COMSE6156: Topics in Software Engineering
Final Project Report

Nazli Tuncer | nt2320 | nt2320@columbia.edu

Can Cross-Platform Dev. Mimic the Native Experience?

1. Aim of Project

This project builds on my midterm paper, which evaluated native and cross-platform mobile development. The main outcome of my analysis was that while cross-platform development introduces convenience to developers (write once, run everywhere), it lacks in two areas: user-friendliness of interface design and performance. During my research I came across React Native, which is a solution proposed by Facebook that claims to deliver an improved cross-platform experience. The framework claims to mimic the entire feel of a native application and make up for disadvantages encountered with other cross-platform tools. Therefore, for my final project I decided to investigate if these claims are true by building two versions of the same mobile application – one in Swift/Xcode (native) and one in React Native (cross-platform).

The main purpose is to evaluate if React Native, a cross-platform tool, can mimic the native experience. This evaluation will consider the following parameters:

- **User-friendliness**
  After building each app, conduct survey with developers & users to:
  - See which app users prefer
  - See if developers with mobile experience can recognize which app is which

- **Ease of Development**
  While building the apps, monitor:
  - Clock time taken for developing each version of app
  - Number of lines of code in final version of each app
  - Availability of online sources

- **Performance**
  Once apps are built, use the "Apple Instruments" tool packaged with Xcode to calculate:
  - CPU Usage
  - GPU Measurements
  - Memory

2. What I Delivered:

- Two iOS "To Do List" applications
  1) Native, built with Swift
  2) Cross-Platform, built with React Native

- Case Study Results with Developers and Users
    - Investigated if developers can differentiate between native and cross-platform versions of the app
    - Investigated which app users prefer

- Development Experience Findings
    - Clock time taken to build apps
    - Number of lines of code in final version
    - Availability of online resources

- Performance Analysis Results (Comparison plots)
    - CPU Usage
    - GPU Measurements
    - Memory

3. How to Access Deliverables:

3.1 GitHub Repo

GitHub Link for Native Code:
https://github.com/nazlituncer94/Swift-To-Do-List

GitHub Link for React Native Code:
https://github.com/nazlituncer94/ReactNativeToDoApp

The above links contain all code written to build the mobile apps. Instructions on how to run the code are included in "README.txt" (for each repo) and will also be referenced here.

Instructions on How to Run Apps
The following instructions are for Mac users.

A. Swift Application
Pre-requisites: Install the latest version of XCode from the Mac App Store to be able to run the app. XCode comes with an iOS simulator.

1. Download (or fork/clone) GitHub repo using link provided and open the "To Do List" folder in Xcode.

2. Click the "Run" button on the upper left corner. This will first build your application and then initiate the iOS simulator so you can view the app.
   Notes:
    - There is no "main file/function" in Swift's code design so you can run without having to select a specific file within the folder.
    - You can change which iOS device is shown on the simulator by clicking drop-down menu to the right of "Run" button (e.g. iPhone SE, iPhone 6s, etc.)
    - The first time the simulator is run it will take a substantial time to load. Please allow ~ 15 minutes.

3. The "To Do List" application is running. Add reminders and delete completed tasks to see the app at work.

B. React Native Application

Pre-requisites:

- Install Homebrew by pasting the following at a Terminal prompt:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install)"
```

- Install dependencies by running following commands in a Terminal:

```
brew install watchman
brew install node
```

    - **Watchman:** Tool by Facebook that improved the app's performance and watches for changes in the file system.
    - **Node:** Installs npm, which is needed to download React Native command line interface (CLI) and package manager for Javascript.

- Install React Native CLI by running following commands in a Terminal:

```
npm install -g react-native-cli
```

- Install yarn, a package manager for Javascript, to simplify the dependency management workflow:

```
npm install -g  yarn
yarn
```

- Download the latest version of XCode from the Mac App Store. React Native does not have a separate simulator but instead works with the iOS simulator in Xcode so this step is required to view the app on your laptop.

1. Download (or fork/clone) GitHub repo using link provided. Open a Terminal prompt and change your directory to where you've downloaded the repo on your laptop, selecting the "workshop-react-native" folder ("`cd user/filename/workshop-react-native`").

2. Within the selected directory, type "`react-native run-ios`". This will run the Xcode iOS simulator for you to view the app (Xcode does not need to be running in the background as the simulator can be initiated separately).

3. The "To Do List" application is running. Add reminders and delete completed tasks to see the app at work.

3.2 Final Report

This report includes screenshots of the working applications as well as findings for the case study conducted to evaluate user-friendliness, notes regarding ease

of development and plots to display performance analysis. Please see Section 6 for results and findings.

3.3 Demo

I displayed a live demo for the mobile applications created on Wednesday, May 3rd at 11:30 AM. Two iOS simulators were used to simultaneously display both "To Do List" applications. Tasks were added and deleted to show that all of the components work. In addition, case study, performance analysis and ease of development findings were summarized during the presentation.

Note: iOS simulators were used as Apple requires developers to enroll in their Apple Developer Program ($99/year) to test applications on physical devices.

4. List of External Software Used

- XCode
  https://developer.apple.com/xcode/
- Swift
  https://developer.apple.com/swift/
- React Native
  https://facebook.github.io/react-native/
- Javascript
  https://www.javascript.com/
- Homebrew
  https://brew.sh/
- Watchman
  https://facebook.github.io/watchman/
- Node.js
  https://nodejs.org/en/
- Yarn
  https://yarnpkg.com/en/
- Apple Instruments
  https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/

5. Tutorials Used:

One of the important goals for this project was for me to become familiar with iOS and cross-platform application development. As a beginner in this field, I followed various tutorials to be able to create both mobile apps.

The following tutorials were used in creation of the mobile applications:

- Treehouse | An Absolute Beginner's Guide to Swift
  http://blog.teamtreehouse.com/an-absolute-beginners-guide-to-swift

- Swift Guy
  How To Create a TableView in Xcode 8
  https://www.youtube.com/watch?v=fFpMiSsynXM

How To Create a To Do List App in Xcode 8 *
https://www.youtube.com/watch?v=LrCqXmHenJY
* This tutorial guided the structure of the Swift To Do app (also referenced in README).

- React Native Documentation | Getting Started With React Native

https://facebook.github.io/react-native/docs/getting-started.html

- Egghead.io Tutorials | Build a React Native Todo Application

https://egghead.io/courses/build-a-react-native-todomvc-application

- Workshop to Teach React Native by Doing a Todo App **

https://github.com/vtex/workshop-react-native
**This tutorial was used to initiate my React Native project (also referenced on README). The repo was cloned as directed on workshop instructions, creating the necessary code structure (e.g. separate ios and android folder). The final app of the workshop class is not the same as the one produced as this was used for guidance. Additional tutorials/documentation provided here was used for modifications.

6. Results and Findings

6.1 Screenshots of Working Applications (with References to Code)

A. Swift (Native)

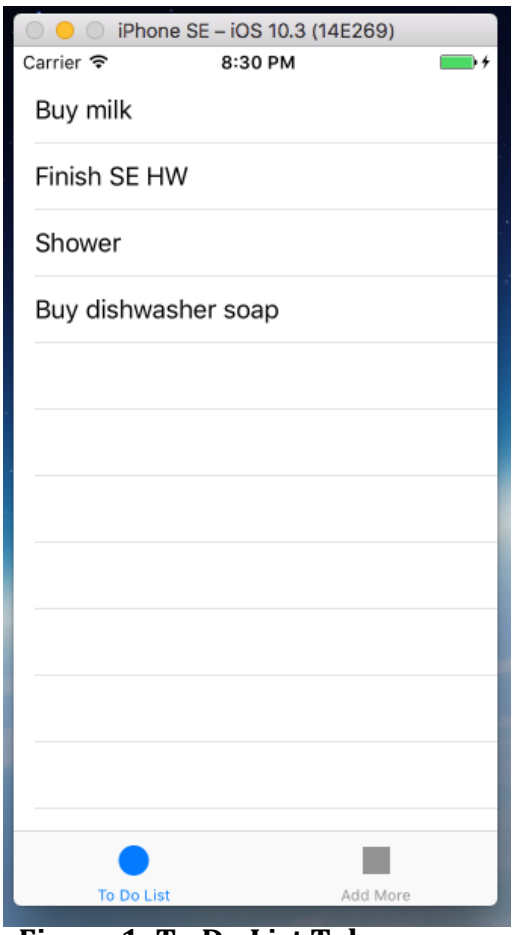


Figure 1: To Do List Tab

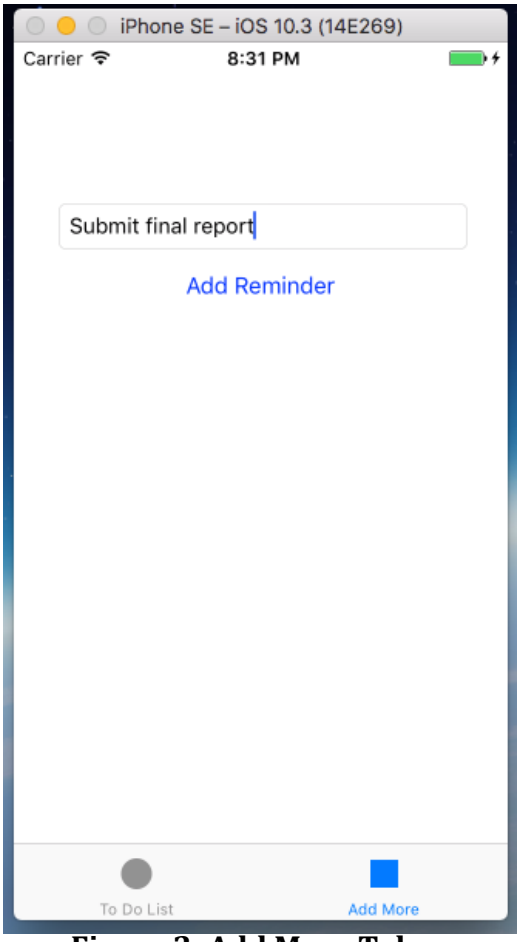


Figure 2: Add More Tab

Swift creates a "Tabbed View" application that is in line with the iOS design specifications (i.e. this tabbed view "feels right" to the user as it mimics the style of other apps on the iPhone).

Figure 1 is the first tab that opens up with the app. It is a listview of items to do added by the user. Figure 2 is the second tab that can be switched to by clicking the square on the bottom. It has an input box for the user to add more items to the list.

The code structure of Swift dedicates a file to each tab view:

- FirstViewController.swift → Code for first tab ("To Do List")
- SecondViewController.swift → Code for second tab ("Add More")
- Main.storyboard → Visual layout of application. Provides conceptual overview of app (see Figure 4 below)

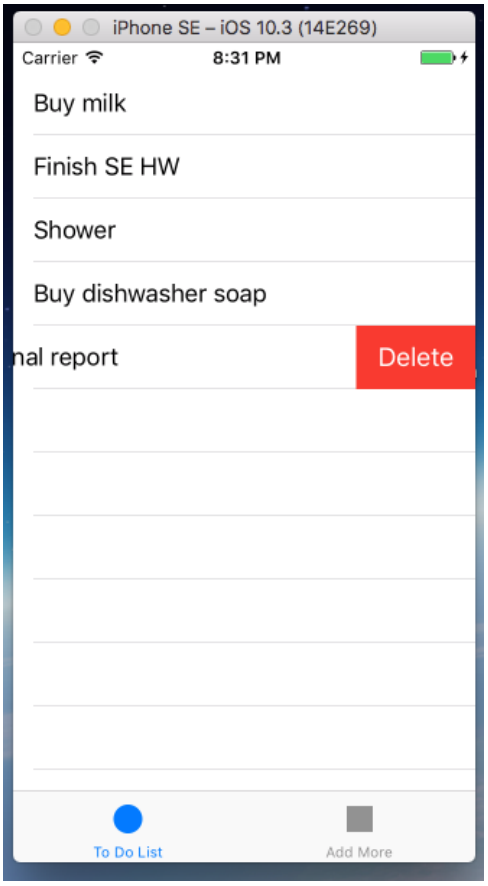


Figure 3: Delete Completed Task

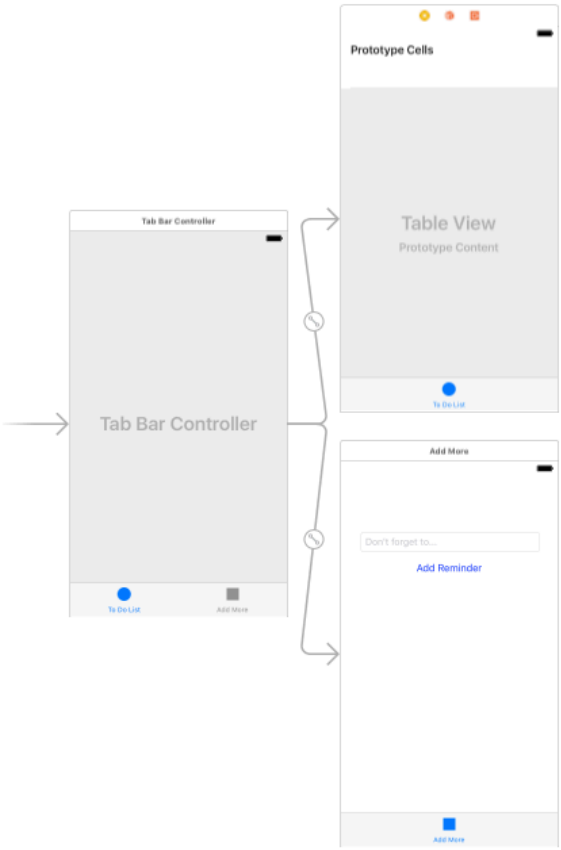


Figure 4: Main.storyboard

The user can delete a completed task from the list by swiping left on an individual item (see Figure 3). This is an inherent capability of the native app.

B. React Native (Cross-Platform)

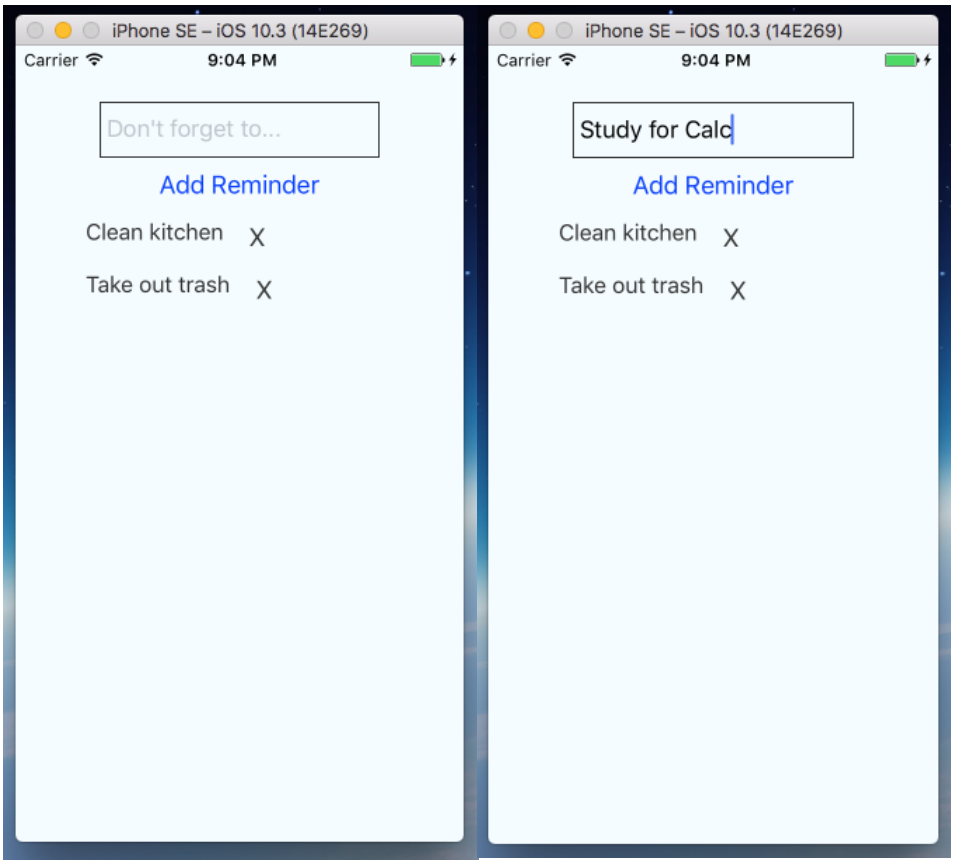


Figure 5: Single View To Do List App

React Native creates a single view application where one tab contains both the input box and the list of items to do. As swiping left to delete is a native feature, users need to click the "X" next to each item to remove completed tasks.

The code structure of React Native is completely different than that of Swift. The code written is not associated with each tab or element. Instead, a common code in Javascript is written. It's common as this code can then be compiled either to an iOS app or to an Android app depending on the run command selected ("run-ios versus run-android"). There are two separate folders – "android" and "ios" – within the project folder that gets populated after the build request. The executable file gets stored in the associated folder. For an iOS application, this file is named "project_name.xcodeproj". This file cannot be manipulated directly within Xcode; it can only be opened inside Xcode to run with a simulator. To make a change, the developer needs to edit using React Native CLI and build/run with run-ios (creating new executable file).

The code structure of the React Native app:

- Common code → Found in the "components" folder within app
  - AddItem.js
  - Filter.js
  - Row.js
  - ToDoList.js

- ios folder & index.ios.js → Contains iOS specific code after compile
- android folder & index.android.js → Contains Android specific code after compile

6.2 Ease of Development

- Clock Time Taken to Build Apps

  Preparation Time:
  The preparation time will not be taken into account for the overall clock time but is included so that future developers set aside a significant portion of time for installation as well as keep in mind potential incompatibilities with OS version and tools used (this was a challenge I faced see section 8).

  Updating macOS to 10.12 – 4 hours
  Downloading Xcode – 6 hours
  Upgrading RAM to 8GB – 2-3 days (shipment time of new RAM)
  Downloading all dependencies – 5 minutes (brew installations are very quick)

  Apple requires the latest version of macOS to use Xcode, which is a necessary tool for both Swift and React Native. The iOS simulator is computationally intensive so lower than 8GB RAM freezes/crashes the entire system.

  Build Time:
  Swift – 4.5 hours
  React Native – 7 hours

- Number of Lines of Code in Final Version of Apps

  Swift – 127 lines
  React Native – 537 lines

  This count deducts white space/empty lines and comments.

- Availability of online resources

  Swift:
  - Greater number of tutorials available.
  - Wider audience (more forum/stackoverflow posts, easier to debug problems).
  - Syntax of language changes frequently (approx. every 6 months) so some older documentation found required modification. However, changes are minor and fixes were easily implemented using Xcode's debugging tools (e.g. UIViewControl becomes UIViewController in latest version of Swift).

  React Native:
  - Attracts more of a niche audience (not as widespread use as Swift).
  - Availability of online sources is limited.

- Drastic changes have occurred within the tool so documentation/tutorials found from 6 months ago or so had been deprecated and were not useful.

Winner: Swift/Native
I found Swift to be much easier to work with than React Native. The app took half the time to build and the total number of lines was $1/5^{th}$ of how long the React Native app is. This is because in React Native there is "common code" and ideally, what is written should be able to compile to an Android app easily (even if only iOS is targeted). The storyboard feature of Swift made it easier to conceptualize my app as I could use shapes/diagrams to plan out the design. As React Native is a CLI (no IDE), it was more difficult to finalize the UI design (e.g. had to manually enter width, height of items).

6.3 User-friendliness

For the case study, the two apps built were simultaneously shown to users and developers by running two instances of the iOS simulator on my laptop.

Participants:
- 20 developers (15 had experience with mobile development)
- 15 Users (exclusively users; no programming background)

Interview Questions:
1- Would you be able to recognize a native app from a cross-platform one? (D)
2- Which app is the native one? (D)
3- How did you recognize it was the native one? (D)
4- Which app is more aesthetically pleasing?
5- Which app is easier to use?
6- Do you recognize lags during your use? If so, with which one(s)?
7- Overall, which app do you prefer?

* D indicates questions asked only to developers. The rest was asked to all participants.
** In addition to these questions, the following demographic information was collected for developers:
- Student or working/professional
- Level of experience with mobile development (familiar versus proficient)
- If working, company size and for how long

*** All users interviewed had no programming/CS background and were college students in the age range of 19-24.

Results:
- All the developers who claimed they could recognize the native app successfully picked the correct one. All of these developers were working in industry and identified themselves as highly experienced/proficient with mobile development. (Numerically, 13 developers picked the correct app.)

- These developers provided the following reasons as to how they could recognize the native app:
  - When dragging the list from top to bottom, the text in native app has a specific bounce that is not seen in cross-platform alternative.
  - The left swipe delete option is solely a native feature.
  - When typing in text field, the keyboard pops up quicker for the native app.
  - The text field built with Swift has round edges, which is in line with iOS design specifications. The cross-platform alternative has rectangular edges.

  The results indicated in red are native features that cannot be mimicked by cross-platform alternatives (even with React Native). Developers mentioned in the survey that the last bullet regarding shape of text field is indeed something I could have fixed/accounted for. As I did not want to alter the application mid-survey, I kept the text field in its original form and mentioned this error on my end in my findings.

- 7 developers said they could not tell the difference and were not able to identify the native app. Only 2 of these developers had experience with mobile development but they identified themselves as beginners. These developers were also students/not working in industry.

- The answers to question 4, 5 and 7 were consistent (i.e. a user picked the same app for aesthetically pleasing, easier to use and overall preference).

- Of the 35 people interviewed, 24 people said they preferred the React Native app overall and found it to be more aesthetically pleasing as well as easier to use. This was an interesting find as the native app was expected to provide a better user experience.

- Of the 35 people interviewed, 19 people recognized lags with the native app whereas there was no lag recognition with the cross-platform alternative. This was an interesting find as the native app was expected to have better performance.

  Winner: React Native/Cross-Platform
  The majority of participants surveyed preferred the React Native app, saying it provided them with a better user experience (both UI design and performance wise). It is seen that React Native's claims about improved user experience are recognizable. Experienced mobile developers, however, were still able to successfully recognize which option was the native app.

6.4 Performance Analysis

"Apple Instruments", a tool packaged with Xcode, was used to evaluate the performance of each app (one app/iOS simulator at a time). The tool can be launched within Xcode by selecting Xcode (upper left corner) → Open Developer Tools → Instruments. Once launched, a list of instruments (i.e. specialized tools)

is displayed. The developer can select associated instrument according to what he/she wants to measure.

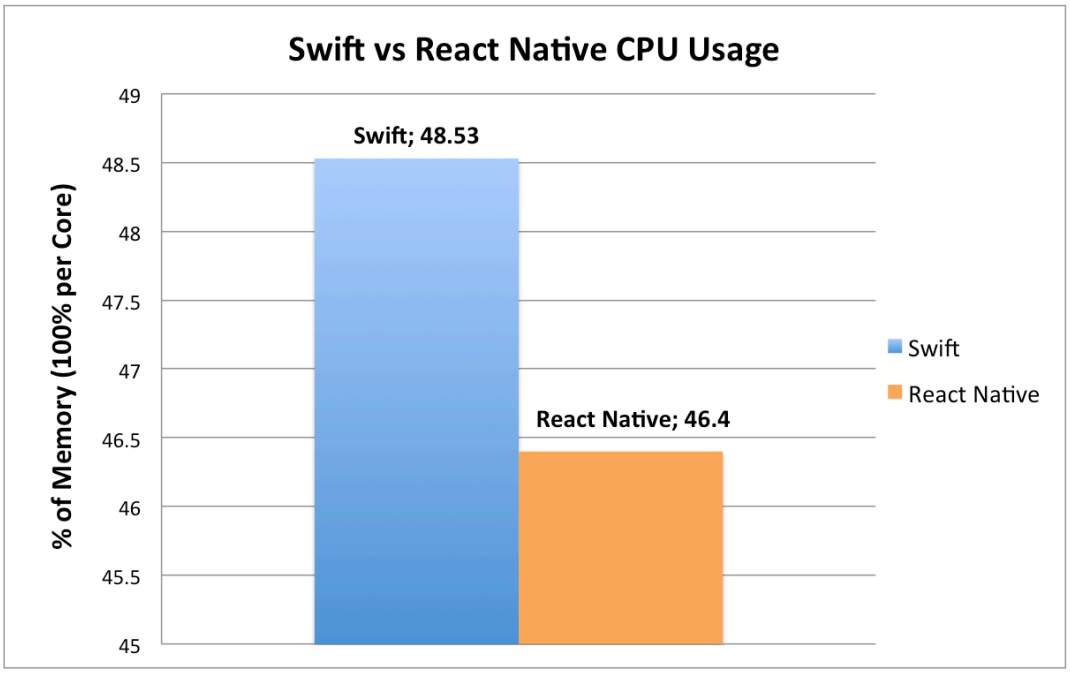
The following instruments were selected (with associated measurement):

- Time Profiler Instrument → Measures CPU Usage
- Core Animation Instrument → Measures GPU Usage
- Allocations Instrument → Measures Memory Allocated

In order to do performance measurements, the "To Do List" apps need to be used while the selected instrument is running. Specifically, items were added and deleted from the list while the tools were running (at these instances spikes were observed indicating successful measurement).

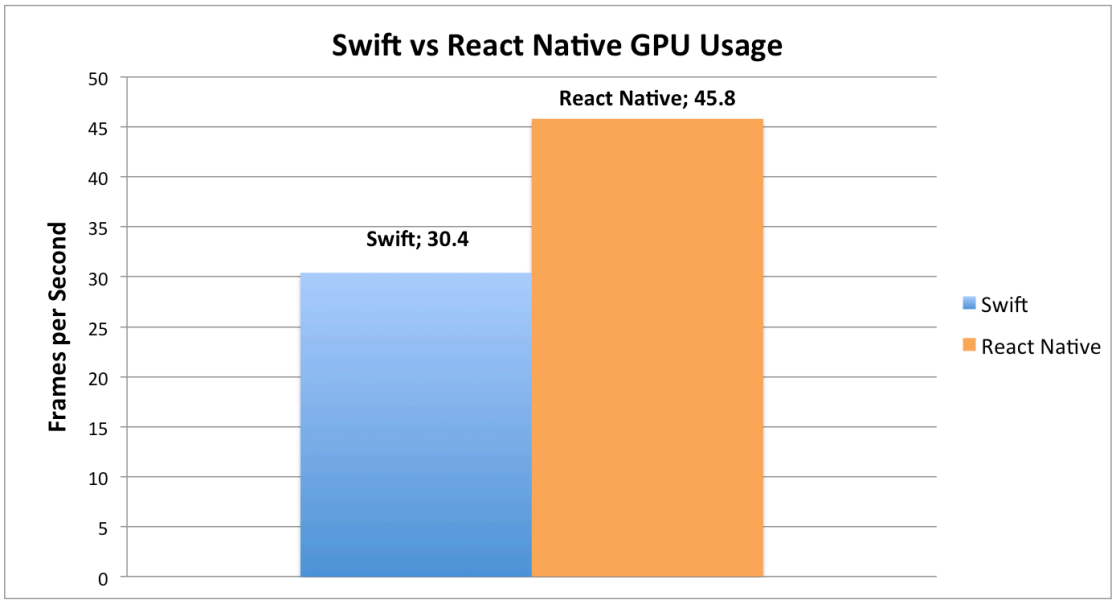
Results

- CPU Usage



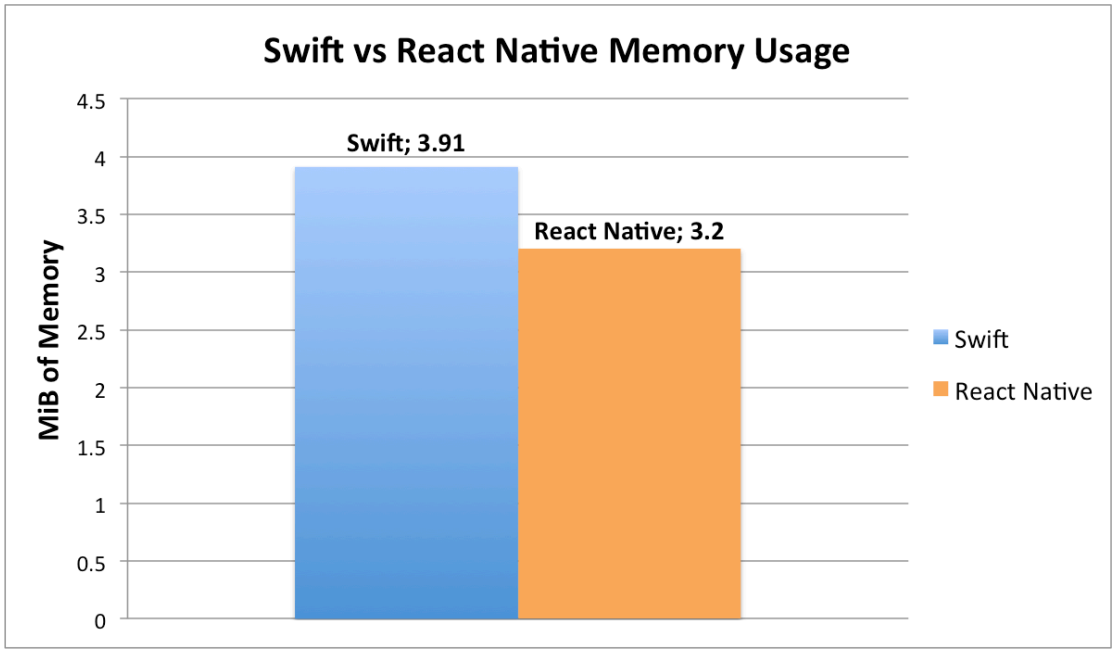
The lower the % of memory usage, the more efficient the application is (i.e. lower value is better). As seen in plot above, React Native is more efficient by 2.14%.

- GPU Usage



The higher the frame rate, the "smoother" the application feels to the user (i.e. higher value is better). With low values of frame rate, the user can recognize lags. As seen in plot above, React Native is superior in this category by running 15.4 frames/second higher. The case study revealed that users were able to pick up this difference in frame rate, as many observed lags with the native/Swift application.

- Memory Usage



As seen in plot above, React Native is superior in this category by using 1.91 MiB less memory. While this is a fairly small difference, it is interesting to

note that React Native used less memory even though the React Native code was approximately five times longer than the Swift code written.

Winner: React Native/Cross-platform
In all three categories evaluated for performance, React Native proved to be superior. The findings were in line with case study results where users recognized lags with the native app and not with the cross-platform one. As mentioned before, this is an interesting finding. It is seen that React Native's claims about improved performance is true. Many optimization measures are implemented with React Native (e.g. Watchman), which drastically improve the app's performance.

7. What Did I Learn? (Focus on what wasn't already known)

Coming into this project, I did not have any experience with iOS and cross-platform development. I had always been interested in mobile development and wanted to gain experience in the area. Now that the project is completed, I feel confident about the subject matter as I became familiar with the necessary tools and gained substantial knowledge going over documentation. Specifically, I learned two new languages – Swift and Javascript. I am now comfortable navigating the React Native CLI and Xcode (tools I had not used before). I can successfully launch the iOS simulator, which will allow me to easily implement and test apps in the future. I learned how to conduct performance analysis for mobile applications. Specifically, I can now utilize Apple Instruments, which means I can better understand and optimize the behavior of any future apps I build. I saw that while building an app with React Native is much more difficult (compared to Swift), the outcomes are very rewarding in terms of performance and user experience. If I want to look into Android development as well, my experience with React Native will come in handy as I can compile an Android application using the "write once, run everywhere" principal. Lastly, I gained experience on how to conduct a case study (e.g. timing, how to reach out to participants, best way to record results), which will come in handy for future academic research.

My biggest takeaway from this project was understanding the overall architecture of mobile development. This knowledge is applicable beyond the scope of the tools used in this project and will allow me to further advance in this area of interest.

8. Unexpected Problems Encountered & Modifications to Original Plan

- Building three To Do List Applications

The initial plan was to build a third application as well using Xamarin and do pairwise comparisons (i.e. Xamarin vs. React Native, Xamarin vs. Swift and React Native vs. Swift). The idea was to contrast React Native to alternative cross-platform tools. This plan had to be modified due to an incompatibility error encountered. As mentioned before, Apple requires the latest version of macOS to

use Xcode, which is a tool necessary for both the Swift and the React Native application. To be able to use Xcode I upgraded my macOS to 10.12 and this update took approximately 4 hours. After the update, I was unable to use Xamarin and the app crashed after numerous trials. I found via forums that other users experienced similar problems and that Xamarin is not an "Apple-friendly" tool. The latest macOS update introduced incompatibility issues therefore I could not build the third application (resolving the issue would require reverting update, which would block access to the other two apps built). Next time I will make sure to research potential incompatibilities in advance, as this problem could have been avoided by initially picking another cross-platform tool.

- Code Reuse

Swift and Javascript are known to be similar languages therefore, I was initially planning to see if any of the code used for one of the applications could be reused in the second version. There are open source transpilers in development (such as ShiftJS) that can convert Swift to Javascript. In the end, there was no possibility for code reuse as I found out the Swift code design is very different than the Javascript code design (see section 6.1). The two languages can be "translated" to each other, as their syntax is highly similar but converting an entire application is much more complex than manually changing variable declarations.

To make up for these modifications in my plan, I interviewed users as well in my case study (initially the plan was only developers) and added three main categories for performance analysis (CPU, GPU and memory usage).