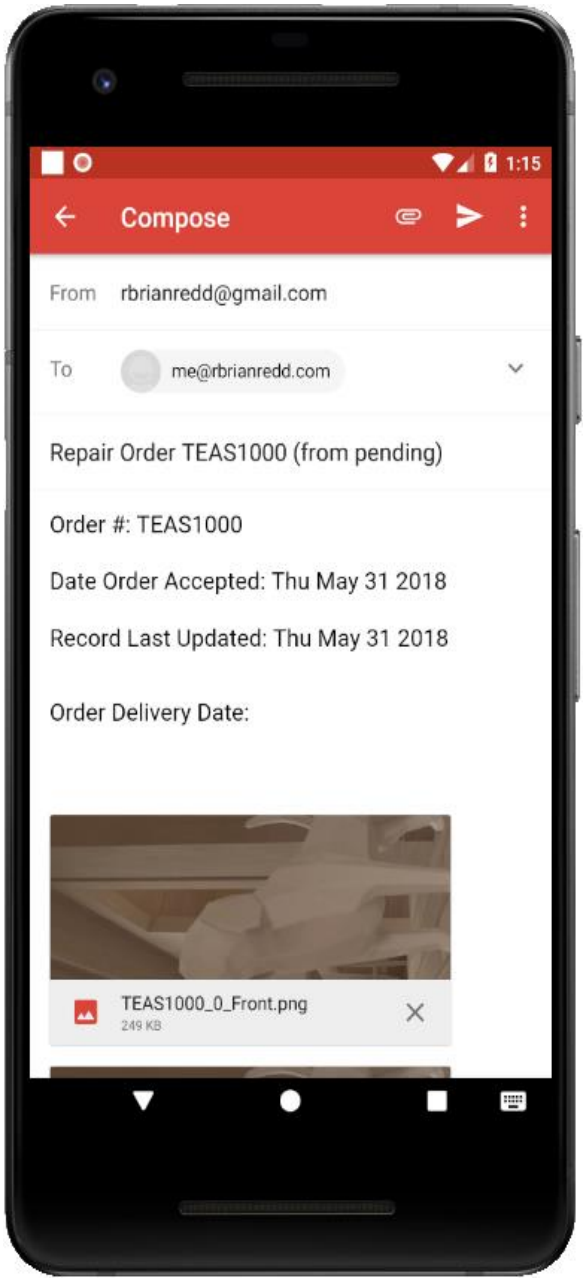
In the Order Summary page, the options to Upload and Close are available, along with appropriate flags that can be edited. For instance, if the order needs to be shipped offsite, then when the order is shipped the order can be opened and the "Shipped Offsite" flag set to true. This will not only record that the order has been shipped, but it will also capture the date the flag was set.

Likewise, if an order was made without payment, and then the customer pays,

While initially built for a single specific custom, the app is designed to be able to be used by any company with similar needs.

Uploading an order fires the device's email application and populates an email with all the order details, including attachments of each photo taken.

The email address that the details will be sent to are specified in the customer JSON-server data.



Implementation Challenges

The first challenge was determining exactly which data was required per customer, and how to store and access that data while the app is running. I used application settings as much as possible, but there are some pieces of data (the list of store locations and repair types) that had to be stored in Couchbase.

The largest challenge so far was with image handling. Initially, I had wanted to use Base64 encoding to convert the images into strings so they could be stored directly in the Couchbase Lite database. Unfortunately, this proved impossible (at least with the information I was able to find on the internet, the NativeScript documentation, and the constraints of time). In the end, I had to satisfy myself by storing the images assets within the app itself and then storing and calling the paths of these images in Couchbase. This proved sufficient as the images from the app can be added into an email as attachments.

Once a server is available where I can upload data directly, I will either have to revisit the Base64 option or simply upload the image assets during the upload process.

Anticipated Future Enhancements

First and foremost will be the integration of a full back-end server where order details can be literally uploaded and stored. This should become available after the next Coursera course on NodeJS, MongoDB, etc.

Since current uploading is done through email (which has its own online/offline handling), I did not include network detection in the initial build of RepairIT. This, however, will be included in a future release. Certain functions, such as upload and even app initialization, will only be available when the device is online.

Currently the app is designed for a single employee's use; however, in many cases a single device may be used by multiple employees (for instance, customers who use Square as their payment/register system may wish to add RepairIT to the same device).

Currently, the Settings page only includes a button to clear application and Couchbase data (used in development). When released, this page will instead allow an employee to sign out, and then, if no current employee is signed in, when the app is opened an employee sign in page will be fired. From here, employees can choose themselves from a list of initialized employees, or create a new employee profile. This is where the optional password entered during employee initialization can come into play, if particular employees wish to protect their profiles.

Finally, company information will be listed in the Information page.

Conclusions

This was a challenging but fulfilling project to work on. I learned a lot about NativeScript and how to integrate a variety of plugins (Couchbase Lite, the camera, email, and local notifications in particular), as well as gaining a further handle on Angular.

References

- NativeScript website: https://docs.nativescript.org/
- NativeScript Plugin Marketplace: https://market.nativescript.org/
- Coursera NativeScript course: https://www.coursera.org/learn/nativescript/
- Couchbase Blog (specifically, a post about storing images as Base64 strings, although the information provided appears to be out of date as it would not work in my implementation): https://blog.couchbase.com/save-captured-images-nativescript-angular-application-couchbase/
- My JSON Server: a mock JSON-server where JSON data stored in GitHub can be read by applications, used for my Customer JSON data: https://my-json-server.typicode.com/