PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS*
Instituto de Ciências Exatas e de Informática

A Proposal of a Multi-Platform Application Architecture with React Native

Diogo Rafael Perillo[1]
Pedro Alves de Oliveira[2]

Abstract

Before developing an application, it is essential to consider that users are spread across multiple platforms, creating a requirement to deploy the app to as many operating systems as possible. However, in most cases, individual and financial resources and a short deadline are typical constraints that prevent this from happening. These challenges lead companies and developers to choose one or two platforms in order to start distributing their products. The objective of this project is to design and implement a framework for building cross-platform applications with React Native and other JavaScript libraries, using a single code base that will be running on Windows, macOS, Linux, Android, iOS, and Web.

Keywords: React Native. React. Cross-Platform. Multi-Platform Application

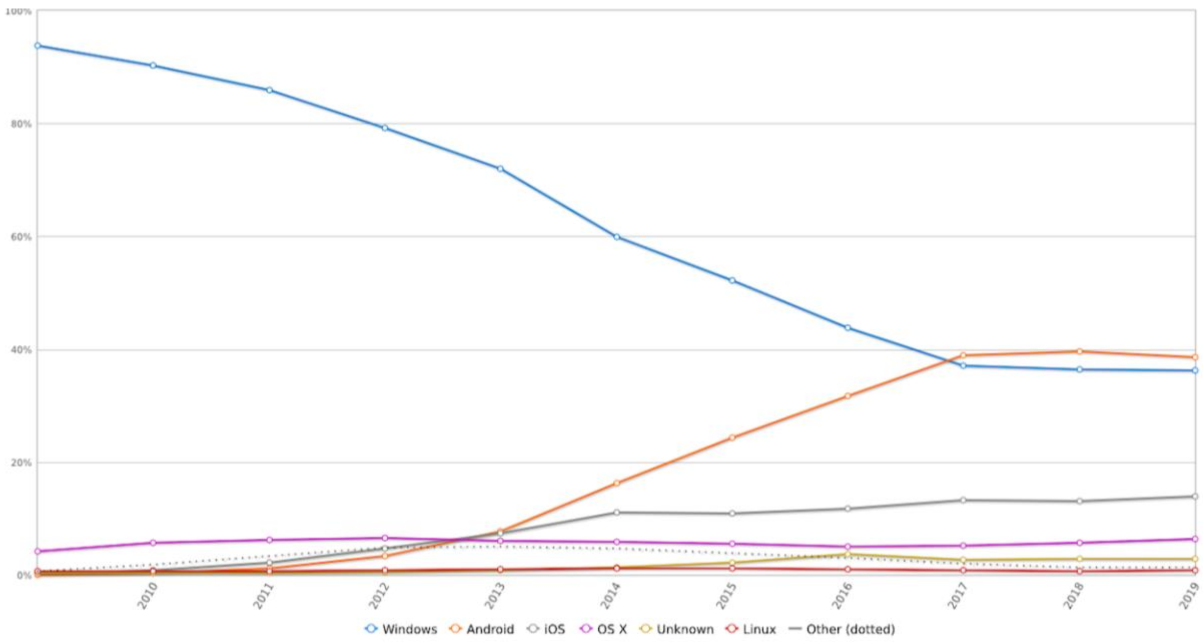
* Article presented to the Instituto de Ciências Exatas e de Informática of Pontifícia Universidade Católica de Minas Gerais.
[1] Student of Information Systems Program, Brazil – diogo.perillo@gmail.com
[2] Professor of Information Systems Program, Brazil – pedroalves@pucminas.br

1 INTRODUCTION

Building an application nowadays is much more complicated than it was a couple of decades ago. Today's applications run on multiple devices and operating systems, are reliably connected to the internet, storing and processing large amounts of data. In the beginning of this millennium the majority of personal computers were based on the Windows operating system, there were already a wide variety of programming languages a developer could use to build an application, but most of them would be compiled to run on the same platform and be distributed to almost all personal computer users.

Since 2009, more operating systems, including other desktop platforms like MacOS and Linux, and mobile platforms like iOS and Android, became popular around the world, this new distribution of users and platforms has resulted in a new need for applications to be released for those different operating systems. Figure 1 illustrates that scenario.

Figure 1 - Operating System Market Share Worldwide



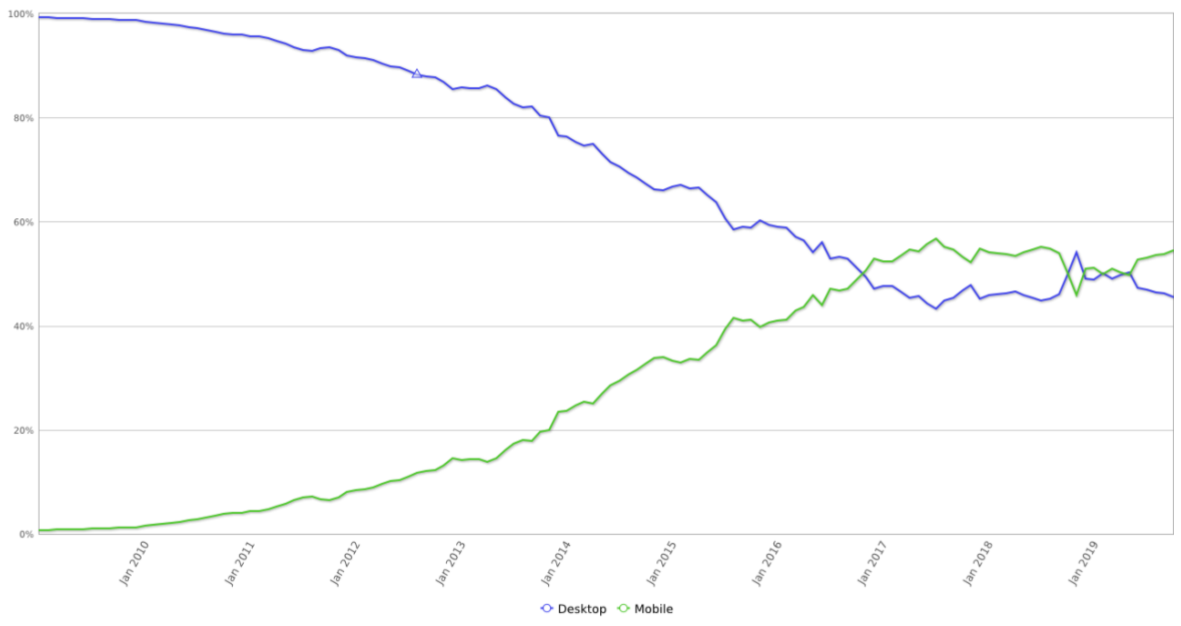
Source: StatCounter – GlobalStats – October 2019

https://gs.statcounter.com/os-market-share#yearly-2009-2019

Web 2.0 brought a new paradigm where the applications are hosted, no longer running in the user's computer, but on a server. It can be accessed by any internet browser from any platform. The distribution and infrastructure are centralized, breaking the barrier of having to build for, and install on multiple operating systems. Since then, developing a web application or migrating desktop applications to the cloud has been one of the most popular decisions made by software development companies.

In 2007, the first Apple iPhone was announced and released, changing the way people were using cell phones. Not too much later, Google announced a new mobile and open-source operating system, the Android OS, which started the competition with Apple for the users of the recently invented smartphones. In the last ten years, the number of mobile devices started to increase drastically (Figure 2), creating a new demand for companies, governments and universities to adjust to the new reality.

Figure 2 - Desktop vs Mobile Market Share Worldwide



Source: StatCounter – GlobalStats – October 2019

https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#monthly-200901-201910

Along with the smartphone devices and mobile platforms also came the application marketplaces and new popular programming languages. Building a mobile application for at least the two major platforms started to become mandatory for a software project to succeed. In this context, in order to build an application that will execute in Android and iOS, using their native architectures, at least two codebases are necessary, with the probability of a minimal, if any, code sharing between them. Besides that, users still expect to be able to use those applications on their personal computers, which makes a web and desktop application still significantly important.

Having to support so many platforms increases, even more, the number of codebases and maintenance needed for an application. All this complexity makes a software development project unfeasible for most companies with small budgets and tight deadlines, forcing them to either choose one or two specific platforms or search for nonnative solutions. Several libraries and software architectures were created seeking to solve this issue. Some of them targeted mobile and web platforms, while others were targeting desktop and mobile platforms (Ionic, Adobe Air, Flutter, and others).

1.1 Objective

The main objective of this paper is to propose an architecture for building applications that will run on as many platforms and operating systems as possible. The specific objectives are to accomplish it with a single codebase, leveraging more technical resources than individuals and with a low development cost achieve maximum efficiency, writing components once and running everywhere.

2 METHODOLOGICAL PROCEDURES

As mentioned before, we have the objective to release an application on all major platforms, using a single codebase. Before exploring our solution, we will mention some of the most popular tools and languages used to build cross-platform apps.

With C# and Visual Studio IDE it is possible to write applications for Windows, web and mobile (with Xamarin) (MICROSOFT, 2019), but even though you can share a good amount of code and logic, the user interface components will need to be different for each specific platform. In Java we could also build for web, desktop with IDEs such as Eclipse (ECLIPSE FOUNDATION, 2019) and NetBeans (APACHE, 2019) and Android with Android Studio (GOOGLE, 2019), but the same problem occurs for UI components and multiple codebases would be required.

Google released a library called Flutter (GOOGLE, 2019) for building cross-platform and native mobile and web applications that provides fast development and great performance. However, it is not available for desktop at the moment and it should be considered a new technology, still building a solid community (SKUZA, MROCZKOWSKA e WŁODARCZYK, 2019).

According to a recent research from Stack Overflow (STACKOVERFLOW, 2018), JavaScript became the most popular programming language in 2018. With JavaScript it is possible to write code that will run in a server or build interfaces for web, desktop and mobile devices. Because JavaScript is so popular, there is a variety of modules and resources available along with a great and active community and many companies investing in tools and infrastructure to support JS applications.

For user interface solutions for example, we can find a large number of libraries (GITHUB, 2019), including the most popular Angular and React JS. With both technologies, it is possible to build for web, and when combined to Ionic, it is possible to build mobile applications. However, even having access to native features, Ionic uses a Hybrid-Web approach, rendering UI components in web views which can cause performance loss and other issues. On the other hand, React can also be combined with React Native, making it is possible to build for Android and iOS. Differently than Ionic, React Native uses a Hybrid-Native approach, where the UI components will be translated to native elements, taking the advantages of each operating system performance (DOSSEY, 2019).

Each one of the resources and technologies described above have their pros and cons. We have decided to use React and React Native not only because it fulfills our needs in supporting multiple platforms, but we are also considering the large demand for professionals (GREIF, BENITTE e RAMBEAU, 2019), the increasing adoption of React and React Native in production applications (Facebook, Instagram, Skype, Uber and others), (FACEBOOK, 2019) and the exponential learning curve of these libraries. React has a small API to work with, so you can spend more time familiarizing yourself with it, experimenting with it, and so on. The opposite is true for large frameworks, where all your time is devoted to figuring out how everything works (BODUCH, 2017).

2.1 The Architecture

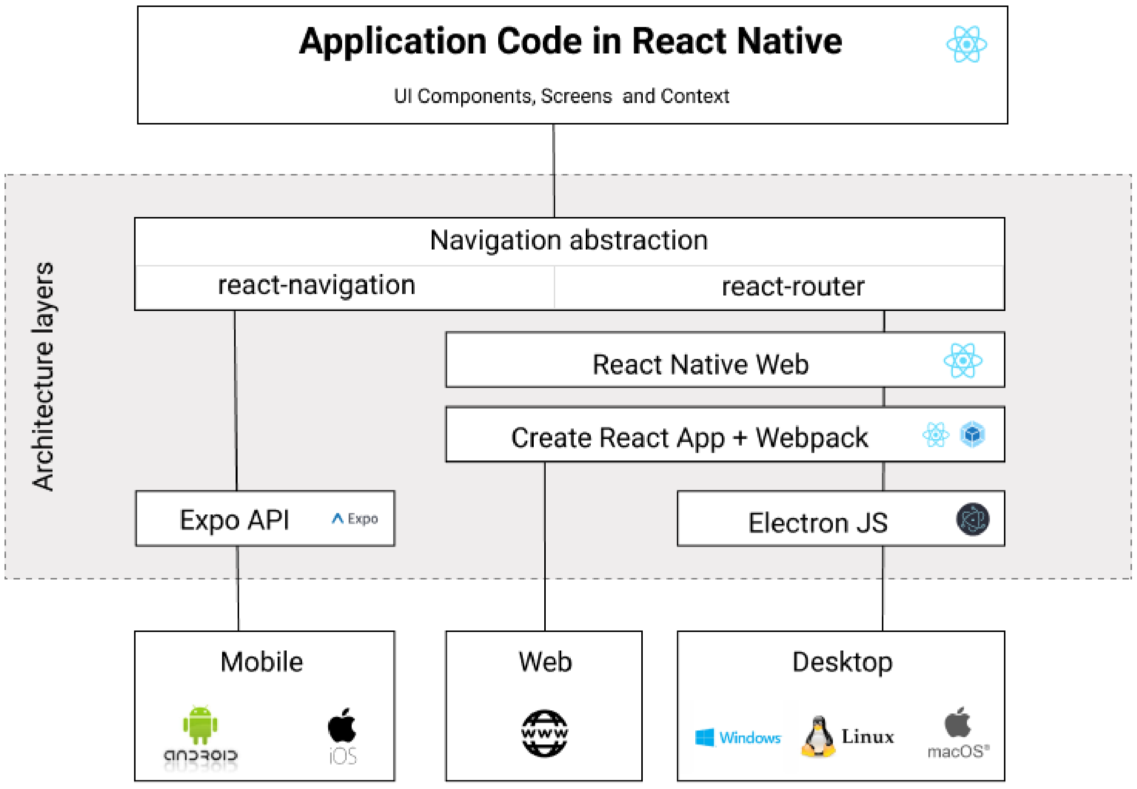
The proposed architecture is composed of a collection of JavaScript libraries that when combined provide a simple way to start developing a cross-platform application using React Native. The architecture code was designed, planned and written by one senior engineer in about eighty working hours, but the expertise and initiative to build the architecture came from years of previous professional experience in a startup.

In 2016, we were challenged to rewrite an application in order to grow the number of users in a small period of time with limited number of engineers. It was a messaging application, so the need to release it to as many operating systems and platforms, including mobile, web and desktop was inherent. We couldn't find a solution in the market that would **address** all of our needs, so we decided to explore ways to use React to accomplish our task. We were already using React and Electron JS to build the desktop application and started to rebuild the iOS application in React Native in order to have it deployed to Android as well, but even having the same logic to connect to the backend and the same requirements in desktop and mobile, it was still a problem to reuse UI components in both places since React Native wouldn't work directly on web or Electron. In our search for a solution, we found the react-native-web library that was still in beta releases at that point but gave us the opportunity to start aggregating all the front-end code in a single codebase.

The experiment was successful. We were able to release the application on all major platforms and build the entire front-end code in about five months of work with a team of four engineers. From that experience came the desire to provide to the community an open source framework with a basic structure of the architecture with some adjustments and updated resources.

In Figure 3 we can see the design of the architecture, with each specific layer up through having the app running on multiple platforms. All the code is written in React Native syntax, in which the developers build UI components, screens, define the application context and data providers. All of this is compiled and distributed in multiple layers until they reach each targeted platform. For mobile, the React Native code is wrapped by Expo and gets ready to run in Android and iOS via the Expo Client app. For web and desktop, the React Native code is converted to DOM element via React Native Web, gets compiled and bundled by Webpack and is ready to run on the browser. However, for desktop Electron is needed in order to generate Windows, MacOS and Linux applications. Each one of these technologies are explained in more details later in this paper.

Figure 3 - Architecture Design



Source: The Author

2.2 Source Control

In this project we have used Git as the source control system, and GitHub as the application to maintain and provide access to the code. The code for the architecture can be found in this URL: https://github.com/diogoperillo/react-native-x-platform

We have also written a proof of concept application, explained in more detail later in this paper, that is located in the same repository, but in a separate branch named "xp-messaging" and can be found in this URL: https://github.com/diogoperillo/react-native-x-platform/tree/xp-messaging

2.3 Development Environment

Writing code in React Native and JavaScript can be done in many code editors. For this project it was decided to use the open source application Visual Studio Code, that can also be installed on all major operating systems and provides a vast library of plugins and resources for multiple programming languages.

For this react native code, the following plugins were installed for a better coding experience:

- ES7 React/Redux/GraphQL/React-Native snippets (dsznajder.es7-react-js-snippets)
- ESLint (dbaeumer.vscode-eslint)
- Babel JavaScript (mgmcdermott.vscode-language-babel)
- Path Autocomplete (ionutvmi.path-autocomplete)
- Prettier - Code formatter (esbenp.prettier-vscode)
- React Native Snippet (jundat95.react-native-snippet)
- vscode-flow-ide (gcazaciuc.vscode-flow-ide)

The following built in plugin was disabled because it creates conflicts with flow IDE:

- TypeScript and JavaScript Language Features (vscode.typescript-language-features)

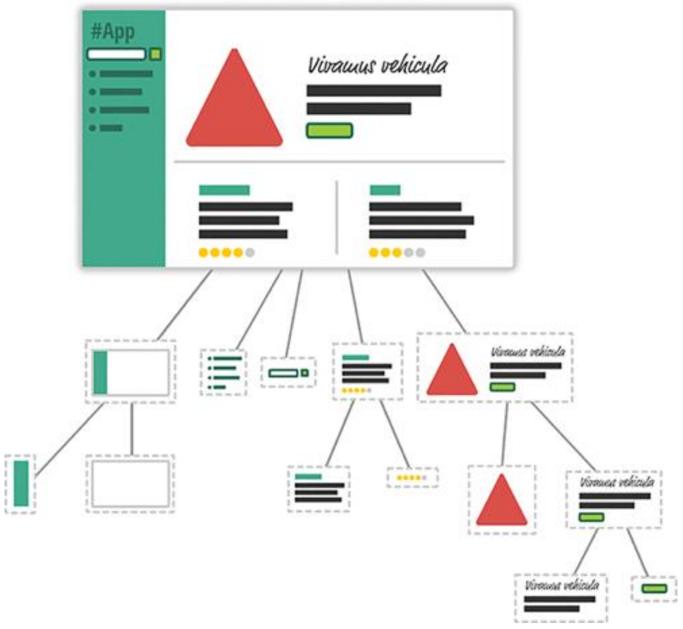
2.4 React

According to its website (FACEBOOK, 2019), React JS is an open source JavaScript library for building user interfaces. It was released by Facebook with the intent to solve problems they were having in their own applications and it's now maintained by the Facebook team and a large community of developers.

Among other challenges, building JavaScript applications was complicated when trying to unify logic and styles, synchronize data across different areas of the app and manipulate the UI. With React, these problems no longer exist (CHINNATHAMBI, 2018). You can split the code into smaller components that have their own logic and state, responding differently to user interactions and incoming properties. React components can also have their own specific visual styles and be responsible to update it when state or properties change.

We can compare a React application with the famous Russian matryoshka dolls, where a big project is composed of many smaller independent pieces (Figure 4).

Figure 4 – An example of how the visuals of your app can be broken into smaller pieces



Source: Learning React *(CHINNATHAMBI, 2018)*

2.4.1 JSX

React also introduces a new syntax for writing elements called JSX that embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called "components" that contain both. (FACEBOOK, 2019)

2.4.2 Handling Global Data

In React applications, data is passed top-down through properties from parent components to their children, but very often, it is convenient to handle data in a single source and update the necessary components when this data changes. There are some libraries made for this purpose such as Flux and Redux. In this aspect, after version 16.x of React the Context API became very useful as a built-in resource.

Context provides a way to pass data through the component tree without having to pass properties down manually at every level (FACEBOOK, 2019). It's designed to store global data and provide methods to update that data replicating the changes to every component listening to it.

2.4.3 Class Components, Stateless Components and Hooks

React components used to be classified into two categories: Class Component and Stateless Component. The former one is written as a class that extends React.Component and contains lifecycle methods, a state object and a render method; while a Stateless Component is written as a simple function that receives properties as parameters and returns the elements based on those properties. Function components couldn't have a state or hold their own data until the release of React 16.8, when introduced the concept of Hooks, a collection of APIs that lets you use state and other React features without writing classes (FACEBOOK, 2019).

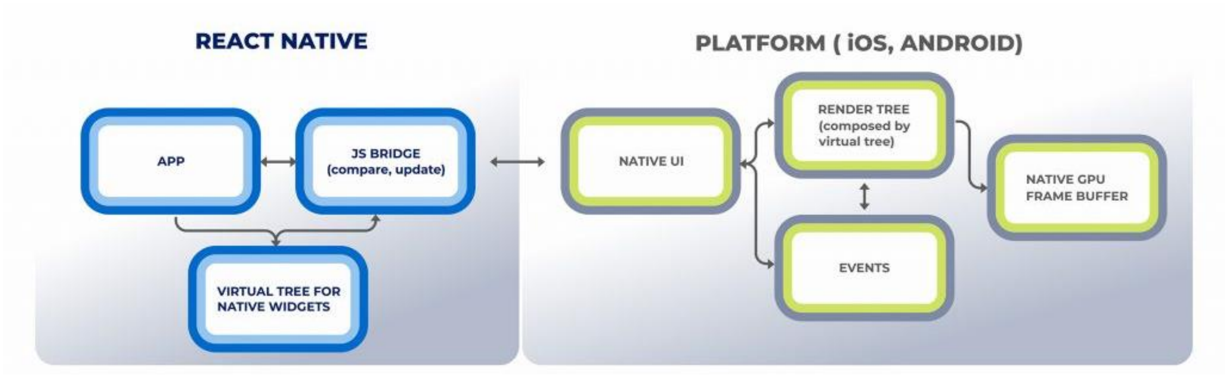
In our platform and proof of concept application we were able to use all these modern React features such as Context and Hooks.

2.5 React Native

React Native was released by Facebook in 2015, allowing developers to build applications for iOS and Android using a single codebase and programming language. React Native apps are also built with React and JSX. However, instead of web elements, it uses native components, so the code written in JavaScript is rendered in each platform specific native element.

React Native also provides a collection of APIs to determine platform specific code and create stylesheets in order to apply visual styles to components. As seen in a Railsware blog post (HEBDA, 2018), the framework uses platform-specific UI controls for native rendering. However, it’s orchestrated by a single-threaded JS code. Also, JS bridge executes the JS runtime and connects the app with its native parts. No hybrid’s WebView implementation makes the app’s functionality and view, resulting in a native product. (Figure 5)

Figure 5 – React Native architecture diagram


REACT NATIVE
PLATFORM (iOS, ANDROID)
APP
JS BRIDGE (compare, update)
VIRTUAL TREE FOR NATIVE WIDGETS
NATIVE UI
RENDER TREE (composed by virtual tree)
EVENTS
NATIVE GPU FRAME BUFFER

Source: Railsware *(HEBDA, 2018)*

2.6 Expo

If we are building a mobile application in React Native, it is important to understand that there are two layers in our app architecture: the JavaScript layer and the native layer that is implemented to convert the JS components into native elements. It is the native layer that will provide to the JavaScript one the ability to use the device features.

In order to help with the native layer, we are using Expo, a set of tools and services for building, deploying and using the native features of Android and iOS devices such as camera, local authentication and other resources. Expo also provides UI components to handle some use-cases not covered by React Native core, e.g. icons, blur views, and more (Expo, 2019).

With Expo it is possible to install and utilize the majority of JS packages. However, if a native resource not provided by Expo is needed, it is possible to eject the application, or in other words, unwrap it from the Expo shell and deal with the native dependencies by ourselves. Since this architecture is designed to provide the basic setup for an application, we are using the wrapped version of Expo.

2.7 React Native for Web

"React Native for Web" is an implementation of React Native components and APIs using DOM elements, making it possible to run React Native code on the web. With some Webpack configuration it is possible to alias all react-native imports and point it to 'react-native-web' when compiling the JavaScript code.

2.8 Babel, Webpack and Create React App (CRA)

Every software project tends to grow and needs to be well maintained, split in multiple files and well organized. The JavaScript language is being upgraded along the years to add new features and follow the best standards of coding, but not all browsers have support to these new syntaxes and resources. With the intention to bring a solution to this problem, a group of volunteer developers brought to the community the Babel library, a toolchain used to convert ECMAScript 2015 ahead to older versions of JavaScript that will be supported by more browsers and JavaScript interpreters.

Although it's not impossible to write React code in earlier versions of JavaScript, it is much better when using the new ECMAScript features such as classes, type annotations, imports, exports and JSX. It is also necessary to have a way to put together all the different JS, CSS, HTML and other types of files into an optimized and minified bundle, so we can have a lightweight application, especially for the web. For this purpose, another tool called Webpack is used. At its core, Webpack is a static module bundler for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph which maps every module your project needs and generates one or more bundles (WEBPACK, 2019).

When React was released, setting up a new application codebase was complicated and a lot of configuration for Babel and Webpack was needed. Ultimately the React team released a tooling library called Create React App (CRA), that works on Mac, Windows or Linux in order to encapsulate all the build configuration and make it easy to start developing, running and generating production bundles. It also supports by default the aliasing of react native web when a user needs to import and use React Native components on the web.

In our architecture we chose to use CRA for our web app and also for the desktop application as described later in this paper.

2.9 ESLint and Prettier

A productive development environment should help the developer write good quality code by offering snippets, highlighting and flagging possible errors and formatting the code automatically. With the intention to have a pattern, help code reviews and readability, it is important to define the rules and formats for our JavaScript codebase.

In this project, we are using two libraries that when combined, give us a simple way to keep the code in good quality. Since JavaScript is a loosely typed language, it can easily lead to errors that won't be found until the code is executed. ESLint is the first library that helps us solve these problems, it is an opensource tool that helps the JS code be analyzed without executing it, so developers can discover possible problems with their code during development. ESLint is also pluggable and easily configurable, to allow each project or team to set their own combinations of rules and patterns. The other library used was Prettier, an opinionated code formatter that loads ESLint rules and configurations and applies to the code after you save a file or type new lines (ESLINT, 2019).

2.10 JS Strong Typing with Flow

Another important consideration for a rich JavaScript codebase is to deal with type checking. As we discussed above, JavaScript itself is dynamically typed, so we need ways to check for errors that may occur when passing and expecting different types of variables.

There are some tools or languages that can be converted to JavaScript such as TypeScript, but in our project, we decided to use Flow. Instead of a new language, Flow is a static type checker that verifies the code for errors by using static type annotations. With Flow it's possible to define what type of variables each parameter of a method expects to receive. It is also possible to create custom types of objects specifying each property and method (FACEBOOK, 2019).

2.11 Electron JS

Electron JS is another open source tool released by GitHub that allows you to build cross-platform desktop applications using HTML, CSS, and JavaScript. It was originally released in July 2013 by Cheng Zhao, an engineer at Github, as part of their effort to produce a new code editor, Atom. Initially, the project was known as the Atom Shell but was soon rebranded simply as Electron.

Electron uses Chromium, the open source version of Google's Chrome web browser. What is included with Electron is technically the Chromium Content Module (CCM), the core code that makes a web browser a web browser. It includes the Blink rendering engine and its own V8 JavaScript engine. The CCM will handle retrieving and rendering HTML, loading and parsing CSS, and executing JavaScript as well (GRIFFITH e WELLS, 2017). Electron also provides other tools to auto update applications and generate binaries for Windows, Mac and Linux from any of these operating systems. We are using Electron to wrap the React web application into a desktop app.

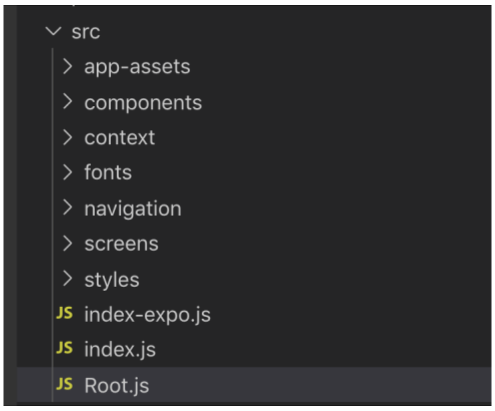
2.12 Navigation

One of the most important pieces of any application is how to connect and navigate between different screens, passing data and parameters. This was one of the challenges in our architecture. Since we couldn't find a library that would work well on web and mobile, we've decided to implement an abstraction using two popular modules for each platform. For web and desktop, we've used react-router and for Android and iOS we've chosen react-navigation, using react-navigation API as the model. The implementation can be found here: https://github.com/diogoperillo/react-native-x-platform/tree/master/src/navigation

2.13 Folder Structure

Figure 6 shows the architecture folder structure. The navigation folder contains the abstraction mentioned before, the screens folder should contain each one of the React components that will represent a route or screen of the application. Styles contains global variables and stylesheets, while components folder holds each reusable component, fonts and assets contains images and font files that should be used in the app. The Root.js file is the entry point where the context is applied, and the Router is rendered.

Figure 6 – Architecture folder structure



```
src
  app-assets
  components
  context
  fonts
  navigation
  screens
  styles
  index-expo.js
  index.js
  Root.js
```

Source: The Author

3 PROOF OF CONCEPT APPLICATION

In order to prove the efficiency of our architecture, this article also includes a simple application built with this codebase that will be running on all targeted platforms. This app was developed by one person in about thirty working hours, proving, even for a proof of concept, that it is a very interesting exercise to deploy for Windows, Mac, Linux, Android, iOS and Web.

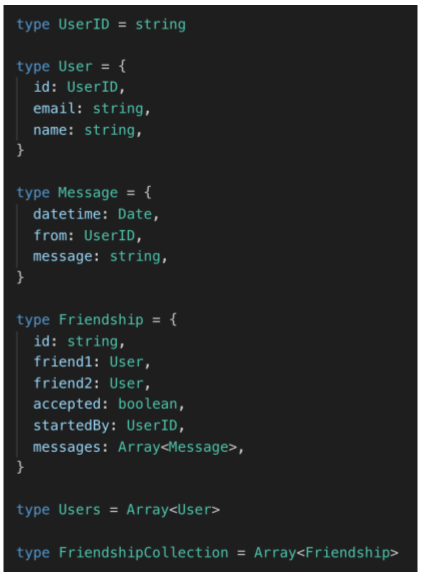
The application is called XP Messaging (Cross-Platform Messaging), a direct message app that allows users to send and receive text messages. XP Messaging enables approved contacts to access their received messages with an account on multiple devices, with live updates. Since the focus of this prototype is the front-end part of an application, we decided to use Google Firebase as the data provider.

Firebase is a suite of functionalities including web hosting, real time noSQL database, file storage, authentication provider and other features such as analytics, crash reporting and more. Firebase uses google cloud infrastructure and it is free for small and under development projects. It has a friendly JavaScript SDK that works very well integrated with React Native.

In this application we used the authentication provider for login and account creation, real time Firestore database to store the messages and keep the UI in sync and file storage to store the profile pictures.

Figure 7 illustrates the database structure described as flow types.

Figure 7 – Database Structure



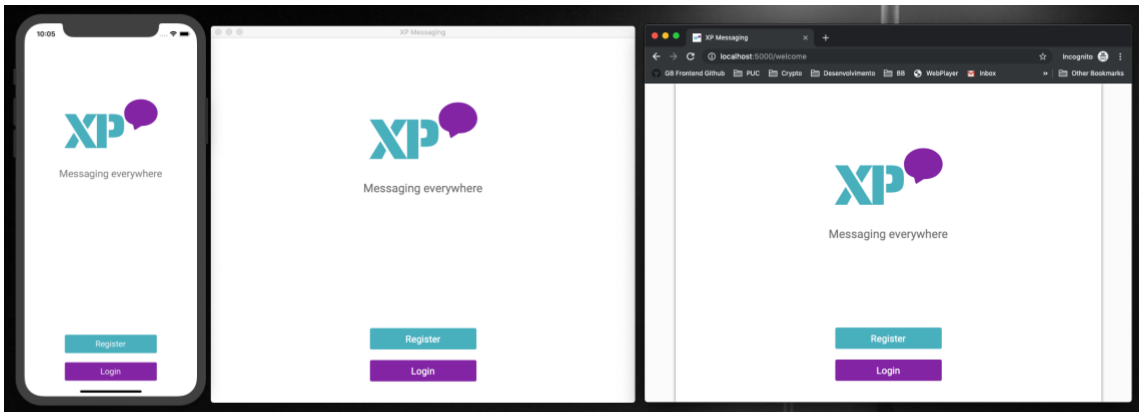
```
type UserID = string

type User = {
  id: UserID,
  email: string,
  name: string,
}

type Message = {
  datetime: Date,
  from: UserID,
  message: string,
}

type Friendship = {
  id: string,
  friend1: User,
  friend2: User,
  accepted: boolean,
  startedBy: UserID,
  messages: Array<Message>,
}

type Users = Array<User>

type FriendshipCollection = Array<Friendship>
```

Source: The Author

3.1 Application Screens

Another tool used in this project was Figma, a tool to design screens and wireframes used by designers and front-end engineers. In Figure 8 we can see some screens demonstrating the result of our experiment.

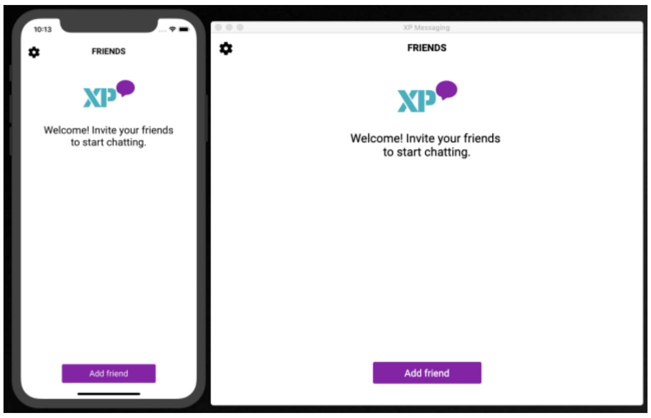
Figure 8 – Welcome screen in multiple platforms



Source: The Author

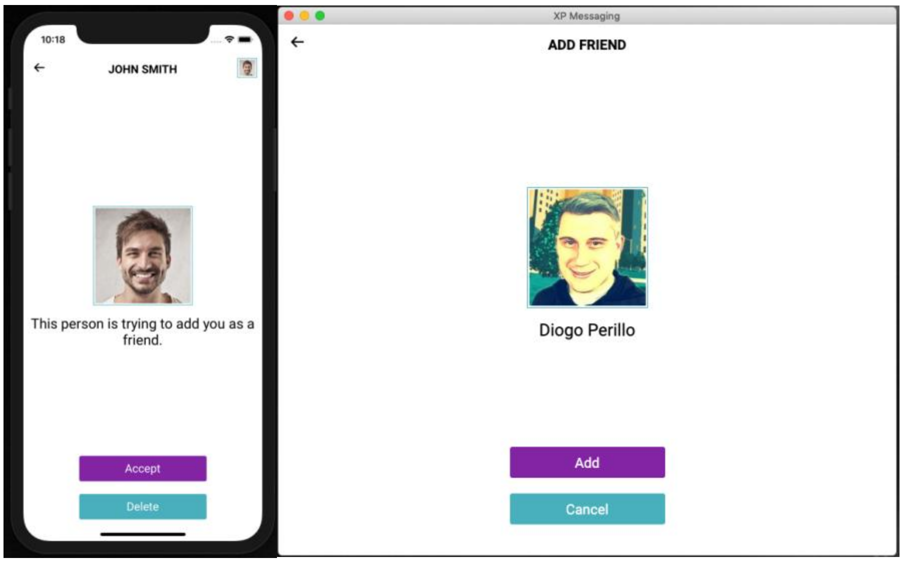
In Figure 8 it is possible to see the application running on three platforms. The left one is an iPhone simulator running the native version, the middle window is a desktop application in macOS and the right one runs in a Chrome web browser. In Figure 9 we can see two different users logged into the app, one on the left in a mobile device and the right one in the desktop version. Figure 10 shows the contact request made to each other.

Figure 9 – Home screen on multiple platforms



Source: The Author

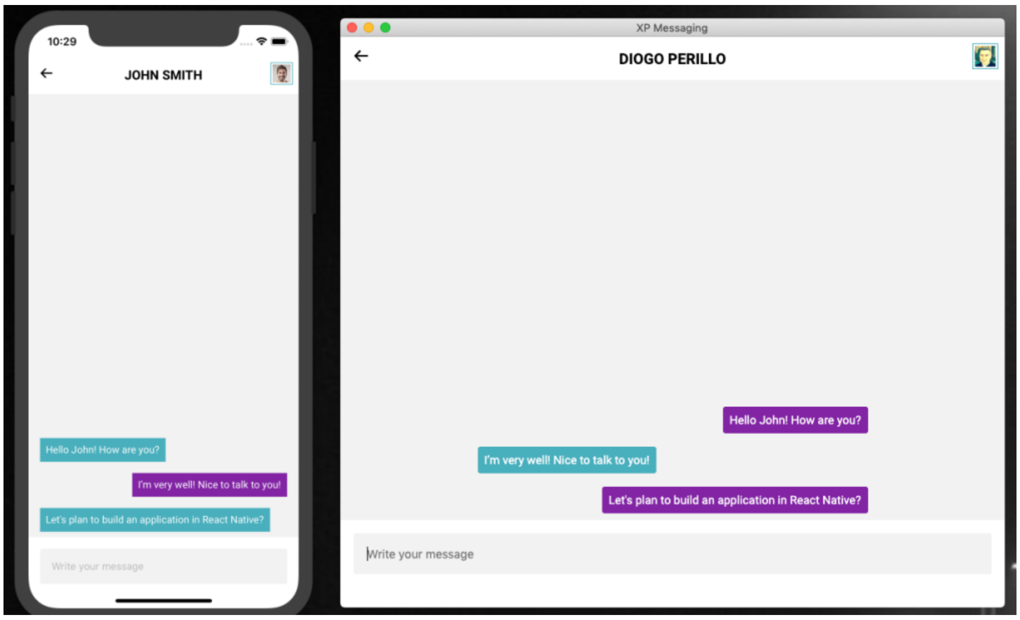
Figure 10 – Friendship request



Source: The Author

After the request made and accepted by the other side, the friendship is started, and they can finally send messages to each other in real time as seen in Figure 11.

Figure 11 – Friendship request



Hello John! How are you?

I'm very well! Nice to talk to you!

Let's plan to build an application in React Native?

Source: The Author

4 CONCLUSION

In this project, we had the experience of designing and implementing the architecture of a framework to build cross-platform applications. The process of development started with researches around possible technologies and contemplated the importance of delivering apps to as many operating systems as possible in order to support a higher number of users. After careful consideration and comparison with other solutions, it was decided to use React Native combined with a collection of JavaScript libraries. In order to prove not only the efficiency of the project but also significant aspects such as development speed and code quality, a proof of concept application was built. With one single developer, it was possible to generate a version of the app for desktop (Windows, macOS, and Linux), mobile (iOS and Android) and web (https://xp-messaging.web.app). The number of hours used to build and have the app ready to be distributed for all mentioned targets was very close, if not the same, as it would be to accomplish it for only one platform (mobile, desktop, or web).

The framework is still open for future improvements, such as adding automated tests and using the project in a production environment: a real-life application with a significative number of users. It is also planned to develop a command-line interface (CLI) to generate new app codebases by downloading the architecture source and saving to a local folder named by the user, as well as providing shortcut commands to create production binaries and bundles for each platform. Besides that, it is desired to build a collection of essential UI components such as text fields, buttons, and other common elements to accelerate the process of developing an app.

REFERENCES

APACHE. Apache NetBeans, 2019. Disponivel em: <https://netbeans.org/features/java/index.html>. Acesso em: October 2019.

BABEL. What is Babel? **Babel**. Disponivel em: <https://babeljs.io/docs/en/index.html>. Acesso em: September 2019.

BODUCH, A. **React and React Native**. [S.l.]: Packt, 2017.

BODUCH, A. **React 16 Tooling**. [S.l.]: Packt, 2018.

CAIRNS, B. **Flutter - A Beginners Course**. [S.l.]: Packt , 2018.

CHINNATHAMBI, K. **Learning React:** A Hands-On Guide to Building Web Applications Using React and Redux. Second Edition. ed. [S.l.]: Addison-Wesley Professional, 2018.

DOBBS, S. How are mobile devices changing the Web? **Micro Mart**, 30 April 2015. p. 58.

DOSSEY, A. A Guide to Mobile App Development: Web vs. Native vs. Hybrid. **clearbridgemobile.com**, 2019. Disponivel em: <https://clearbridgemobile.com/mobile-app-development-native-vs-web-vs-hybrid/>. Acesso em: September 2019.

ECLIPSE FOUNDATION. Eclipse, 2019. Disponivel em: <http://www.eclipse.org/downloads/packages/release/kepler/sr1/eclipse-ide-java-developers>. Acesso em: October 2019.

ESLINT. About ESLint, 2019. Disponivel em: <https://eslint.org/docs/about/>. Acesso em: 15 September 2019.

EXPO , 2019. Disponivel em: <https://docs.expo.io/versions/v33.0.0/>. Acesso em: September 2019.

FACEBOOK. Getting Started. **Flow**, 2019. Disponivel em: <https://flow.org/en/docs/getting-started/>. Acesso em: 15 September 2019.

FACEBOOK. React JS. **React JS**, 10 September 2019. Disponivel em: <https://reactjs.org>. Acesso em: September 2019.

FACEBOOK. React Native Showcase, 2019. Disponivel em: <https://facebook.github.io/react-native/showcase>. Acesso em: October 2019.

GITHUB. Front-end JavaScript frameworks. **Github Collections**, 2019. Disponivel em: <https://github.com/collections/front-end-javascript-frameworks>. Acesso em: 20 October 2019.

GLOBALSTATS. Desktop vs Mobile Market Share Worldwide. **StatCounter**, 2019. Disponivel em: <https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#monthly-200901-201910>. Acesso em: 15 October 2019.

GLOBALSTATS. Operating System Market Share Worldwide. **StatCounter**, 2019. Disponivel em: <https://gs.statcounter.com/os-market-share#yearly-2009-2019>. Acesso em: 15 October 2019.

GOOGLE. Android Studio, 2019. Disponivel em: <https://developer.android.com/studio>. Acesso em: October 2019.

GOOGLE. Flutter, 2019. Disponivel em: <https://flutter.dev/>. Acesso em: September 2019.

GREIF, S.; BENITTE, R.; RAMBEAU, M. State of JS 2018, 2019. Disponivel em: <https://2018.stateofjs.com/front-end-frameworks/react>. Acesso em: September 2019.

GRIFFITH, C.; WELLS, L. **Electron:** From Beginner to Pro: Learn to Build Cross Platform Desktop Applications using Github's Electron. [S.l.]: Apress, 2017.

HEBDA, A., 7 August 2018. Disponivel em: <https://railsware.com/blog/react-native-vs-native-app-development-ios-and-android-in-one-go>. Acesso em: September 2019.

LI, D. **Building Enterprise JavaScript Applications**. [S.l.]: Packt, 2018.

MICROSOFT. Xamarin. **Xamarin**, 2019. Disponivel em: <https://dotnet.microsoft.com/apps/xamarin>. Acesso em: 10 October 2019.

SKUZA, B.; MROCZKOWSKA, A.; WŁODARCZYK, D. Flutter vs React Native – what to choose in 2019? , 2019. Disponivel em: <https://www.thedroidsonroids.com/blog/flutter-vs-react-native-what-to-choose-in-2019>. Acesso em: October 2019.

STACKOVERFLOW. Programming, Scripting, and Markup Languages. **Developer Survey Results**, 2018. Disponivel em: <https://insights.stackoverflow.com/survey/2018#technology-_-programming-scripting-and-markup-languages>. Acesso em: October 2019.

WEBPACK. Webpack, 2019. Disponivel em: <https://webpack.js.org/>. Acesso em: September 2019.