Comprehensive Analysis of Innovative Cross-Platform App Development Frameworks

Tim A. Majchrzak
University of Agder, Kristiansand, Norway
Email: timam@uia.no

Andreas Biørn-Hansen
Westerdals, Oslo, Norway
Email: andreasb.nor@gmail.com

Tor-Morten Grønli
Westerdals, Oslo, Norway
Email: tmg@westerdals.no

Abstract

Mobile apps are increasingly realized by using a cross-platform development framework. Using such frameworks, code is written once but the app can be deployed to multiple platforms. Despite progress in research on cross-platform techniques, results (i.e. apps) are not always satisfactory. They are subject to tedious tailoring and the development effort tends to be notable. In these cases, either pure web apps (realized through web browsers) or native apps (realized for each platform separately) are chosen. Recent activities have led to new approaches. In this paper, we have a closer look at three of these, namely React Native, the Ionic Framework, and Fuse. We present a comprehensive analysis of the three approaches. Our work is based on a real-world use case, which allows us to provide generalizable advice. Our findings suggest that there is no clear winner; the frameworks incorporate notable ideas and general progress in the field can be asserted.

Index Terms

Mobile, App, Mobile App, Cross-Platform, Mobile Frameworks, React Native, Ionic, Fuse, Evaluation

1. Introduction

The market for mobile platforms is mainly divided between Google's Android and Apple's iOS [1]. These two remain incompatible (cf. e.g. [2]), both in terms of user mobility between device vendors and of apps. When developing apps that should run on Android and iOS – and possible on further platforms such as Windows Phone – there are three choices [3]: First, a pure web app based on HTML5, CSS and JavaScript can be used. Second, native apps can be developed, commonly multiplying the effort by the number of targeted platforms. Third, cross-platform development frameworks can be employed: An app is developed *once* but deployed to multiple platforms [4].

There is a variety of options for cross-platform development. Frameworks follow different paradigms [4][3] and there are many to choose from (see Section 2.1). However, only few approaches have found widespread adoption. Most notably, the Web-technology based PhoneGap is used. While results are appealing in general [4][5], approaches such as PhoneGap arguably are not the *cure* to cross-platform problems. Approaches that originate in the scientific community and are *theoretically sound* have provided manifold insights (cf. e.g. [6]). They typically are not easily adopted by industry, though (cf. [7][8]). Without question, more research is needed [7] and the field will benefit from work on further approaches.

To deepen the understanding of state-of-the-art cross-platform development and to contribute to the knowledge on existing approaches, we have assessed three frameworks. React Native [9], Ionic Framework [10], and Fuse [11] are rather new and have not undergone extensive study. While practitioners and hobbyists discuss these frameworks actively, virtually no scientific papers on them or their underlying ideas exist to the best of our knowledge. With this paper, we set out to close this gap. In particular, we strive to give business-oriented advice by assessing the frameworks experimentally with a real-world scenario focusing on User Experience (UX) [12]. While analyses of other cross-platform frameworks are not scarce, few papers follow a practice-oriented assessment and even fewer analyze next generation frameworks. These three frameworks are particularly interesting for study since they mark a new step of approaches that also introduces paradigmatic shifts. While our approach is design-oriented, we have combined it with an informal survey to enrich our findings.

This paper makes several contributions. First, it provides an assessment of three innovative cross-platform frameworks that have not yet been extensively studied. Second, it introduces work on a prototype app, which should prove useful for evaluation beyond our work. Third, it generalizes findings and summarizes advice. The remainder is structured as follows. We introduce the background of our work in Section 2. In Section 3 we describe the prototype that is used for the evaluation of frameworks in Section 4 along with implementation details. We discuss our findings in Section 5 before drawing a conclusion in Section 6.

URI: http://hdl.handle.net/10125/41909
ISBN: 978-0-9981331-0-2

2. Background

In the following, we describe the background of this paper. Starting with related work, we move on to our research method and the choice of frameworks.

2.1. Related Work

A myriad of papers and also textbooks cover topics of app development. Cross-platform development is addressed by at least some of these, although not necessarily as the main topic. To shed light on related literature, we take a look on such work in the following that compares several cross-platform app development frameworks. We do not include papers here that discuss techniques and technology for building web apps.

Web technology provides one means to develop apps that span platforms (cf. [13][14]); however, developing web apps can be seen as an alternative to using a cross-platform framework [15] and web apps can be used as a benchmark for assessment [4]. Likewise, we do not include papers here that tackle native app development for more than one platform. Native apps are also useful for benchmarking cross-platform approaches, particularly concerning their look & feel and their performance.

An overview of papers that have compared cross-platform development framework is given in Table 1. Quite notable is the first peak in 2012 and 2013. After proposals to understand the proliferation of cross-platform development frameworks had been made, several teams of authors set out to describe the field as a whole. That efforts were not sustained in 2014 is unfortunate, even though publications from 2015 show that the topic did not lose relevance.

Besides comparison papers, several others works can be seen as preconditions to our evaluation, including such on the challenges of app development in general (e.g. [16]). Where applicable, works directly related to aspects of our assessment are cited in the remainder of this paper.

It is also notable that specialized papers have emerged. An example is a particular focus on the energy consumption of apps created with cross-platform development frameworks [17]. Moreover, general assessments of the various platforms [18] need to be taken into account. In addition, Huy and van Thanh [19] have proposed criteria for evaluation of apps. They do not provide an actual assessment but rather propose "how to do it". Their idea is to take different viewpoints, namely that of developers, users, and service respectively content providers.

Paper	Year	Evaluated Frameworks	Particularities
[20]	2015	WebWorks, PhoneGap, Titanium; native Android and iOS for comparison	Extensive work including a performance evaluation and GUI considerations
[21]	2015	AngularJS, jQuery Mobile, HTML5/JS, RhoMobile, PhoneGap, Sencha Touch	Criteria definition and qualitative comparison
[22]	2015	PhoneGap, Smartface App Studio Titanium, Xamarin	Rather short paper; does not cite prior work on the topic
[17]	2015	PhoneGap, Titanium	Special focus on energy consumption
[23]	2014	Intel XDK, PhoneGap, Titanium	Evaluation by two independent teams; focus on User Experience
[12]	2014	jQuery Mobile, MoSync, PhoneGap, Titanium	Focus on animations
[24]	2013	(PhoneGap, Titanium)	The paper has a more general focus; proposing much fundamental work
[25]	2013	PhoneGap, Sencha Touch, Titanium	Assessment criteria; apps developed for performance evaluation
[26]	2013	PhoneGap (and Sencha Touch), Rhodes, Titanium	Comparison based on a sample application
[27], [4]	2012	PhoneGap, Titanium; Webapps and native apps for comparison	Proposing criteria for evaluation; widely cited paper
[28]	2012	DragonRad, MoSync, PhoneGap, Rhodes	One of the first comprehensive studies
[29]	2012	*(none)*	Theoretical assessment of cross-platform possibilities in general

Table 1. Cross-Platform App Framework Comparisons

2.2. Research Method

The choice of method is not in the focus of this article due to its technical nature. We briefly highlight our approach nonetheless to provide a better understanding and underline the rigor of our research.

Design-Science Research (DSR) [30], a methodology aided by artefactual design and development, was adopted. It helps to combine the information systems research focus on the topic with an experimental approach typical for computer science. We deem this a reasonable compromise between entirely technical work and a purely empirical assessment.

We conducted a short survey to enhance the backbone and decision basis of our experimental work. This online survey, informed by [31], was used to gather data on framework popularity, issues related to cross-platform development, and to identify key decision points from practitioners and the industry. A total of ten questions were asked and they were all scored using a five-point Likert scale. After first running a pilot test on the questionnaire, the finished version was distributed and gained 101 responses. These responses helped form the initial stage of the requirements analysis and the first round of evaluation.

Three technical instantiations were developed iteratively and incrementally. For each iteration the evaluation results where fed back into a new planning, development and evaluation loop. The initial set of requirements were informed by both industry needs and through the literature review, thereby considering both academic and industry challenges. In particularly, we have been working with an industry partner who provided us with a real-world case.

The results gained from evaluation were fed back to artefact development and given to the stakeholder groups, closing the loop of information to industry and academia.

2.3. Choice of Frameworks

To ensure that the comparison and evaluation of the frameworks were as objective as possible, only frameworks leveraging JavaScript as their main programming language were chosen. Our study thereby is deliberately different to some other works that explicitly compared such approaches that were paradigmatically different. Arguably, this has been done quite extensively (see Section 2.1) and provided rather general insights. For industrial application, typically some if not all but one paradigm is ruled out due to a host of preconditions. With the emergence of new frameworks, technology choice is not easy. Due to the possibility of underlying differences in frameworks using other languages, those using anything but JavaScript were excluded.

Three frameworks for cross-platform development were chosen due to their novelty, recentness, and (perceived) developer interest. Identified academic research on the topic mainly consisted of prototype development using frameworks such as jQuery Mobile, Appcelerator Titanium and PhoneGap. In an effort to push for the inclusion of new technologies, the choice of frameworks was partially based on

1) the literature search and review process,
2) a non-scientific – industry-view driven – analysis of the market through following developer hype and community contributions, and
3) partially on a survey conducted targeting mainly mobile developers (as explained in Section 2.2).

The initial analysis was conducted by following the JavaScript and cross-platform mobile developer communities. This, together with the literature review, was the foundation for the questionnaire survey conducted to gain additional and up-to-date insights from active mobile developers. The survey questions were formed based on existing literature and identified patterns in the practitioner communities.

In conclusion, React Native 0.22 [9], the Ionic Framework 2.0.0-beta.3 [10], and Fuse 0.11.1 [11] were chosen for assessment.

3. Prototype Design and Implementation

In the following, the implementation of an evaluation prototype is described. The prototype instantiates an industry case for an app. For ease of reading, when naming the frameworks, we use **bold** font in this and in the following section; this is not repeated if it is named again within the same paragraph unless weighted against another

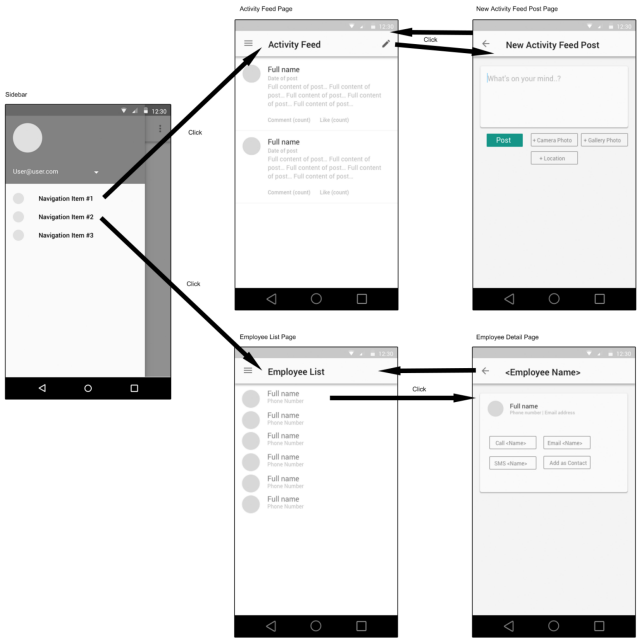

Figure 1. Proposed Design and Flow in Prototypes

framework there. `Typewriter` denotes libraries, functions, and technical assets.

3.1. Design and Feature Requirements

The case application provided a basis upon which we designed a proposal for the prototype. The proposal contained some of the core aspects of the case app, such as user navigation patterns and flow. This was in turn used as the foundation throughout the implementation of features and requirements. Figure 1 depicts the design and application flow proposal. We give code examples were appropriate.

A use-case diagram was developed, including a set of core features from the case application (Figure 2). The use case, being informed by literature review, industry survey and real world use case from the case company highlights the three important areas that features were related to: UI element navigation, multimedia (Gallery and Camera), CRUD manipulation (Contact list and phone book). These elements will highlight ease or difficulty of use/access across frameworks and there by test elements crucial in any business application.

The desired outcome of the feature diagram was to have requirements for evaluation of the comprehensiveness and state of the chosen technical frameworks.

3.2. Implementation

The use-case diagram in figure 2 was the foundation for the technical implementations. Using each of the three

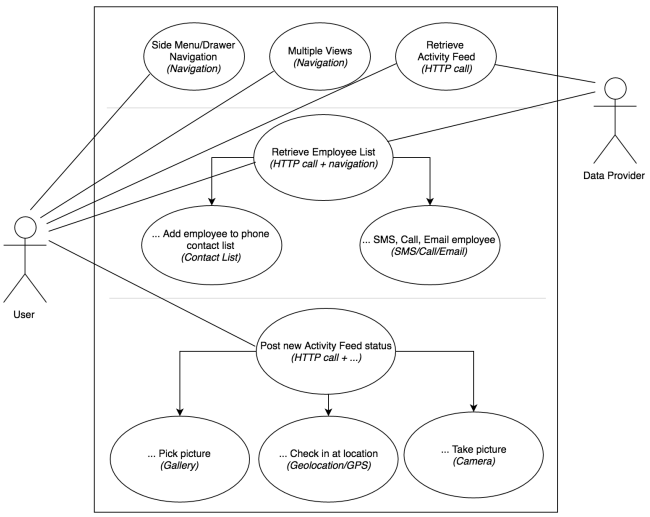

Figure 2. Functional Requirements

aforementioned frameworks (Section 2.3), three prototypes were (separately) implemented for evaluation purposes. Being based on a real-world case application, the features were of a complexity to achieve relevance and applicability. Implementation details are given in the following.

3.2.1. Side Menu and Navigation. The side menu and navigation from the case application were closely coupled, while still being separate technical entities. The side menu was the main navigation pattern, accessible both through a *swipe-to-right* gesture and a *hamburger* icon button.

In **React Native**, navigation was solved using a third-party library called `react-native-router-flux` [32]. The library allowed for easy implementation of navigation, defining all routes (pages and views) inside a router, and then programmatically calling the router with a defined route to conduct a navigation. The library exposed a simple API for integration of side menu together with `react-native-drawer` [33], another third-party community library.

In **Ionic Framework**, the Ionic 2 beta followed a page stack-based approach to navigation, much like an actual native app ([34]). Views would be pushed and popped instead of basing navigation on URLs, regardless of Ionic's WebView and Hybrid approach to cross-platform. The root JavaScript file in the Ionic project declared the launch view of the app by defining a component presenting the view. Ionic had standardized built-in support for side menu integration. An array held the different views the users could navigate to from the side menu. The array was traversed in a corresponding view file, generating a list of `<button />` components navigating to the different views.

In **Fuse** due to lack of proper navigation examples and documentation of the feature, the implementation may not be representative of the actual navigation feature. Fuse exposed three different navigation types, `LinearNavigation`, `DirectNavigation` and `HierarchicalNavigation` [34]. Upon implementation, both LinearNavigation and HierarchicalNavigation had unexpected behavior, where at one point the navigation depended on the order of `<Page />` components in the code base. This resulted in behaviour where the *Go Back* button would navigate to another Page instead of popping the navigation stack and go to the previous page. The final implementation used DirectNavigation which replaced the navigation stack at each navigation, causing the *Go Back* functionality to require a custom implementation. Additionally, no transition animation would run while navigating. Fuse had a component declared `<EdgeNavigator ux:Name="EdgeNavigator">`, which wrapped around the entire root main view, making the side menu available from anywhere in the app. Using the `ux:Name="EdgeNavigator"` identifier of the navigator, components could be assigned to the left of the view as expected. The regular swipe-to-reveal gesture pattern did not work as expected while implementing the menu, so a standard "hamburger" button in the navigation bar was added to replace the gesture.

3.2.2. HTTP Requests and Network Communication. Fetching remote data from RESTful API endpoints was at the core of the case application. The endpoint used to fetch data in the prototypes was a publicly exposed mocking API with the sole purpose of aiding with prototyping. Regardless of the structure of the data exposed, the method for fetching data would be the same within the respective frameworks.

In both **React Native** and **Fuse**, the standard implementation of data fetching over `HTTP` was based on Chrome's `fetch` method. A simple request using `fetch` may have the following form:

```
fetch("http://demo0242870.mockable.io/ employees")
```

In **Ionic Framework**, an `HTTP` module was provided by the AngularJS framework, which Ionic built upon, to be used when fetching remote data. The `HTTP` module was imported and made available through Angular's *Dependency Injection* system:

```
this.http.get("http://demo0242870.mockable.io /employees")
```

3.2.3. Developing User Interfaces. As **React Native** embraced the *interpreted* paradigm, all user interfaces generated and displayed in the app were actual native UI components. There are no WebViews or HTML5 solutions. User interfaces in **React Native** were declarative and component-based by design, as was React.js by itself. The React JSX syntax for view components was written inside of

JavaScript classes, uniting view and functionality into one file. Styling components was done in a CSS-like fashion using JavaScript objects, with web-like Flexbox support available. The React Native framework exposed different UI components which were retrievable by import from the `react-native` node.js module.

User interfaces in **Ionic Framework** were developed in HTML and CSS with JavaScript for logic using Angular.js. Logic and the view was separated into `.js` and `.html` files, and linked together by a `templateUrl` property in the JavaScript file. The different components provided by Ionic for developing interfaces were in reality Angular.js' directives, enabling non-standard HTML tags to be rendered as native-looking interface components.

As **Fuse** embraces the *interpreted* paradigm, all user interfaces generated and displayed in the app were actual native UI components. There are no WebViews or HTML5 solutions, similar to **React Native**. User interfaces in **Fuse** were powered by an X(A)ML-like syntax named *UX Markup* [34], a proprietary markup language developed by **Fuse**. When running on device, the markup was compiled to native code and then native components were rendered on the screen. JavaScript was executed in either the V8 or JavaScriptCore engines depending on platform.

3.2.4. Using the Geolocation. For **React Native**, the framework documentation explained [35] that the framework followed the same implementation specification for the Geolocation interface as a Webapp would use in the browser. Without having to import anything, the `navigator` interface was available from anywhere in the code base:

```
navigator.geolocation.getCurrentPosition( ↩
    options, callback);
```

In **Ionic Framework**, an importable package for accessing the Geolocation API was available through the Cordova [36] abstraction library `Ionic Native`. It exposed a simple API with a JavaScript promise to retrieve the current location coordinates of the device:

```
Geolocation.getCurrentPosition();
```

Geolocation in **Fuse** involved importing the `GeoLocation` module from the Fuse module library. When called, the module returned the device location coordinates through a JavaScript promise:

```
GeoLocation.getLocation();
```

3.2.5. Using the Camera. For **React Native**, a community-made package named `react-native-image-picker` [37] exposed a simple API abstraction to consume and work with both the camera and the device photo gallery:

```
ImagePickerManager.launchCamera(options, ↩
    callback/promise);
```

For **Ionic Framework**, an importable package for accessing the camera was available through the Cordova abstraction library `Ionic Native`. It exposed a simple API with a JavaScript promise to open a camera view and return the file data after taking the picture. The `Camera.getPicture()` method accepted an `options` object as an argument, where `options.mediaType` defined whether to set the camera to video or photo mode:

```
Camera.getPicture(options, callback/promise)
```

For **Fuse**, accessing the camera involved importing the `Camera` module from the Fuse module library. The method accepted an `options` argument. After successfully taking a photo, the module returned the file through a JavaScript promise:

```
Camera.takePicture(options, callback/promise)
```

3.2.6. Using the Image Gallery. For **React Native**, a community-made package named `react-native-image-picker` [37] exposed a simple API abstraction to consume and work with both the camera and the device photo gallery:

```
ImagePickerManager.launchImageLibrary(options ↩
    , callback/promise)
```

For **Ionic Framework**, an importable package for accessing the image gallery was available through the Cordova abstraction library `Ionic Native`. It exposed a simple API with a JavaScript promise to retrieve the URI of the selected image(s):

```
ImagePicker.getPictures(callback/promise)
```

For **Fuse**, at the time of prototype development, no component for accessing the image gallery was identified.

3.2.7. Using the Contact List. For **React Native**, a community-made package named `react-native-contacts` [38] had a partial cross-platform implementation for working with the device contact list. An `addContact()` method was available matching the requirement specification of the case app:

```
Contacts.addContact({dataStructure})
```

In **Ionic Framework**, due to problems with the versioning of the installed `Cordova Command Line Interface` tool in combination with the `Ionic-Native` library, the *contacts* feature was not fully implemented.

For **Fuse**, at the time of development, only a skeleton for accessing the contacts list was available through a community-made package named `fuse-contacts` available through GitHub [39]. No API for creating and adding new contacts was available, thus eliminating the reason to implement the package for the prototype according to the use case diagram in Figure 2.

3.2.8. Using Call, Email and SMS. For **React Native**, leveraging the community-made package `react-native-communications` [40], a set of

exposed methods made use of communication protocols accessible from the **React Native** app. An example is given in Listing 1.

Listing 1. Implementation of communication protocols in React Native

```
Communications.phonecall(phoneNumber, true)
Communications.text(phoneNumber)
Communications.email(email, ...null)
```

For **Ionic Framework**, implementation of phone call, email, and SMS depended on WebView URI schemes. The schemes were implemented into the WebView to communicate with the respective device communication APIs [41]. An example is given in Listing 2.

Listing 2. Implementation of communication protocols in Ionic

```
call() {
  window.location = `tel:${this.person. ↩
      phoneNumber}`;
}

mail() {
  window.location = `mailto:${this.person. ↩
      email}`;
}

sms() {
  window.location = `sms:${this.person. ↩
      phoneNumber}`;
}
```

For **Fuse**, at the time of development, the only API for communication was `phone.call(number)` [34]. Thus, it lacked methods for email and sending short messages.

3.2.9. Summary of Design and Implementation. Table 2 displays a high-level summary of the design and implementation. The different subsections are summarized into a simple table for easy comparison between frameworks. The table allows comparing the features of the frameworks.

	React Native	Ionic	Fuse
Paradigm	Interpreted	Hybrid	Interpreted
Framework Version	0.22	2.0.0-beta.3	0.11.1
JavaScript Version	ES2015	ES2015	ES5
View Engine	JSX	Angular.js	UX Markup
Camera Access	Yes	Yes	Yes
GPS Access	Yes	Yes	Yes
Image Gallery Access	Yes	Yes	No
Contacts Access	Yes	Yes (Untested)	No
Navigation Implementation	Intermediate	Simple	Complex
Sidebar/Drawer Impl.	Simple	Simple	Simple
Remote Data Fetching	Simple	Simple	Simple
Debugging	Simple	Simple	Simple
Framework Setup	Simple	Simple	Simple

Table 2. Summary of Information About the Frameworks

4. Evaluation

In the following section, we present the evaluation of the frameworks. This is done step-wise in subsections, reflecting on a variety of aspects of framework utilization.

4.1. Information on Framework & Developer

React Native is developed and maintained by Facebook, and was launched with iOS-only support in January 2015 [42]; Android support was released later that year [43]. React Native, leveraging the view-rendering library React.js, is built and used activly in production by Facebook, Facebook Ads Manager, Facebook Groups, and Instagram to name a few [44].

Ionic Framework is an extensive open-source hybrid app development suite maintained by Drifty Co. The app development framework is only one piece of their ecosystem; in addition to the framework, they also provide services such as prototyping tools, analytical tools, and a push notification service through their platform Ionic.io.

Fuse is developed by Fusetools, a company based in Norway and Palo Alto, USA. The framework belongs to the *interpreted* paradigm, meaning user interfaces are actual native components. Apps are written mainly in JavaScript and by using their XML-like markup language, *UX Markup* for layout, design and animations [34]. Fuse also proposes a new programming language called *Uno*, a C# dialect working as a compilable abstraction over native code [34]. In addition to Uno, Fuse includes a concept called *Foreign Code* where Java and Objective-C can be written directly into Uno code and executed on the targeted mobile platforms.

4.2. IDEs and Text Editors

While **React Native** has no official IDE or text editor, Facebook previously released Nuclide, an extension to the Atom editor.

Ionic Framework does not enforce any IDEs or text editors. No official recommendations of such could be identified, either.

Fuse has no official IDEs or text editors. However, it provides plug-ins and add-ons to existing text editors such as Sublime and Atom.

4.3. Developer Experience

For **React Native**, possessing prior knowledge of React.js was a clear productivity boost while developing in React Native. As most of the basic knowledge from React.js could be transferred to the React Native framework, any developer starting out making apps should possess some degree of knowledge of React by itself. An important feature was *Live Reload*, made possible by how JavaScript was interpreted in the JavaScriptCore (or V8 while debugging). When new code was produced and saved in the text editor, React Native would bundle the code and files together, and serve it to the device. The latter only had to re-interpret the new JavaScript and display the updated app. React Native made

the developer experience encouraging, limiting time spent by developers and enhancing the phase of rapid prototyping in the development. The prototype was built during two full work days.

As the **Ionic Framework** follows the WebView-based or *Hybrid* paradigm, the developer experience was similar to what one would expect in regular web development. The main differences in terms of developer experience compared to regular web development was the Ionic CLI, ready-made interface components, and dependency on Cordova for hardware and device API access. The ready-made interface components available in Ionic's component library made development of standard-looking user interfaces a matter of copy-paste of code. Their component library [45] provided Ionic-specific HTML code necessary to create a wide range of interfaces, from complex lists to segment controls, grids and modals. Most of the prototype developed in Ionic got its user interface from the library. Another time-saving feature in Ionic was the `--livereload` flag available in the CLI when running the project in development mode. Whenever `saving` the project in the text editor or IDE, the emulator or web browser window should refresh the app to reflect the latest changes. This feature remained unstable during implementation. Overall, the developer experience using Ionic was encouraging. The prototype was finished in approximately three 7-hour days of work. This included installing the Ionic CLI, getting familiar with the Angular 2 framework, and searching for components.

For **Fuse**, accommodating developers in creating native apps was advertised as a true goal. Increased developer satisfaction through real-time device and emulator preview of apps, OpenGL support, and facilitation of collaboration between developers and designers were all focussed by framework [34]. In terms of the development phase, the *Live Reload* feature in Fuse allowed for quick iterations when developing logic and interfaces. Not having to compile, build and re-deploy the application on every change was a time-saver, especially compared to native app development. The live reload functionality was, however, unreliable and occasionally slow. The prototype was developed in approximately four 7-hour days, including installing necessary software. Despite the lack of prior in-depth framework knowledge, Fuse made the developer experience encouraging.

4.4. Developing User Interfaces

The **React Native** framework is packed with components out of the box. There are 31 components documented by Facebook [46], some of which are platform-specific and named `<Component><Platform>`. The rationale behind this is the differences in implementation between Android and iOS. React Native does not rely on WebViews for user interfaces. Instead, the framework communicates through *native bridges* with the device APIs, resulting in actual native user interfaces. The result of this were 60 FPS (frames per second) high-performing interfaces [47]. Writing JSX syntax with predefined components would render native user interface components to the screen.

In **Ionic**, being a WebView-based *hybrid* app framework, user interfaces are developed in HTML, CSS and JavaScript. Ionic's additional CSS styling resulted in the user interfaces becoming native-looking, in other words the prototype app developed looked like a regular native app. The framework delivered a comprehensive set of ready-made components. Thus, complex interfaces could to some degree be developed simply by copy-pasting example code from the documentation. Logic was written in JavaScript using Angular.js which enabled an MVC design pattern.

In **Fuse**, user experience and interface development are core features. A custom XML-like language, *UX Markup*, is provided by the framework, giving easy access to a range of view components, event handlers, animation handlers, and such. An ever-expanding *Examples* web page filled with official examples and associated code was freely available through their website [48].

4.5. Code Reusability

All the prototype apps had 100% shared code bases for iOS and Android, with **Ionic** as the exception for additionally supporting the Universal Windows Platform. Code reusability is an important factor when choosing the cross-platform approach over native development, where the same app would require a separate code base for each targeted mobile operating system – allowing for a more rapid time-to-market. All tested frameworks provided excellent opportunities for reuse. However, an important factor to consider is maintenance over time as this must be evaluated over a significantly longer period due to API evolution and framework maturement.

4.6. Communicating with Device and Hardware APIs

React Native uses `bridges` to expose native hardware and device APIs to JavaScript. Such bridges were interpreted by the JavaScript interpreter on the device, JavaScriptCore. This allowed for native user interfaces and simple asynchronous hardware communication.

As **Ionic Framework** is a hybrid-approach framework, all the markup and logic was executed within a WebView. The WebView was able to communicate with hardware and device APIs through Cordova.

For **Fuse**, JavaScript is executed in JavaScriptCore on iOS and V8 on Android, allowing JavaScript to be used for writing logic on the client. Additionally, UX Markup-made user interfaces are compiled to native code ([34]). Because

the communication with hardware APIs was abstracted through such *bridges*, the implementation of native features in JavaScript is handled by callbacks or promises, making the communication strategy simple to follow.

4.7. Community and Popularity

At the time of assessment, **React Native** was the 20th most popular repository overall on GitHub, and the 9th most popular within the JavaScript category. In Facebook's *"A Year in Review"* blog post from 13 April 2016, the React Native team explained that *"In February 2016, for the first time, more than 50 percent of the commits came from external contributors"* and *"With so many people from the community contributing to React Native, we've seen as many as 266 new pull requests per month (up to 10 pull requests per day)"* [49]. Examples of pull requests to the React Native GitHub repository were documenting enhancements, performance improvements, bridged native components, and additional UI components.

The **Ionic Framework** is actively maintained. New features are delivered and fixes are provided on a regular basis [50]. According to Ionic.io, more than 2 200 000 mobile apps have been developed using the Ionic framework [10] as of 06 May 2016. This number might be so high due to including variants of apps; it demonstrates popularity nonetheless. The official Ionic websites also features a job listing [51] for external companies seeking Ionic developers. On the community side, Ionic maintained an highly active official forum board [52] boasting more than 105 000 users as of 06 May 2016.

The **Fuse** community forum and Slack channel are the predominant communication hubs for enthusiasts and developers using the framework. Both channels are actively used, mainly for support and development help. Due to the then closed-source nature of the project, no GitHub statistics were possible to present. This makes it impossible to assess statistics regarding community involvement. A search for *Fusetools* on GitHub returned 21 matching repositories containing sample apps and community-made modules. Searching for *Fuse* by itself returned more than 3 000 results, where no repositories were immediately identified as relevant to the framework.

4.8. Development Operating System Support

Operating system support is heavily influenced by target platform. Android apps can be developed on both Windows, OS X and Linux. IOS apps require OS X due to Apple restrictions. Furthermore, Windows Phone/Universal apps require a Windows machine to deploy and run the respective applications. A summary is given in Table 4.8.

All of the major mobile operating systems are supported, but their implementations vary in feature completeness in implementations of the different frameworks. This support must constantly be monitored, as constantly features are added and removed through the evolvement of the mobile frameworks. A summary is given in Table 4.8.

Framework	Windows	OS X	Linux
React Native	Yes	Yes	Yes
Ionic Framework	Yes	Yes	Yes
Fuse	Yes	Yes	No

Table 3. Supported Development Platforms

Framework	iOS	Android	Windows (UWP)
React Native	Yes	Yes	Yes
Ionic Framework	Yes	Yes	Yes
Fuse	Yes	Yes	No

Table 4. Supported Mobile Platform

5. Discussion

To provide a generalization, we discuss the evaluation, insights, and limitations gained from open questions.

The frameworks compared in this research were fundamentally different in some ways (paradigm, developer focus, or end-product focus). Where Ionic made development of interfaces and app flow easy through their component library, React Native left more architectural choices to the developer. Both ended up being well-suited for the prototype, with its views and interactive components.

While all the frameworks had good options and solutions for creating interfaces, there were clear distinctions between the approaches. While React Native mostly delivered unstyled and "not native-looking" interface components with the option of styling the components to fit the app, Ionic provided a massive library of ready-made and pre-styled components making the development of standard native-looking interfaces a breeze. Fuse mostly provided pre-styled components as well, but limited compared to Ionic.

While employing additional features such as GPS, contacts and camera access, the design and interface implemented in the prototypes were mainly based on lists and detail views, thus possibly making more hardcore UX and animation frameworks like Fuse less suitable.

The importance of open source as a factor for decision making should not be neglected as it may lead to technical debt if not accounted for since one are using libraries maintained by third parties. This is not uncommonly seen as a factor for increasingly challenging maintenance, increased costs and ultimately requiring rewrites of hardest infected code bases. The amount of third-party developed components and packages varied between the frameworks. Consequently, the prototypes complied with the requirements to a varying degree.

Maintenance and potential technical administration is also increasingly made more complex by several releases and new versions of the frameworks each year. The frameworks

in the cross-platform app development landscape are impacted by this by becoming a victim of limited compatibility, deprecation of APIs, and demands of new hardware to support additional features. All the frameworks analyzed suffer from this, but it is easier to handle for frameworks with large community user bases, rapid maintenance cycles, and mature architecture.

Scientific work typically has limitations. These can either be fundamental boundaries, or borders that could not yet be crossed. Despite the increasing literature footprint of cross-platform development, our paper still tackles a novel topic. This makes it particularly bound to limitations.

First, our work is limited by the choice of assessed frameworks. This choice is deliberate, as we attempt to complement the existing literature. However, we can only present a partial picture.

Second, the assessment is a snapshot due to the rapid proliferation of the field. Several updates of the frameworks were released in the meantime. This illustrates the impossibility to provide work that includes the latest developments while demanding rigor in assessments.

Third, our assessment focuses on technology. This allows for a deep insight into working with the frameworks, as e.g. illustrated by the code examples in Section 3. A qualitative and possibly also a quantitative assessment of the frameworks would need to be added. Admittedly, due to the complexity of the matter no papers exist so far that combine experimental, quantitative, and qualitative work *and* are broad in scope. This is a current boundary of the field as a whole. While these limitations are considerable, they do not impede the value of the presented work. In fact, our assessment is rigorous yet industry-relevant. Thus, the limitations can be seen as the foundation of future work.

It was interesting that the issues identified in previous literature were still very much relevant according to the survey results. Developers seemed cautious about adoption of cross-platform development tools mainly due to believed performance hits and potential loss in terms of end-user experience. Without rigorously testing apps developed using different approaches and paradigms on end-users, those are only speculations until future research has been conducted.

The main open questions still revolve around paradigm and framework choice. While the field matures, future research needs to keep on investigating for even more developer-friendly frameworks that sustain performance and user friendliness of apps, if not surpassing them.

Despite the expected high effort, research that takes into account a higher number of existing frameworks for comparison would be very valuable. As illustrated in Section 2.1 such work is yet to be seen. Moreover, we deem more research necessary that combines technical evaluations (i.e. experimental work), and developer and user impressions (i.e. qualitative work). Moreover, an extensive quantitative study would make a main contribution to the field.

6. Conclusion

We have presented a study and discussed issues related to success factors for cross-platform development. Our work includes an in-depth assessment of three novel cross-platform app development frameworks with industry-anchored research goals. Based on this real world industry case, our research started with an online survey questionnaire, receiving over 100 responses. The survey together with academic literature and discussions with the case company formed the basis for the requirements and the technical implementations. Findings from both prototypic development and survey were discussed and elaborated upon in the context of previous research and industry relevance. The three instantiations produced are novel artifacts used to compare the maturity and functionality of the frameworks. Moreover, they provide insights for practitioners through mapping survey findings to common issues identified in cross-platform app development.

We confirmed user experience, technical implementation, app performance, and testability to be the most common issues related to cross-platform app development. Furthermore, our combined knowledge from survey and artefact development highlights concerns related to user interface development (UI), remote data fetching, navigation, and developer experience as major success factors when comparing and selecting a cross-platform app development framework.

There are still many problems to tackle, such as the human side particularly in the form of end-user experience. Future work will include a longitudinal research study to investigate framework maturity, as well as an in-depth perspective on user interface design and interaction.

Acknowledgements

This work is based on the (not published) Master thesis of Andreas Biørn-Hansen, written at Westerdals Oslo ACT. Findings have first been condensed before we extended and revised the work. We would like to acknowledge our industry partner Acando AS. We have been provided with ample feedback that helped us to provide applied, relevant research and in general improve the paper.

References

[1] V. Woods and R. van der Meulen, “Gartner says worldwide smartphone sales grew 9.7 percent in fourth quarter of 2015,” 2016, http://www.gartner.com/newsroom/id/3215217.

[2] H. Heitkötter, H. Kuchen, and T. A. Majchrzak, “Extending a model-driven cross-platform development approach for business apps,” *SCP*, vol. 97, Part 1, pp. 31–36, 2015.

[3] T. A. Majchrzak, J. Ernsting, and H. Kuchen, “Achieving business practicability of model-driven cross-platform apps,” *OJIS*, vol. 2, no. 2, pp. 3–14, 2015.

[4] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Evaluating cross-platform development approaches for mobile applications," in *LNBIP*. Springer, 2013, vol. 140, pp. 120–138.

[5] J. Ohrt and V. Turau, "Cross-platform development tools for smartphone applications," *Computer*, vol. 45, no. 9, pp. 72–79, 2012.

[6] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, "Cross-platform model-driven development of mobile applications with MD^2," in *Proc. SAC '13*. ACM, 2013, pp. 526–533.

[7] T. A. Majchrzak and J. Ernsting, "Reengineering an approach to model-driven development of business apps," in *LNBIP*, vol. 232. Springer, 2015.

[8] J. C. Dageförde, T. Reischmann, T. A. Majchrzak, and J. Ernsting, "Generating app product lines in a model-driven cross-platform development approach," in *Proc. 49th HICSS*. IEEE Computer Society, 2016, pp. 5803–5812.

[9] "React Native," 2016, http://nuclide.io/docs/platforms/react-native/.

[10] "Ionic," 2016, http://ionic.io/.

[11] "fuse," 2016, https://github.com/fusetools/.

[12] M. Ciman, O. Gaggi, and N. Gonzo, "Cross-platform mobile development: A study on apps with animations," in *Proc. 29th Annual ACM SAC*. ACM, 2014, pp. 757–759.

[13] A. Connors and B. Sullivan, "Mobile web application best practices," W3C, Tech. Rep., 2010, http://www.w3.org/TR/mwabp/.

[14] H. Heitkötter, T. A. Majchrzak, B. Ruland, and T. Weber, "Comparison of Mobile Web Frameworks," in *LNBIP*, vol. 189. Springer, 2014, pp. 119–137.

[15] A. Charland and B. Leroux, "Mobile application development: web vs. native," *Commun. ACM*, vol. 54, pp. 49–53, 2011.

[16] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *IEEE ESEM*, 2013, pp. 15–24.

[17] M. Ciman and O. Gagg, "Measuring energy consumption of cross-platform frameworks for mobile applications," *LNBIP*, vol. 226, pp. 331–346, 2015.

[18] T. M. Grønli, J. Hansen, G. Ghinea, and M. Younas, "Mobile application platform heterogeneity: Android vs Windows Phone vs iOS vs Firefox OS," in *28th Int. Conf. on Advanced Inf. Networking and Applications*, 2014, pp. 635–641.

[19] N. P. Huy and D. vanThanh, "Evaluation of mobile app paradigms," in *Proc. 10th MoMM*. ACM, 2012, pp. 25–30.

[20] S. Dhillon and Q. H. Mahmoud, "An evaluation framework for cross-platform mobile application development tools," *Software – Practice and Experience*, vol. 45, no. 10, pp. 1331–1357, 2015.

[21] A. Hudli, S. Hudli, and R. Hudli, "An evaluation framework for selection of mobile app development platform," in *Proc. 3rd MobileDeLi*, 2015.

[22] T. Volkan and c. Z. Erdoğan, "Comparison of popular Cross-Platform mobile application development tools," in *2. Ulusal Yönetim Bilişim Sistemleri Kongresi (YBS2015)*, 2015.

[23] E. Angulo and X. Ferre, "A case study on cross-platform development frameworks for mobile applications and ux," in *Proc. XV Int. Conf. on HCI*. ACM, 2014, pp. 27:1–27:8.

[24] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proc. 6th BCI*. ACM, 2013, pp. 213–220.

[25] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein, "Survey, comparison and evaluation of cross platform mobile application development tools," in *Proc. 9th IWCMC*, 2013, pp. 323–328.

[26] A. Sommer and S. Krusche, "Evaluation of cross-platform frameworks for mobile applications," *LNI*, vol. P-215, 2013.

[27] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Comparing Cross-platform Development Approaches for Mobile Applications," in *Proc. 8th WEBIST*. SciTePress, 2012, pp. 299–311.

[28] M. Palmieri, I. Singh, and A. Cicchetti, "Comparison of cross-platform mobile development tools," in *Proc. 16th Int. Conf. Int. in Next Gen. Networks (ICIN)*, 2012, pp. 179–186.

[29] C. P. R. Raj and S. B. Tolety, "A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach," in *INDICON*, 2012, pp. 625–629.

[30] V. K. Vaishnavi and W. Kuechler, *Design Science Research Methods and Patterns*, 2nd ed. CRC Press, 2015.

[31] E. Taylor-Powell, *Wording for rating scales*. University of Wisconsin, 2008.

[32] "React Native Router," 2016, https://github.com/aksonov/react-native-router-flux.

[33] "React Native Drawer," 2016, https://github.com/root-two/react-native-drawer.

[34] "Learn fuse," 2016, https://www.fusetools.com/learn/fuse.

[35] "React Native: Geolocation," 2016, https://facebook.github.io/react-native/docs/geolocation.html.

[36] "Apache Cordova," 2016, https://cordova.apache.org/.

[37] "react-native-image-picker," 2016, https://github.com/marcshilling/react-native-image-picker.

[38] "React Native Contacts," 2016, https://github.com/rt2zz/react-native-contacts.

[39] "fuse-contacts," 2016, https://github.com/bolav/fuse-contacts/.

[40] "react-native-communications," 2016, https://github.com/anarchicknight/react-native-communications.

[41] "WebView," 2016, http://developer.android.com/reference/android/webkit/WebView.html.

[42] "React.js conf 2015 keynote – introducing react native," 2015, https://www.youtube.com/watch?v=KVZ-P-ZI6W4.

[43] P. von Weitershausen and D. White, "React native for android: How we built the first cross-platform react native app," 2015, https://code.facebook.com/posts/1189117404435352.

[44] T. Occhino, "React native: Bringing modern web techniques to mobile," 2015, https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/.

[45] "Ionic Components," 2016, http://ionicframework.com/docs/v2/components.

[46] "React Native: Getting Started," 2016, https://facebook.github.io/react-native/docs/.

[47] "React Native Performance," 2016, https://facebook.github.io/react-native/docs/performance.html.

[48] "fuse: Examples," 2016, https://www.fusetools.com/examples.

[49] M. Konicek, "React native: A year in review," 2016, https://code.facebook.com/posts/597378980427792/react-native-a-year-in-review/.

[50] "driftyco/ionic," 2016, https://github.com/driftyco/ionic.

[51] "Ionic: Latest Jobs," 2016, http://jobs.ionic.io/.

[52] "Ionic: Forum," 2016, https://forum.ionicframework.com.