KŌSOKU
INTERACTIVE

Sample App Report

Prepared for: Acme Co.

June 29, 2007

AppRescue℠

KŌSOKU
INTERACTIVE

TABLE OF CONTENTS

KŌSOKU
INTERACTIVE

EXECUTIVE SUMMARY

Acme Co. hired an offshore development firm out of South America to develop their new marketplace app for iOS. The development firm has been working on the app for a little over one year and Acme Co. is beginning to have concerns regarding the app's development progress.

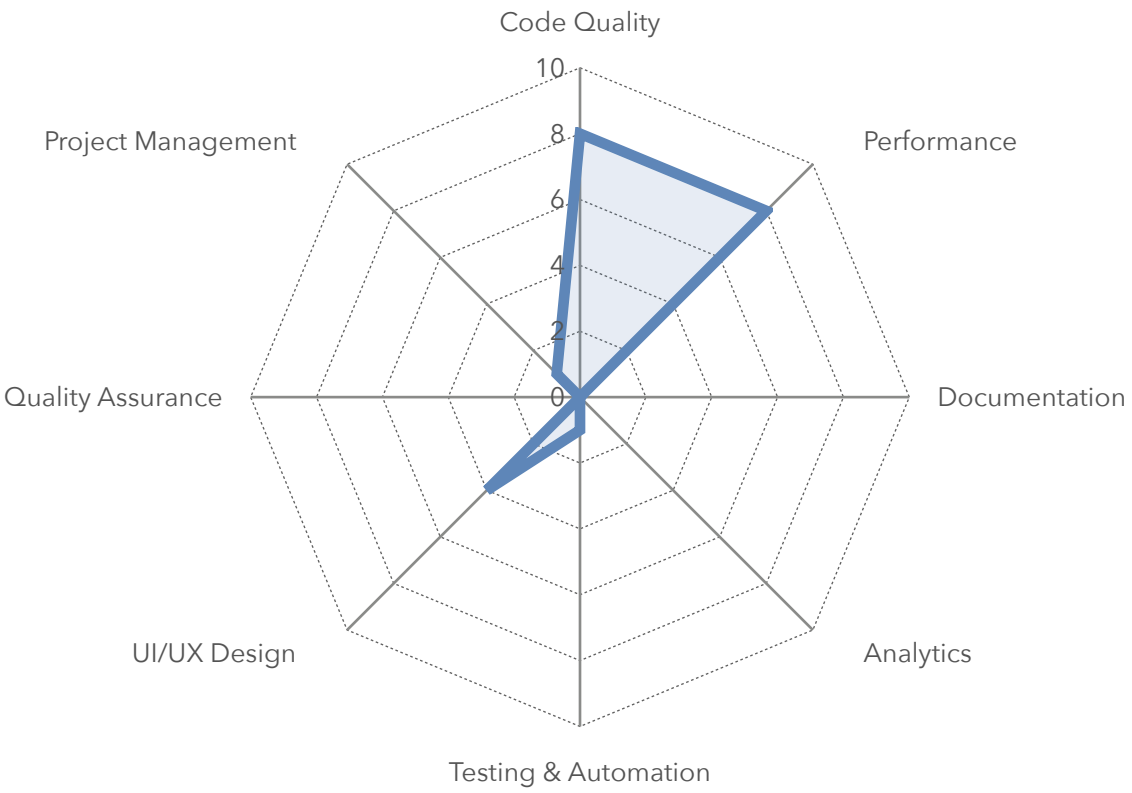
During the discovery meeting, Acme Co. voiced concern over how the agency allows the stakeholders to review the app. To date, no version of the app has been made available to the stakeholders at Acme Co. and the only insight into its progress is through controlled demoes over a video conference call. It's starting to feel like the app is a sort of smoke and mirror show.

After a thorough review, it is safe to say that the code base isn't a total smoke and mirror show. There does seem to be a functioning data and networking layer, although there are no references to a live server instance anywhere. The assumption is that the controlled demoes are likely due to the backend being either incomplete, or not available on a live server instance. It seems the mobile devs have been using networking stubs to replicate the server environment on their local computers and likely for the demoes as well.

However, the software development process and the app itself are not without issues. Please find the score card and explanations below.

KŌSOKU
INTERACTIVE

SCORE CARD



AppRescue

DETAILED REPORT

Code Quality

Does the app follow best practices and leverage established architectural design principles?

Score: 8 out of 10

The code base does a decent job of compartmentalizing code and makes good use of extensions, inheritance, and composition to avoid duplicating code throughout the project. However, the code base also makes heavy use of RxSwift, a 3rd party framework for performing functional reactive programming. This approach has its benefits, however the learning curve can be quite substantial and mistakes can cause serious memory leaks that are difficult to track for inexperienced developers. If Acme Co. plans to eventually take the development in-house, this is something to take into consideration. In addition, the use of 3rd party frameworks is not without risk, as you are reliant on the maintainers of the 3rd party code to always be up to date with the latest OS releases. The Rx community is well-established and updates are published at a regular cadence, however Acme Co. should be aware of all 3rd party dependencies added to the app and should weigh the risk of each one independently. Concerns should be brought to the development team as quickly as possible to avoid time-consuming refactors.

It seems all calculations regarding marketplace sales are figured within the app's code base. I recommend moving these off to the server so that this behavior can be more closely controlled. When you're ready to release an Android version of the app, with the

current architecture, you'll be required to duplicate this calculation logic in both apps and keep them in sync. Worse yet, if you decide to change the transaction fees for your service, you'll have to coordinate updates to both apps at the same time and ensure everyone using the app are all upgraded to the latest version. Moving this responsibility to the server ensures every version of the app is always charging the correct amounts to your user base.

Another area of concern is the lack of localization and accessibility best practices. If Acme Co. intends to release this app with multi-language support, or if they wish to support vision or hearing impaired customers, then the development team should begin refactoring their code to include these features as soon as possible. The iOS operating system has built-in support for localization and accessibility, but the developer has to implement the APIs in order to get the best experience for each use case.

Performance

Does the app leverage crash reporting tools, have efficient load times, properly handle errors, and is it free from memory leaks?

Score: 8 out of 10

There are no memory leaks and the app loads in a reasonable amount of time. However, per comments above, the use of RxSwift could lead to difficult-to-detect memory issues if not used properly.

The developers did a good job of subclassing the system error classes, however stopped short of providing any meaningful failure reasons in their subclasses. So, instead of having

a multi-lingual supported error message that can be displayed to the end user, it's just an empty string. Perhaps the development team is waiting on Acme Co. to provide meaningful content to supplement the errors, however displaying an empty string can make it difficult to determine where content messages are missing. Rather, a deliberate placeholder string, something easily searchable within the code base, can help organize what errors have been accounted for and which ones still require content.

There doesn't seem to be any crash reporting implemented at this point, despite included dependencies for Google's Firebase SDK. Perhaps an oversight, but the crash reporting features in Firebase were never turned on. It is recommended that some level of crash and error reporting be added to the app so that defects can be addressed as soon as possible after discovery.

Documentation

Does the code base contain complete and accurate documentation?
Score: 0 out of 10

The code base is completely void of any documentation. There are only a handful of comments within the code base itself and nothing in the code repository that would give a new developer any idea about the app and what it does.

At the very least, the developers should add DocC style comments to each of their classes, extensions, and functions so that simple documentation can be auto-generated from the code itself. There should also be a default README.md file in the repository that

gives an overview of the app, architectures and coding conventions used, how to build and run the project, and who to contact if anything goes wrong.

Analytics

Does the app follow Apple & Google's guidelines for user privacy, track meaningful metrics, and abstract any 3rd party dependencies?

Score: 0 out of 10

There has been no effort to build in any form of analytics or behavior tracking within the app. While it might seem more important to get an app up and running as quickly as possible, stakeholders should give serious thought as to how they will determine the app's success and how that success will be measured.

Efforts to evaluate user behavior should be built in to the code base foundations rather than sprinkled in as an afterthought post the version 1.0 release. While the offshore team could help to make suggestions on the kinds of metrics to track, this should be coming from the stakeholders at Acme Co. Recommendations available in the Action Plan at the end of this report.

Testing & Automation

Is the code base adequately tested and are there mechanisms in place to automate testing and deployment?

Score: 1 out of 10

The code base presently only contains a single unit test that checks to see that all expected image and color assets exist, and nothing else. There are no UI flow tests, and no meaningful unit tests in the app.

There is no automation in the repository, which is unsurprising considering the development agency hasn't provided any regular builds to the stakeholders directly. However, as Acme Co. transitions to a more formal development process, automating builds and app testing will be important for keeping things running smoothly and efficiently. See detailed recommendations in the Action Plan at the end of this report.

UI/UX Design

Does the design follow well-established patterns, use controls appropriately for each platform, and support accessibility APIs?

Score: 4 out of 10

App taxonomy is somewhat disjointed. This is likely the reason for the slide out draw navigation in lieu of a more traditional bottom tab navigation. The problem with a slide out drawer navigation is that it interferes with the system-level built-in navigational shortcuts (like swiping from left to right to go back to the previous screen), as the slide out navigation drawer intercepts the touch gesture. This can be jarring for users accustomed to standard behavior in nearly all other iOS apps. The other issue with a slide-out navigation drawer is that it leads to lazy design decisions. Instead of weighing the choices of where to place access to certain screens against UX testing results, it's can be easier to

just throw it into a scrolling navigation drawer as you don't have to consider the space constraints of bottom tab navigation.

The app design also lacks in the basic fundamentals; contrast, repetition, alignment, and proximity. Everything is on a white or dark green background with a solid green for any accents and buttons. All text in the app is the same size and color, so it's very difficult to determine a hierarchy of design information. None of the content appears to be organized in any intuitive way, and trying to perform specific tasks within the app is a bit like swimming up stream. The app should be designed around specific outcomes you expect the user to perform, and those outcomes should be tracked with measurable metrics as mentioned in the Analytics section.

The app design also doesn't follow Apple's Human Interface Guidelines (HIG) and opts for building complete custom controls rather than use the built-in ones. Of course there is a time and place for custom controls, namely when the built-in controls don't provide the control needed for a particular outcome, but there are no cases in the design of this app that would merit the creation of custom controls. Opting to build your own controls means losing out on the built-in accessibility support and standard behaviors that users have come to expect when using an iOS device.

Additionally, there is no support for switching between light & dark mode or for landscape vs portrait orientation. This may be a conscious choice by Acme Co. or an attempt to save time and budget by limiting development to a single color scheme and orientation, however steps should be taken in the code base now to leverage the system appearance APIs so that support for dark mode and landscape orientation can be easily implemented once design has been approved to support it.

Quality Assurance

Are defects tracked and tickets well-written, is the QA workflow well-defined, are and the acceptance criteria for each feature easily understandable?

Score: 0 out of 10

Either the development agency employs their own QA team behind the scenes, or there is no formal QA process in play. Acme Co. really should consider injecting themselves into a formal QA role and request regular builds for testing. This will not only provide more visibility and transparency into the development process, but also help to address potential issues sooner as stakeholders won't be surprised by certain unexpected behaviors.

To do this correctly will require implementing some new tools and workflows and will demand the offshore team include Acme Co. more directly in their weekly interactions. A recommendation for managing the QA process included in the Action Plan.

Project Management

Is the scope of each sprint planned and communicated to the team, are user stories and tasks well-written, are technical requirements understood by both the dev team and stakeholders, and is the code change management system visible to the team and are code changes easily mapped back to the tickets in each sprint?

Score: 1 out of 10

Outside bi-monthly meetings between the offshore project manager and the stakeholders, there doesn't appear to be much of a formal project management plan in place for the app's development. Considering the stakeholders are all bilingual and fluent in Spanish, there should be no major cultural or language barrier issues to contend with. It is unknown as to why Acme Co. has left much of the project management tasks to the offshore development agency, but going forward, I recommend placing your own project manager, perhaps in a "Product Owner" role, and participate more directly in the offshore agency's planning and review activities.

ACTION PLAN

I. Crash reporting – have the development team follow the instructions here (https://firebase.google.com/docs/crashlytics/get-started?platform=ios) for setting up crash reporting in the app. Once crash reporting is set, you should have someone from your team added to the project dashboard and set up to receive notifications when crashes occur. You should have a triage plan in place to track crash issues, work them into development sprints, and release the fixes into production as quickly as possible to avoid a diminished user experience.

II. Description/errorDescription strings – the development team should create a custom `ErrorType enum` inside the `AcmeError` class and switch over that to determine the appropriate string to return. This will make it easier to group the errors by domain and to reason about error types instead of creating static instances of each error type.

    They should also take advantage of `CustomNSError` class to allow the passing of other data within the error class, which can be helpful in determining errors that could be recoverable and allow the code to self-correct when appropriate.

```
//AcmeError.swift

//Before

final class AcmeError: NSObject, LocalizedError {
    let message: String

    init(_ message: String) {
        self.message = message

        self.init()

    }


    override var description: String {

        return ""

    }

    override var errorDescription: String {

        return ""

    }


    static let unhandled = AcmeError("Unknown error")

    static let parsing = AcmeError("Parsing error")

}
```

```
    //After
    final class AcmeError: NSObject, CustomNSError,
LocalizedError {
        enum ErrorType: String {
            case unhandled
            case parsing

            static let defaultCode = 0
            var code: Int {
                switch self {
                case .default: return Self.defaultCode
                case .unhandled: return 100
                case .parsing: return 101
                }
            }
        }
```

```
        static var errorDomain: String {
            “AcmeError.error"
        }

        let errorType: ErrorType

        init(_ errorType: ErrorType) {
            self.errorType = errorType
        }

        var errorCode: Int {
            switch self.errorType {
            //could add logic here to determine if the
error should be end-user presentable or held internally
            default:
                return self.errorType.code
            }
        }
```

```
        var errorUserInfo: [String: Any] {
            //this user info object can be used to pass
helpful data about the error or process occurring during
the error to help the code recover from the issue
            var retval = self.localizedErrorUserInfo
            if let error = self.error {
                retval[NSUnderlyingErrorKey] = error
            }
            return retval
        }
    }

    var description: String {
        switch self.errorType {
            case .unhandled:
```

```
                return String(localized: “Unhandled
error”
            case .parsing:
                return String(localized: “Parsing error”
        }
    }
    var errorDescription: String? {
        switch self.errorType {
        case .unhandled:
            return String(localized: “A user-facing
message about the error that occurred”)
        }
        case .parsing:
            return String(localized: “A user-facing
message about the error that occurred”)
        }
    }
```

III. Localized content strings and string catalogs – text strings within the app that are displayed to the user should leverage the new `String(localized: )` methods and String Catalogs to manage the various languages that the app supports. The error messaging recommendations in the previous point demonstrate their use in code. Developers can follow the documentation from Apple for more details here: https://developer.apple.com/documentation/xcode/localizing-and-varying-text-with-a-string-catalog

IV. DocC comments & README.md file – the project git repository should contain a README.md file that explains the project architecture, how to build/run the project, and who to contact with any issues. It should be reviewed on a regular schedule to make sure it stays up to date. The rest of the code should include DocC style comments for developer-facing API explanations for how the code works. These style comments can be set up to auto-generate an easy-to-read document package highlighting the code base and how it works. More information can be found in Apple's developer documentation here: https://developer.apple.com/documentation/xcode/documenting-apps-frameworks-and-packages

V. Analytics wrapper for Google Analytics – the development team should create a separate layer within the app for handling metrics for the flows Acme decides it wants to track. This wrapper should be generic enough to handle custom message types related to the metrics and then forward those tracking points to Google Analytics. Creating this kind of wrapper around the analytics tracking service will make it easier to change to a different analytics service in the future, should Acme want to track their metrics data in a service other than Google Analytics.

VI. Analytics tracking/suggested metrics – considering the nature of the app, having a high number of users with items for sale in the market place will be important for attracting business. If there's no one putting items up for sale, then no one will stick around long enough for the idea to take off. Growth should be the primary metric you want to track, and measuring that growth against the effectiveness of your marketing campaigns will be important in determining the success of your efforts. Start by focusing on your growth strategies and make sure the metrics for measuring those strategies are baked into the code from the start.

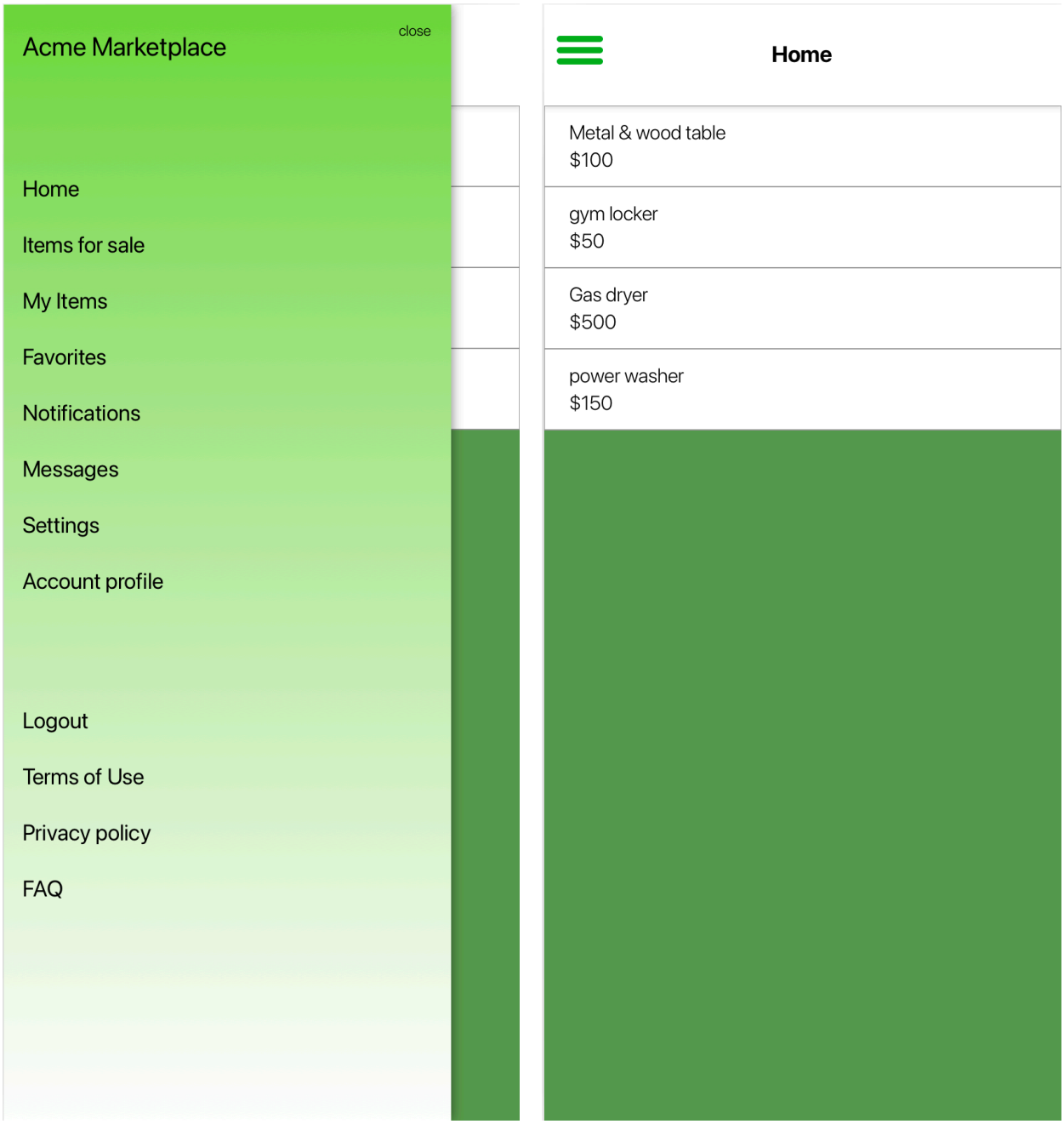
VII. Increased unit testing/UI testing, Fastlane scripts & GitHub actions for automated builds to TestFlight – if moving the payment calculations to the server is not possible at the moment, then I recommend adding unit tests to the app to ensure these formulas remain correct. If a code change accidentally removes your service fee, for example, then you've lost important revenue and will continue to lose it until a patch is released.

The only other logic factor you might consider for unit testing could be the mapping features showing items for sale within a particular radius. Although the built-in mapping APIs have methods for limiting map points within a radius, you'll want to make sure your local database fetching is accurately filtering out the results by latitude/longitude for the coordinates being stored. Regardless of whether the development team chooses to implement automated testing or not, they should definitely implement actions in the repository that try to build the project before it gets merged into the primary branch. This will ensure no one accidentally commits code that breaks the build. From there, you can add actions for deploying the build to TestFlight for internal testing, and even add web hooks to pull in feature and issue

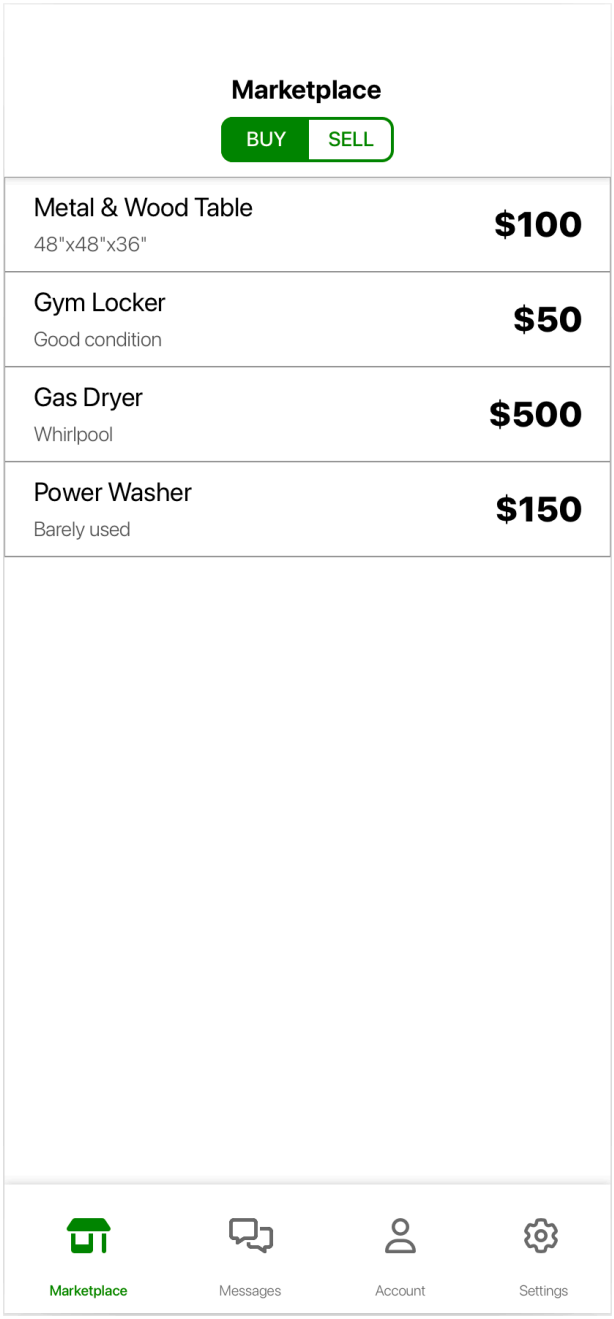
tickets from your tracking system to let everyone know exactly what is available for testing in the current build. I recommend the development team check out Fastlane (https://fastlane.tools/) for building out these automation scripts, and GitHub Actions (https://github.com/features/actions) for triggering tests and builds whenever code is committed or merged into the primary branch.

VIII. UI/UX design – I recommend the designer reconsider the continuity from screen to screen. They should decide on a standard margin/spacing around all items and the content from the edges of the screen. They should also define font styles for all the standard sizes of content through out the app (e.g. title, subtitle, caption, body, etc). The color scheme is also quite jarring. If the designer isn't sure how to mix colors in a way to elevate the content and not distract from it, you may want to hire another designer to create a new treatment. The attached screen layouts are just a hint at how to improve some of the content organization issues, but a complete redesign of the app is out of scope for this report.

Navigation – Current State



Navigation – Recommendations



As mentioned in the report section, the drawer style slide-out navigation menu makes it difficult for users to find exactly what they're looking for. Furthermore, the navigation menu taxonomy itself is overly segmented. For example, there's no reason to separate "Items for sale" and "My Items" when what you're really referring to is two sides of the same coin. It's the marketplace. Consolidate items in the menu and move them to a bottom tab-bar instead. Likewise, "Notifications" and "Messages" are really just an inbox of messages, either from Acme or from other users. Move them all into the "Messages" tab and allow the user to filter/toggle the display based on what they're interested in.

"Favorites" could just be a filter setting on the "Marketplace" tab, and "Home" is really unnecessary. At present "Home" is just a map view of items for sale nearby. But, honestly, when users go online to

shop for something, they're not often thinking about "What can I buy that's near me", they're after a particular item. They can filter their marketplace search results by distance, so the map view is really just wasted space. So, remove the "Home" item from navigation and just drop the user into the "Marketplace" tab when they start the app.

As for design, you need your designer to consider the hierarchy of information and use typeface treatments to show what's most important. The suggested layout above is by no means a recommended finished design as a complete redesign is out of scope for this type of report, but this should highlight some of the differences between the current "Items for Sale" section and this reimagined "Marketplace" tab.

IX. QA testing, defect tracking, and workflows for software lifecycle management — when it comes to testing features of an app, the stakeholders at Acme need to be extremely thorough in their description of the feature and what the criteria for acceptance is. These feature descriptions are traditionally referred to as "user stories" and QA will refer to the acceptance criteria outlined in the user story to validate that the software does what it should. Considering there isn't any formal QA happening on the Acme side of things, and assuming the offshore team is performing this work with their internal QA department, I recommend Acme assign a few of their own people to test the app as soon as possible.

When it comes to tracking the features and defects, since you're already using GitHub, it would probably be easiest to just adopt their built-in project tracking features. Once you're comfortable with the development lifecycle process, you can then determine if there are other tools out there that might be a better fit for your organization. GitHub

project tracking information can be found here (https://docs.github.com/en/issues/planning-and-tracking-with-projects).

Finally, regarding the software development lifecycle, you need to work with the design & development team on a regular schedule. At the start of each development cycle, you should determine what features to prioritize and try to limit turn around time to within 1~2 weeks. Write detailed tickets in the project tracking system that explain how the feature should work and what constitutes a "completed" state. Let the developers worry about the implementation details, and don't rely on designs as a crutch. Be as explicit in the written descriptions as possible. As developers complete their tickets, new builds should be deployed as each ticket is merged into the primary branch. QA can then use these builds to test that the features work as expected. At the end of each 1~2 week time block, reevaluate the remaining tasks and see if the priorities still hold true. Check to see if any defects need to be addressed before moving on. Establish regular communication with the dev and QA team leads to make sure there are no bottlenecks to the development flow.

X. Project Management – at the very least, Acme should assign an internal team member to be the "Product Owner" and primary point of contact for all decisions about the app. The Product Owner should participate in regular planning meetings with the offshore development team to ensure features are being prioritized appropriately and that any roadblocks that would cause delay are remedied as quickly as possible.