# Mobile Cross-Platform Development from a Progressive Perspective

An analysis of NativeScript applications based on the characteristics of progressive web applications

**Nils Mehlhorn**
mail@nils-mehlhorn.de
Matriculation Number: 649194

Bachelor Thesis
B.Sc. Media Informatics

May 17, 2017

Hochschule Düsseldorf
University of Applied Sciences

Fachbereich Medien
Faculty of Media

*Supervisors*

Prof. Dr. Manfred Wojciechowski

Andreas Hartmann

## Declaration of authorship

*English*

I hereby declare that the paper submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published.

*German*

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt und indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht.

_____ _____

first and last name city, date and signature

## Abstract

Progressive web applications (PWAs) seem to be the next big thing in the mobile landscape. These are web applications with “super-powers” which are meant to provide the kinds of user experiences previously only native mobile applications could. Therefore they are able to match native applications in their capabilities but still thrive on the reachability of the web. They achieve this by implementing certain characteristics which originate from innovative standards and sophisticated best practices. Conveniently, as it is the web, PWAs are working seamlessly cross-platform. One of the closest things to a PWA so far may be an application created with the NativeScript framework. Such an application is also developed with web technologies yet able to use any native interface directly. This is made possible by using a designated runtime for mediating between JavaScript and the mobile system. Eventually, NativeScript is able to offer a high degree nativity yet also of convenient abstraction during development.

This paper is set out to deliver further insight into both approaches by contrasting them based on the characteristics of PWAs. For this purpose, these characteristics are elaborated and adjusted to be applicable to native mobile applications. Hence a reasonable basis for what to expect from a native application is derived. Then NativeScript is assessed on this basis by the means of transferable concepts and technologies as well as a prototypic mobile application. Finally, an informed discussion and conclusion is performed based on the results.

The comprehensive characteristics of PWAs resolve previous shortcomings with well thought out concepts and new technologies. These are transferable to native mobile applications to a far extent and may also be put into practice with NativeScript. The framework’s approach may be very well called ingenious, however, in the long run it might fall short of the innovative concept of PWAs. Having said this, it is still able to serve as a practical measure for creating appealing user experiences for multiple systems with arguably little effort. Due to large overlap a transformation between the two application types is a realistic option. Either way, web technologies are far from what they used to be and it is exciting to see how the mobile and web landscape will evolve in the future.

# Table of contents

# Figures

# Tables

## List of abbreviations

| | |
|---|---|
| AOT | Ahead-Of-Time Compilation |
| API | Application Programming Interface |
| APK | Android Package Kit |
| BaaS | Backend as a Service |
| DOM | Document Object Model |
| dp | device-independent pixel |
| DPI | Dots per Inch |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | HTTP Secure |
| JSON | JavaScript Object Notation |
| MVC | Model-View-Controller |
| PWA | Progressive Web Application |
| REST | Representational State Transfer |
| SPA | Single-Page Application |
| SSL | Secure Sockets Layer |
| TLS | Transport Layer Security |
| URL | Uniform Resource Locator |

# 1 Introduction

## 1.1 Structure

In the first chapter of this paper general conditions are laid down and a motivation for the paper's topic is established. Afterwards appropriate and necessary background information is provided in the second chapter. The third chapter contains the foundation for the analysis and its derivation. In chapter four the analysis itself is described and its results presented. Lastly, the discussion of the results is to be found in chapter five with a consequential conclusion and outlook in chapter six.

## 1.2 Goals

The substantive goals of this paper are outlined as follows

(a) The specifically covered field of mobile cross-platform development and development of progressive web applications (PWAs) is to be sufficiently defined and delimited. Furthermore, the general approach for a comparative analysis of both fields is to be described.

(b) PWAs as well as the characteristics of their development are to be illustrated. In doing so, the reasons or motivation for their emergence are to be presented in detail and aspects of popularity to be considered. In this context the concepts behind the characteristics are to be examined.

(c) The results from (b) are to be contrasted with the challenges from the field of mobile cross-platform development. Eventually a set of criteria for evaluation of the mobile cross-platform solution NativeScript is to be developed from the requirements for PWAs (insofar as applicable).

(d) The attainability of the set of criteria from (c) by NativeScript applications is to be analyzed by the means of transferable concepts and technologies as well as a prototypical implementation of a mobile application.

(e) The findings derived by the paper are to be summarized and weighed. The opportunities and areas of application of both approaches are to be comparatively debated. Moreover an outlook in regard to the foreseeable future progression is to be given.

## 1.3 General approach

In advance, intensive scholarly research is conducted. As PWAs are a rather recent emergence, a literary landscape is pretty much not yet existent. The books on the topic of PWAs used are in an unfinished state at the time of publication of this paper. Yet, other major sources regarding the topic are fully available online and therefore primarily consulted. Based on the gained insights and set goals, a scope for the paper is defined. Hereby it is intended to form an understanding about the covered field and avoid confusion about substantive aspects of the paper. Afterwards a problem statement is given to provide motivation for the topic itself. Subsequent to this introduction, the field of PWAs is illustrated by reflecting their definition, examining underlying concepts and describing their development. These findings are then considered regarding their practicality for mobile cross-platform development. This is done with respect to the eventual evaluation of NativeScript applications to form an assessable set of criteria. This assessment is consecutively conducted and includes the prototypical implementation of a mobile application. Hereby multiple aspects of PWAs are meant to be specifically reproduced in a proof-of-concept fashion. Furthermore, transferable concepts and technologies are to be assessed. The defined approaches for evaluation are conducted and the results presented. These results are then summarized and weighed in a discussion. Eventually a conclusion is formed and an outlook on foreseeable future progression in the field formulated.

## 1.4 Scope and delimitations

The term of “(mobile) cross-platform development” used in this paper refers to the creation of software products from one main codebase targeted to run on multiple contemporary mobile operating systems.

The main parts of this paper’s scope consist of the examination of PWAs and their concepts themselves as well as the assessment of NativeScript’s ability to fulfill the requirements made to the former. A fair amount of knowledge about web applications and mobile cross-platform development is assumed. This paper does not focus on the development of web or mobile cross-platform applications as self-contained topics, but is rather set out to contrast two commensurable approaches in these fields. This is done with a general regard to consumer applications.

To ensure a reasonable scope for the paper any integration concepts and implementations for an underlying mobile platform are limited to explanations regarding the Android system.

## 1.5 Problem statement and motivation

Put in simple terms, the goal of mobile cross-platform development is the creation of mobile applications which run on multiple platforms with ideally native experience while originating from just one codebase (1 p. 14). The software solution NativeScript offers promising prospects for fulfilling this goal. However, the development of NativeScript applications shows great proximity to how web applications are developed these days (1 pp. 34-35). Therefore it could proof to be worthwhile to approach the solution from a different angle. Yet, evaluating NativeScript on its capabilities for developing web applications when its output is meant to perform as native mobile applications would not be very meaningful. A recent emergence in web development provides a way more reasonable basis for analysis. With the introduction of PWAs a certain set of requirements is laid down regarding aspects of reliability, performance and engagement (2).

> *"Progressive Web Apps bring features we expect from native apps to the mobile browser experience in a way that uses standards-based technologies [...]" (3)*

Back when smartphones were introduces the web just could not compete with the features provided by native applications. By now, this shortcoming seems to be increasingly patched up (4 p. 9). Normally it would be difficult to argue what exactly is to be expected from a native mobile app. In this case, with the definitions for PWAs, specific metrics are in place to allow for convenient evaluation (5). With the combination of these concrete requirements and such a suitable candidate for evaluation as NativeScript, insights on aspiring concepts and approaches for working with foreseeable changes in the landscape of consumer application development may be gained.

# 2 Background

## 2.1 Progressive web applications

### 2.1.1 Emergence and classification

PWAs are actually not a technological innovation which was introduced from one day to the next. Rather the term is meant to group together several characteristics of really modern web applications. The adoption of new standards in browsers, especially the specification for service workers, allowed for a further evolution of web development (6). In 2015, the Google Chrome engineer Alex Russell took a closer look at these trending characteristics and labeled the corresponding web applications as progressive. According to him, this "[…] new class of applications […] deliver[s] an even better user experience than traditional web apps [are able to]" (7). This improved user experience is achieved by the usage of capabilities which ordinary web applications lack. Compared to the mobile web, users spent a considerably high amount of time on mobile apps (see Figure 1). The engineers behind PWAs substantiate this fact with the inferior capabilities of web applications on mobile. Without these, they would just not be able to be as engaging as native mobile apps were (8). After the native app is installed, it enjoys several benefits over its web-based complement. An icon on the user's home screen and push notifications allow them to reconnect with users in convenient ways. So far, web applications would not have these chances for making the user come back (4 pp. 12-13).

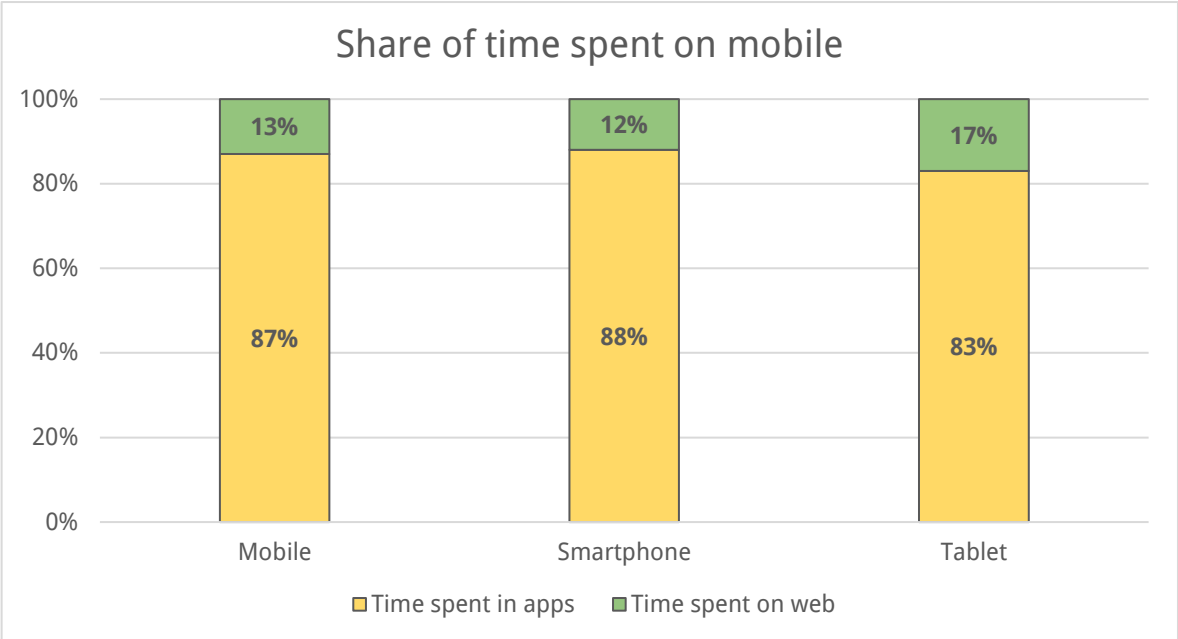


Figure 1 Share of time spent on mobile apps and web, data source: 9 p. 12

PWAs are meant to provide remedy. In the right environment they can perform in an app-like fashion. They are able to send out push notifications, launch in hardware-accelerated full-screen and use a wide range of sensors. Various innovative interfaces like the Push API or Geolocation API are making this possible (10) (11). Hereby, features which were previously reserved for native applications are made accessible in a standardized way. In concept, PWAs are meant to be on a par with native apps regarding their capabilities (see Figure 2).

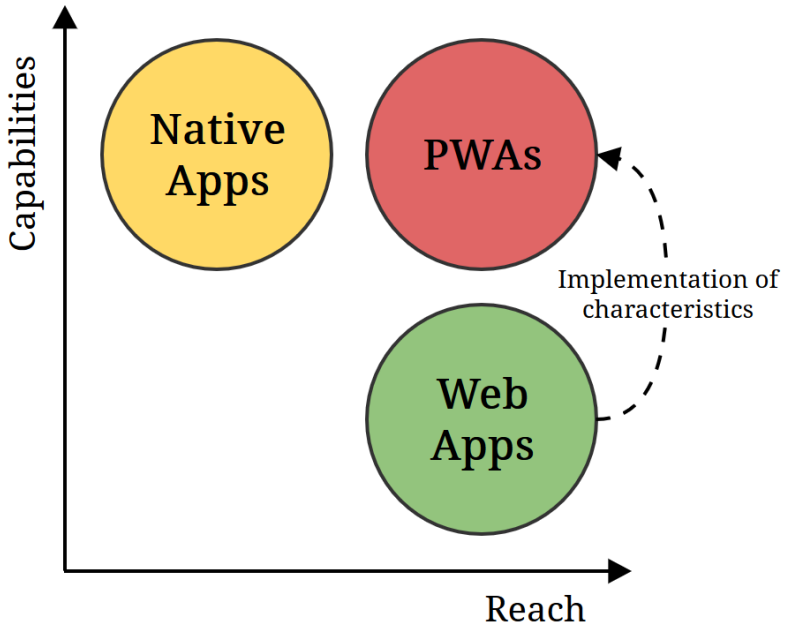


Figure 2 Envisaged classification of PWAs regarding capability and reach, inspired by: 8

The average monthly audiences of mobile web properties are about three times the size of the ones of comparable mobile apps. In addition, the mobile web audiences grow with twice the speed of their counterpart (9 p. 15). This means that web apps have a major advantage regarding reach on mobile systems. A PWA is meant to leverage this fact by being just that eventually: a web application.

> *"Establishing app audiences is harder, but their real value is in their loyalty." (9 p. 19)*

Although the web predominates in terms of audience size, the engagement of their audiences needs to be considered, too. As already stated, the time spent on apps is outpacing the mobile web by far. Users show far more engagement for them. Yet, this engagement is mostly limited to a handful of apps (9 p. 30). PWAs are supposed to combine the loyalty for native applications with the reachability of the web.

Several approaches in the field of cross-platform development stem from the incentive for building mobile applications with web technologies. Solutions like Apache Cordova bundles web resources into a native application wrapper. This workaround allows for building applications with the means of web development while leveraging native features (1 pp. 19-21). In theory, with PWAs it should no longer be necessary to deliver a web application in any kind of proprietary native wrapper. Implementing the defined technical requirements makes "[…] good old web sites […] exhibit super-powers […]" (12). Thus, the mobile browser itself takes care of filling the gap to the system. The result is a standardized solution for building web applications which may feel like native apps.

### 2.1.2 Characteristics

It becomes clearer which role PWAs are meant to play in the mobile landscape when looking at their characteristics one by one. Originally named by Russell, a PWA should be all of the following:

- Responsive
- Installable
- Linkable
- Fresh
- Safe
- Connectivity independent
- Re-engageable
- Discoverable
- App-like interactions

These are the dictated characteristics a web application needs to have in order to certify as progressive. Every characteristic is represented by certain technical properties. A subset of these properties forms a baseline for web applications to be detected as a PWA by a browser (5) (13). Google also provides a way for asserting many of the baseline properties automatically with a tool called Lighthouse (14). For the following elaboration the characteristics are organized according to the definitions on the Mozilla Developer Network as these arguably make up for a more suitable separation in this case (12). The associated classifications by Russel are listed accordingly.

#### Network independent (Fresh, Connectivity independent)

A central component of a PWA is the service worker. It represents a previously mostly non-existent instance between a web page and the corresponding server. The service worker is defined and registered via JavaScript for a single or multiple pages. After its registration it listens to events broadcasted by the web page (4 pp. 15-17). With this new instance in place, the offline state for web applications is meant to be improved. Instead of ending up with no functionality

at all in a situation with no network or low transmission rate, a reasonable offline experience can be delivered. The service worker is able to offer cached assets and data independent of network availability. In the best case the application can allow for browsing previously visited pages while displaying cached content. The bare minimum for the characteristic of being network independent would be displaying a custom offline page (13).

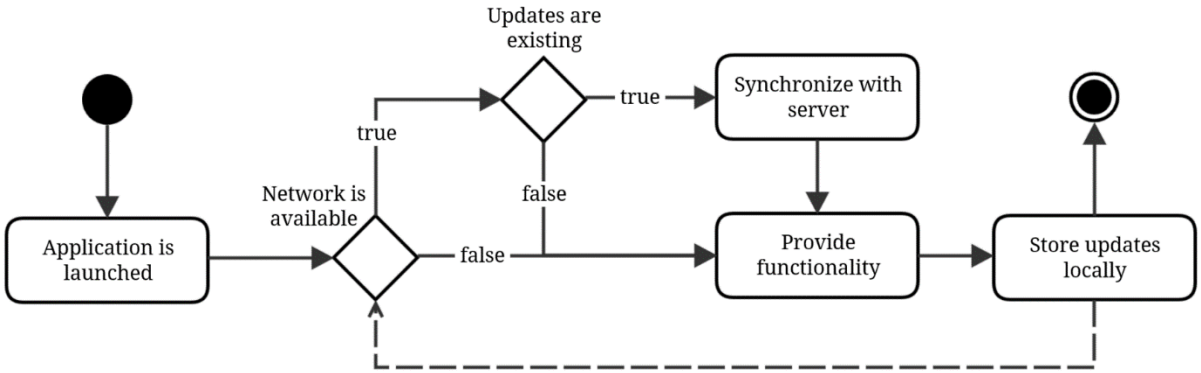


Figure 3 UML activity diagram for the workflow of an offline capable app implemented by service workers, inspired by: 15

As service workers guaranty a response for requests made by the application, be it populated with cached data, they need to intercept HTTP communications for being able to alter or replace their contents. To allow for this process to be carried out in a secure environment, web pages registering a service worker have to be served via secure HTTP (HTTPS). Otherwise man-in-the-middle attacks can take place (4 pp. 27-28).

### Safe

So PWAs have to be safe, thus use HTTPS, for service workers to work. But there are more reasons justifying a standalone characteristic of safeness. HTTPS is based on the successor of the cryptographic specification for the Secure Sockets Layer (SSL), called Transport Layer Security (TLS). It prevents tampering of web communications. Without the protocol in place, intruders may be able to access sensitive information or exploit the connection to insert advertisements, for example. But it is not just useful to secure sensitive connections with HTTPS. Even web pages which seem irrelevant from a security perspective can be violated for gathering usage data illegitimately (16). Implementing HTTPS even favors how a web page is ranked in search engines (17). Moreover, some web technologies may work even better on HTTPS connections (4 p. 28).

### Discoverable

Unlike native mobile applications, web applications do not have central points like app stores or market places for being discovered. Instead they may be discoverable via search engines or social media links. This plays into the aspect of reach. A central place for app distribution is limited in the number of apps it can represent efficiently and thus a new application "[…] can seem like a grain of sand on a beach" (18 p. 6).

> *"These apps aren't packaged and deployed through stores, they're just websites that took all the right vitamins." (7)*

However, PWAs may be market in a way more dynamic way. Any platform complying with the standards is able to handle them. A fundamental artifact for this characteristic is the web app manifest. This file contains metadata in the JavaScript object notation (JSON) format with essential information about the application. Assets like app icon or splash screens are defined and identification data like the application name or author is provided. Moreover an entry point for the application is to be specified in the manifest. The manifest compares very well to something like the application manifest for native Android applications. Just like it describes how the Android system should handle a packaged application, the web app manifest describes the PWA in a way any modern browser may be able to understand (18 pp. 3,6) (19) (20) (21).

### Installable (App-like-interactions)

The web app manifest is also required for making the web application installable. In this context installable refers to being able to add an icon to the user's home screen. This way the PWA can be started in a similar manner to native mobile applications. For this to work within Google Chrome on an Android system, a couple of requirements have to be met. Firstly, the manifest has to provide basic information about the app. Its name, a shortened name, an icon image and the already mentioned entry point in form of a Uniform Resource Locator (URL) are required. Furthermore, an appropriate display mode for a screen filling presentation has to be specified. Secondly, the application has to be able to start while offline. As already stated, this is to be ensured via service workers. Lastly, implicated by the service worker, the web site has to be served over HTTPS for being installable. Eventually, if the requirements are met and a certain degree of engagement is determined, the browser prompts for adding the PWA to the home screen (13) (22).

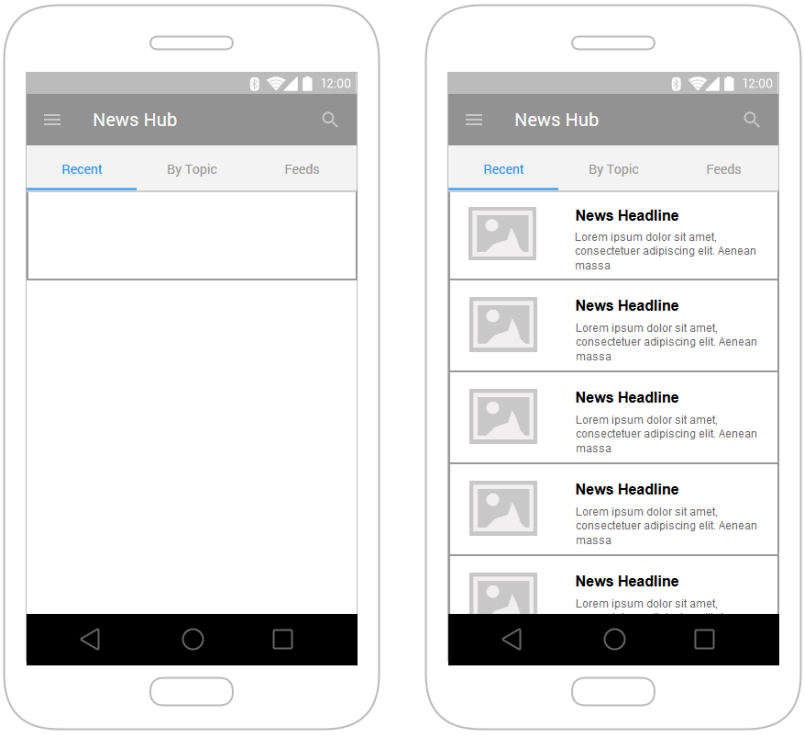


Figure 4 Illustration of a PWA's application shell without content (left) and filled with content (right)

A PWA launched from the home screen should execute in an application shell providing a fast startup. The shell may consist of static user interface elements which are cached during the installation. This guaranties a network independent provision of the application infrastructure which can then be filled with content (see Figure 4). With this separation of infrastructure from content the perceived performance and thus the user experience are enhanced (18 pp. 18-23) (23).

### Linkable

Having a discoverable PWA means it is approachable in terms of technical handling. It allows for accessing the app similar to how a native mobile app would be accessed. In contrast, the fact that PWAs are linkable refers to approachability from the web perspective. To use a specific feature of an ordinary mobile app it is required to be installed first. And even then the ways for accessing the feature might be limited.

Web applications can be accessed almost at any point without preliminary work by just having the right URL. The high reachability of the web thrives from this simplicity. A high amount of traffic originates from unassignable sources resulting in a phenomenon called "dark social" (24). While its extent might be

controversial, it shows how there may be access channels to an application which might not be seen in advance. PWAs leverage this aspect by working without installation and therefore being easily shareable (7).

### Re-engageable (App-like-interactions)

As pointed out, the engagement for web applications is rather low. PWAs are meant to overcome this. With an icon sitting on the home screen and push notifications they are supposed to re-engage the user just like native apps can. Google's model example for this characteristic is the e-commerce site Flipkart. When their web presence was relaunched as a PWA, the time that users spent on the site tripled. More than half of their users now visited the site via the home screen icon. Among them the conversion rate would be 70% higher compared to the average user (25). Flipkart substantiates these numbers with the use of the new technologies. Based on the Push API they were able to send messages to their clients' service workers, which in turn would then notify the user via the Notifications API. As the service workers "[…] live beyond the lifetime of the browser" (26), such ways of interacting with the user would now be possible for web applications (26).

### Responsive

> *"PWAs are quickly becoming a set of best practices. The steps you take to build a Progressive Web app will benefit anyone who visits your website, regardless of what device they choose to use." (18 p. 4)*

One of the best practices brought together by PWAs is the one of responsive design. Mobile-first approaches changed the way websites are designed substantially, and one might say rightfully so. Today, two thirds of the time spent on digital media takes place on mobile systems (9 p. 6). This grants valid reason for making an effort to deliver highly mobile friendly websites. With a heterogeneous landscape regarding device specifications, the goal has to be a proper display regardless of form factor. This is commonly achieved by using media queries and advanced features of CSS like device adaption and the flexible box layout (27).

### Progressive

Just like a PWA should display properly on any device regardless of its form factor, it should also work properly regardless of the browser in use. Not everyone is able to keep pace with the mentioned emergence of advanced interfaces. For example, service workers are not yet overall supported by common browser vendors (28). The established web development principle of progressive enhancement is used to deal with this circumstance. Similar to mobile-first, it relates to building a web application by starting with a small but working foundation and then layering more functionality on top. The usage of advanced technologies (e.g. low-level APIs such as the ones for push notifications) would be admissible as long as some kind of fallback is provided. Thereby, the website is made "[…] more accessible to all audiences" (4 p. 27).

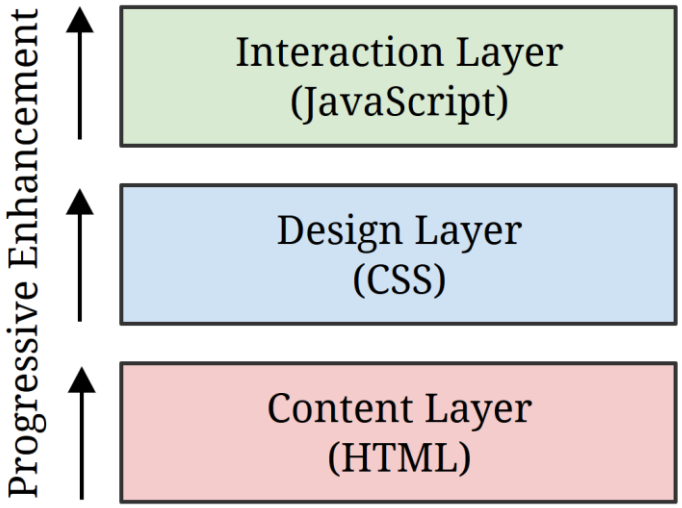


Figure 5 Illustration of layered view on progressive enhancement

The principle is often described as being layered. The base layer would be made up by the site's content with subsequent layers for design and interaction. This way, even clients that are not supporting JavaScript or CSS may get access to a site's content. The metaphor can also be applied within these layers where different levels of support may be present (see Figure 5). For example, some browsers may be proficient in JavaScript but may not implement the most recent standards (4 p. 27). With progressive enhancement these clients will still be able to deliver less yet reasonable functionality. Regardless of a client's currency, the website may not be completely broken.

Hereinafter, all characteristics of PWAs are listed collectively with a certain described intent and the technical properties used to instantiate them. The characteristics Linkable and Progressive lack specific technical properties as their implementation is innate to web development itself. With them, it is rather about the definition of their concept than the emergence of a new technology.

Table 1 Characteristics of PWAs, each with associated intent and technical properties

| CHARACTERISTIC | INTENT | TECHNICAL PROPERTIES |
|---|---|---|
| *NETWORK INDEPENDENT* | Enhance offline experience | • Service workers |
| *SAFE* | Secure communication | • HTTPS |
| *DISCOVERABLE* | Identification of applications | • Web app manifest<br>• Service worker registration |
| *INSTALLABLE* | Enhance performance and user experience | • Web app manifest with required information<br>• Service worker registration<br>• HTTPS |
| *LINKABLE* | Allow easy access for reachability purposes | *Innate to the web* |
| *RE-ENGAGEABLE* | Enhance user engagement | • Push API<br>• Notifications API<br>• Service workers |
| *RESPONSIVE* | Allow for proper display on different form factors | • Media queries<br>• CSS device adaptation<br>• CSS flexible box layout |
| *PROGRESSIVE* | Allow for an acceptable execution in any browser | *Innate to the web* |

### 2.1.3 Notable aspects of development

As mentioned, PWAs are no singular technology but rather a combination of existing concepts. Therefore no further dependencies are dictated beside the standardized browser interfaces. Developing a PWA does not differ from developing any other web application in terms of the tools used. Although there are several solutions available which may help with the task, the only mandatory requirement is the implementation of the characteristics by using the associated technical properties (see Table 1).

As a matter of fact, the development of PWAs is somewhat progressive in itself. There is no particular need for redeveloping a web application from scratch to make it progressive. Rather small steps can be taken to implement one characteristic after another. Thereby the capabilities of the web application and thus the user experience are successively enhanced (18 p. 16).

## 2.2 Mobile cross-platform development with NativeScript

Web applications, which PWAs are in the end, rely on standards and web browsers supporting these standards. Hereby they can run on any platform with a sufficiently capable browser. For native mobile applications such a principle is not in place. Google's Android and Apple's iOS are the market leaders for mobile operating systems (29). When wanting to deploy an app to these platforms, one would have to carry out two considerably different software developments. With Android development being performed with Java and XML whereas iOS applications are developed with Objective-C and Swift, one ends up with two separate code bases. This means the application has to be written twice while delivering ultimately the same functionality. Cross-platform solutions are used to avoid such expensive development overheads. These solutions allow for serving applications for multiple platforms at once from a single code base.

Yet, the differences between platforms do not exists only on a programmatic level. Mobile users are accustomed to a specific user experience on their device. Appearance and behavior need to conform to their expectations. Therefore a lot more is needed when transferring an app to another platform than just rewriting the code in a different language. Several aspects of developing native applications have to be abstracted whilst ideally delivering the same result. Cross-platform solutions have the goal of delivering applications which perform like they were developed natively while being significantly more efficient in regard to the application development itself.

The NativeScript framework addresses the requirements made to mobile cross-platform solutions with a unique approach. Just like the already mentioned Apache Cordova, NativeScript leverages certain web technologies to allow for abstracted development. Application code is written using JavaScript, but instead of recreating user interface components with HTML, the framework instantiates native view elements. These are typically defined in a declarative way using XML. Styling and animations are to be applied using CSS. Not just the ones for view management but any system interface is available within NativeScript applications. This is enabled by a platform-specific runtime which is bundled with the application code and provides access to the system runtime directly from JavaScript (30 pp. 7-9) (31).

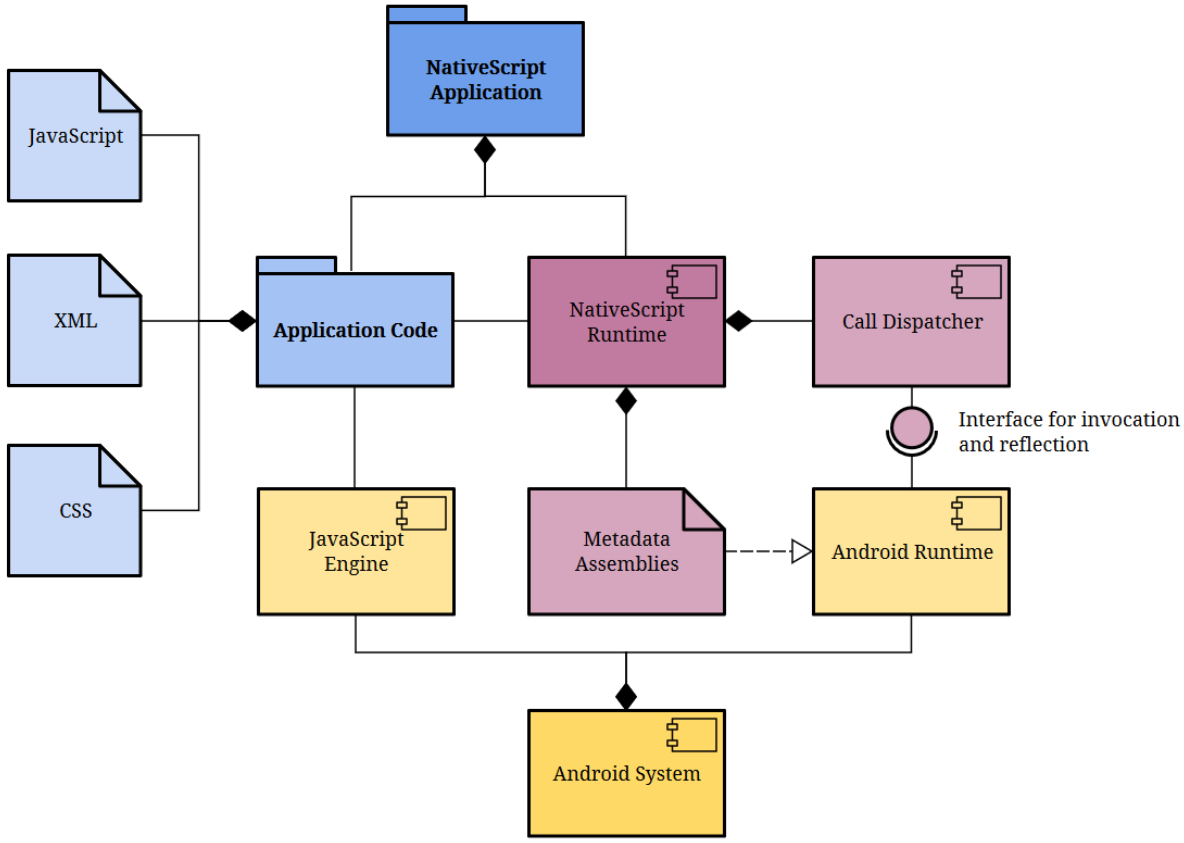


Figure 6 UML component diagram illustrating the architecture of NativeScript applications and their integration with the Android system

The application code executes inside the system's JavaScript engine. Preliminary to its execution, proxies for the system interfaces are injected into the JavaScript namespace. These are JavaScript objects which are connected to appropriate Android types beforehand generated during build-time. Any call made to the proxies is forwarded to the associated native complement and in return any changes originating from the system are reflected upon the JavaScript objects

(see Figure 7) (31) (32). With this mechanism the app executes very close to how a native app would do, resulting in a high nativity. This in turn accounts for a relatively good performance of NativeScript applications, which are eventually able to provide a decent user experience (1 pp. 30-33).

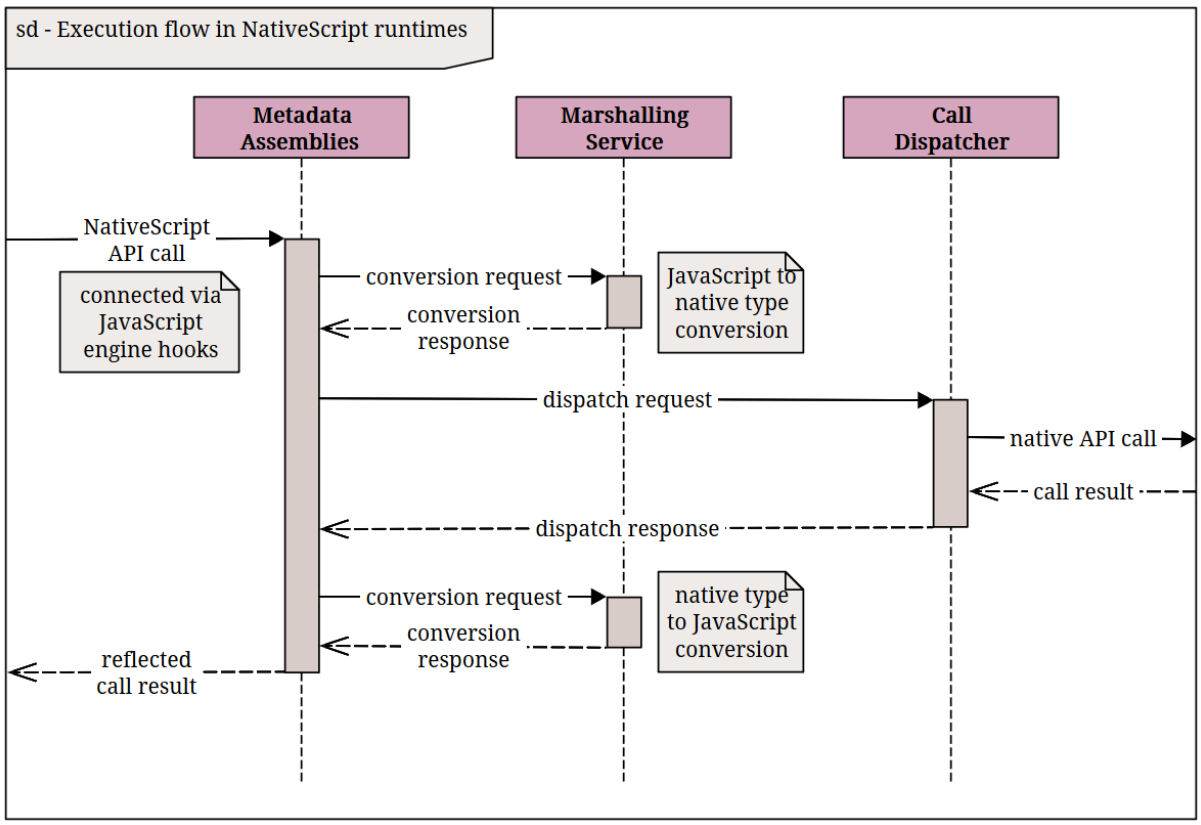


Figure 7 UML sequence diagram illustrating simplified execution flow in NativeScript runtimes

The calls to the specific system interface are usually wrapped into designated modules. This way the application code can be written in a more abstract fashion. The modules are meant to distinguish between the systems interfaces while the business logic can be agnostic of the underlying platform (33 NativeScript Modules). In combination with single-page application (SPA) frameworks the development of NativeScript applications can be performed very similar to web developments. Frameworks like Angular enforce concepts such as separation of concern or the Model-View-Controller (MVC) pattern. With these, a proper architecture can be constructed which encapsulates accesses to the platform in a maintainable way. NativeScript approaches platform integration and application architecture in a way that offers a high degree of nativity regarding execution but also of abstraction during development (34).

# 3 Developing a progressive perspective

## 3.1 Preliminary considerations

The presented solutions may differ in their starting point, yet they overlap in their general intent. The concept of PWAs is supposed to elevate web development to a place where it is able to compete with native implementations. Mobile cross-platform development, as represented by NativeScript, aims to mitigate the need for developing natively and therefore developing multiple times. Both solutions are set out to replace native development of mobile applications. While PWAs are coming from an abstracted point which gains instantiation through new standards, NativeScript takes the matter into its own hands. The execution takes place on an unrivaled low level, however the result is an installable application and ideally very similar to native ones. PWAs are just similar to native applications in regard to their capabilities, yet meant to be more versatile. Still, similar ways are used to reach similar goals. The actual application development is emphasized while the execution is meant to be handled by the appendant infrastructure. With an appropriate architecture the similarities and differences become apparent. Commonly, SPA frameworks are used in both cases, thus the solutions align on a logical layer. At the same time wide disparities regarding presentation and system connection are present (see Figure 8). In the eventual application usage these differences may be potentially irrelevant as both approaches are set out to deliver sophisticated user experiences. However, in regard to the technical result it may be very much relevant which approach is more convenient or sustainable.

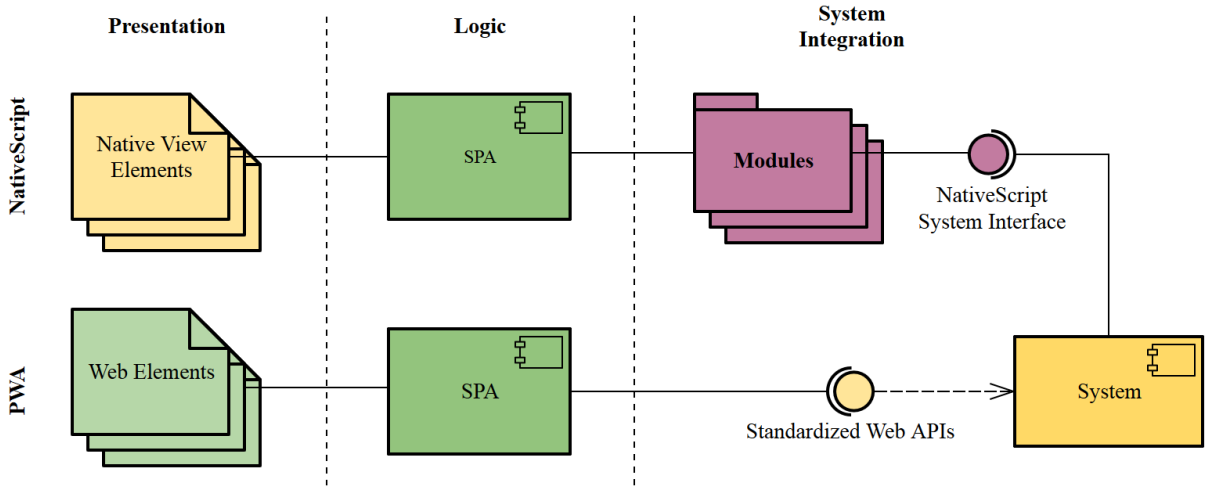


Figure 8 UML component diagram illustrating differences and similarities between progressive web and NativeScript application architecture layers

## 3.2 Criteria derivation

Hereinafter, the criteria for assessing NativeScript by will be derived from the characteristics of PWAs (see Table 1). This is to be done by mapping out how these characteristics may apply to natively running mobile applications. Due to the differences between both solutions, some characteristics may not be transferable at all or only in a modified way. Omissions or modifications as well as underlying characteristics for a criterion are noted accordingly.

### Network independence (Characteristic: Network independent)

Installed applications are innately network independent to a certain extent. The resources for displaying blank user interface elements are allocated as a result of the installation. Therefore a native application would be able to execute regardless of network availability without any further preparations. Yet, the actual goal of the associated characteristic is enhancement of the offline experience. The corresponding criterion therefore may refer to the provision of a meaningful behavior or functionality when no network connection exists.

Without network independence of any kind, a web application will end up with a generic browser error message when the network is offline. This may be rated as an arguably substandard offline experience. If an installed application does not implement any offline workflow (alike illustrated in Figure 3) it may offer a similarly or even worse result as the web counterpart. Exemplary, a generic error message or an application crash would definitely imply a failure in regard to the criterion. Instead the application may provide useful data from a previous session and allow for data input which may be synchronized later on.

Obviously, the application's general functionality has to be essentially network dependent for the criterion to be applicable. An application which might work solely in an offline state would not provide for a meaningful assessment.

### Security (Characteristic: Safe)

PWAs are safe through the usage of HTTPS for all internet communication. This aspect is easily transferable into a criterion with the same goal and similar measures. Although the application itself is not provided over the web, the concept can be applied to its server communication. Besides the already mentioned application resources acquired during installation, any subsequent server requests may use the proper protocol for secure communications.

### Marketability (Characteristics: Discoverable, Installable)

Discoverability can hardly be transferred in a meaningful way as the products of NativeScript developments are supposed to resemble native applications and are therefore already identifiable as such. A web app manifest cannot be assessed, yet its native complement the already mentioned application manifest will be present for the Android platform. Therefore, the requirement derived from the characteristic may rather refer to the existence of proper metadata in such a manifest. This also serves towards a possible publication of the application. For a release on the Google Play store, the application's name, id and package name may be provided (35).

For the same reasons a similar modification has to be applied to the characteristic regarding installation. The application metadata is necessary for building a valid Android application, thus making it installable. Yet, the registration of a service worker is not suitable within a native application as it may use the platform's threading model (36). Furthermore, resources like splash screens and application shell may be already present before the first time the application is started. Due to these considerations, both characteristics may be transferred into a criterion of marketability. The goals of application identification and enhancement of user experience are meant to further the application's approachability in this case. By having a properly configured application with appropriate metadata, one may be in a good position for publishing it. Through this, compatibility issues, licensing concerns and other aspects of production readiness may be streamlined (37).

> *"Branded launch screens provide momentary brand exposure, freeing the UI to focus on content." (38)*

At the same time, by following the launch screen pattern an elegant initial impression is to be conveyed. Memorable brand assets may be displayed during a cold launch, meaning the initial execution. This way brand recognition can be enhanced (38).

### Linkability (Characteristic: Linkable)

While PWAs are made linkable fairly easy due to their residence on the web, this characteristic is hardly found in native applications. These applications are not on the web, thus they are not able to profit from its innate reachability. Yet, with the right preparations it may be reproduced. Android applications may define certain ways for being accessed, namely intent filters. Usually used for inter-app

communication, they can also allow for deep linking to specific parts of an application. Just like on the web, the desired content can be reached by having the right URL. Such a link will redirect a user to a certain activity inside the application, thus allowing for the application to be called linkable (39). Moreover, a linkable application becomes accessible to search engines. These may crawl the application's content and subsequently offer new convenient ways for reaching them (40).

### Re-engagement (Characteristic: Re-engageable)

Web applications are gaining capabilities for reengaging the user through emerging interfaces. Native applications already have access to all these capabilities. As of their installation they present themselves on the user's home screen in form of an application icon and therefore may be revisited more frequently. Push notifications can be sent via a combination of a designated cloud messaging API and client implementations for displaying them (41) (42). So, the possible ways for reconnecting with the user are basically the same. Therefore the resulting criterion is analogue to the corresponding characteristic. The user experience is to be enhanced by dispatching push notifications and thus repeatedly initiating interaction with the application.

### Responsiveness (Characteristic: Responsive)

In contrast to web applications, native mobile applications are virtually never initially designed for desktop computers. Therefore the question of mobile friendliness may not be applicable. Yet, these applications still have to be responsive in a similar way. They also face a variety of screen sizes and resolutions which they shall adapt to. In cross-platform developments, this aspect may weigh even heavier as even more devices are meant to be supported by the same application. The ways for making responsive web applications are standardized while "each of the three major mobile platforms (Android, iOS and Windows) offers its own way of dealing with device fragmentation and loading resource files (layout, images etc.)" (43). As a result, certain mechanisms have to be in place for dealing with different hardware specifications. These need to eventually integrate with the platforms approaches for ensuring responsiveness or provide different ways of adapting the user interface appropriately.

### Progressiveness (Characteristic: Progressive)

For native implementations the barrier of standardized APIs is non-existent and therefore any device capability provided may be used in an application. Yet, this

also leaves the problem of handling different feature sets to the application code itself. Backwards compatibility has to be ensured so that the target audience is not limited to a certain system version or specific device. On the Android platform, this aspect is generally covered by the usage of designated support libraries. These allow for "[…] backward-compatible implementations of import, core platform features" (44) and may be used to "[…] create more modern app interfaces on earlier devices" (44). Ideally, these ought to be leveraged by a NativeScript application. Alternatively, different approaches for providing progressive behavior may be applied.

Table 2 Evaluation criteria derived from the characteristics of PWAs, each with associated objective and functional requirements

| CRITERION | OBJECTIVE | FUNCTIONAL REQUIREMENTS |
|---|---|---|
| *NETWORK INDEPENDENCE* | Provision of meaningful offline experience | • Presentation of cached data<br>• Subsequent synchronization |
| *SECURITY* | Protection of server communication | • Usage of HTTPS |
| *MARKETABILITY* | Further approachability and production readiness | • Display of launch screen<br>• Provision proper application metadata |
| *LINKABILITY* | Converge to web in terms of reachability | • Deep linking support |
| *RE-ENGAGEMENT* | Enhance user experience by initiating anew interaction | • Dispatch of push notifications |
| *RESPONSIVENESS* | Allow for proper display on different form factors | • Integration of platform mechanisms for responsiveness or alternative |
| *PROGRESSIVENESS* | Allow for an acceptable execution on different system versions or devices | • Integration of platform mechanisms for backwards-compatibility or alternative |

# 4 Assessing NativeScript

## 4.1 Prototype

The laid down criteria is now to be assessed by the means of a prototypic NativeScript application. The concept for this application is loosely based on the Kanban method used in lean software development approaches. More precise, the application is meant to allow for working with Kanban boards. Kanban boards are tools for managing individual tasks in form of cards. These cards are organized in multiple columns or lists (see Figure 9). Usually, the cards are moved from left to right over the board as the corresponding project tasks are processed (45 pp. 87-89).

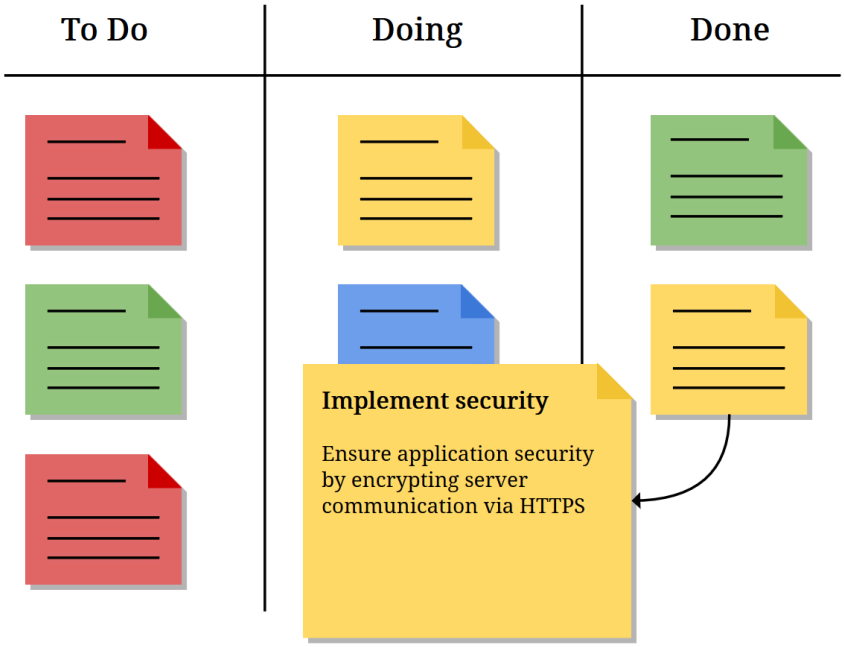


Figure 9 Simplified illustration of a Kanban board

The actual Kanban method enforces arguably strict rules upon then operation of the board which are not transferred to the prototypical application. Instead, a more liberal approach for interpreting the method is taken. The ability of boards to visualize tasks in a convenient and useful way is emphasized. Thus the application is rather meant to be used along the lines of methods like Personal Kanban (46).

To determine basic functional requirements for the application, the general interactions with a Kanban board are transformed into exemplary use cases (see Figure 10). The most obvious one is the creation of a board itself. Such a board is subsequently managed by, among other things, the creation of lists and cards.

The lists are holding the actual cards which may be moved from one list to another. Several more features come to mind for managing boards, lists, and cards, like updating their properties or deleting them. These are excluded here as they may be omitted as major use cases, yet the application is meant to implement such features eventually.

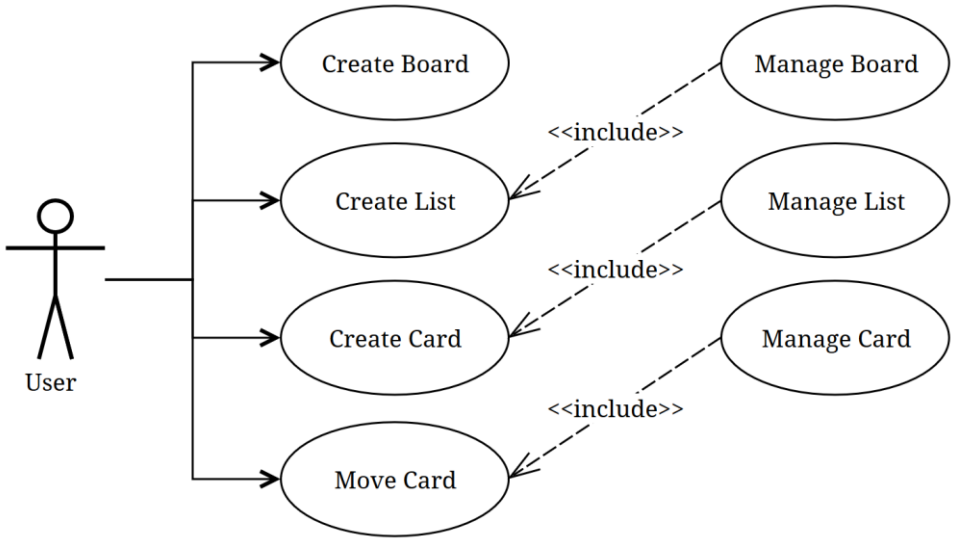


Figure 10 Partial UML use case diagram for the Kanban board application

Implementation of the mobile application is performed with NativeScript 2.5, consisting of the corresponding Android runtime, the core cross-platform modules and command line interface (CLI). This setup is complemented by the usage of the SPA framework Angular in its fourth version. As a result, the application architecture aligns with the common one of PWAs in the desired way (compare Figure 6). The Angular framework is written in TypeScript, a typed superset language of JavaScript (47). The language is also used for large parts of NativeScript itself ensuring "[…] integration with all NativeScript APIs and even all native APIs when you use TypeScript" (48). Out of convenience, the application code is also implemented with TypeScript (48) (49 pp. 4-5).

The prototype consists out of three main views. The entry point for the application displays the existing boards and allows for the creation of new ones. Selecting a board leads to a view for its detailed representation (see Figure 12). Here the board's lists are laid out with compact depictions of the cards they contain. In this view, additional lists and cards can be added to the board. Clicking a card brings the user to its detail view (see Figure 11). Here the user

may edit the card's description or provide a picture for it. The card may also be renamed or moved to another list at this point.

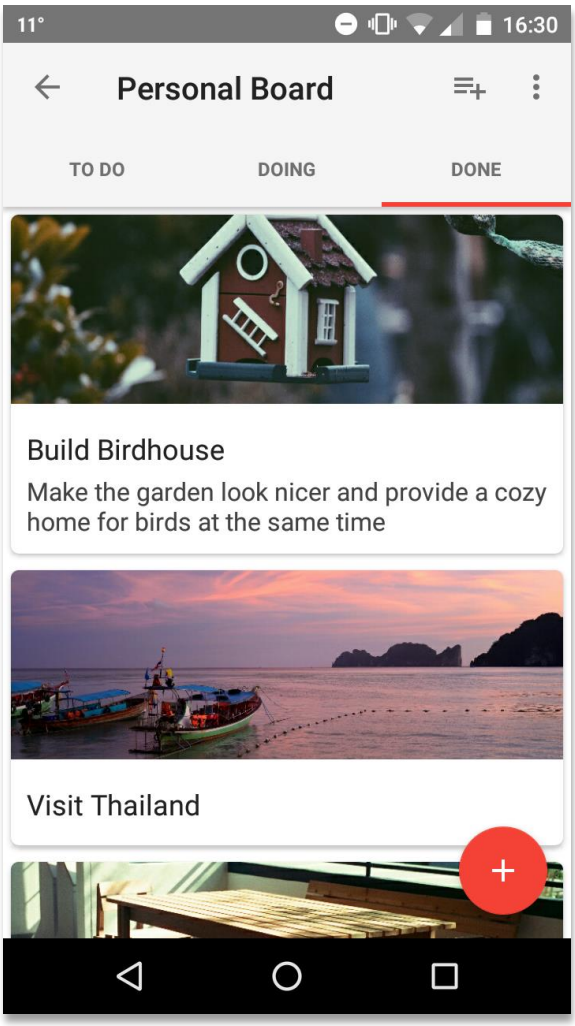


Figure 12 Detail view for single board in the Kanban board application on Android

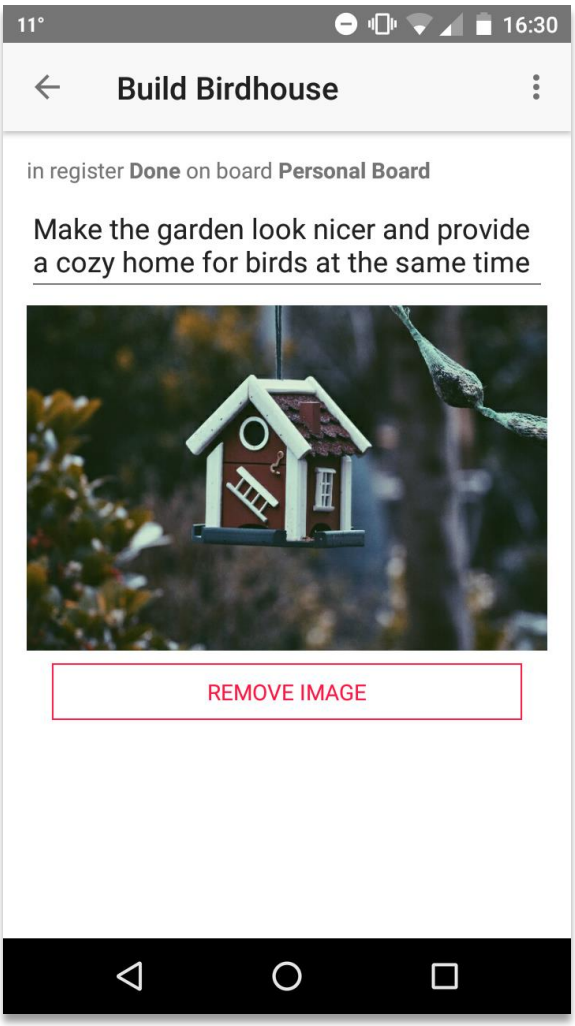


Figure 11 Detail view for single card in the Kanban board application on Android

The previously defined functional requirements for the prototype are not only dependent on client-side implementations. They also require extensive backend support to implement features such as push notifications. To streamline the prototype development, the backend as a service (BaaS) provider Firebase is used. This platform provides a real-time database and several other cloud-based services such as for messaging or file storage. Firebase is used in a design pattern where it acts as the single source for dynamic content. Consequently, a two-tier architecture is formed where the mobile application itself may only consist of static assets and is directly connected to the real-time database (see Figure 13). Database security and consistency may be achieved through the use of rules which are to be defined on the data schemes. The BaaS may also be extended by custom backend code to adapt it to the application's needs (50) (51).

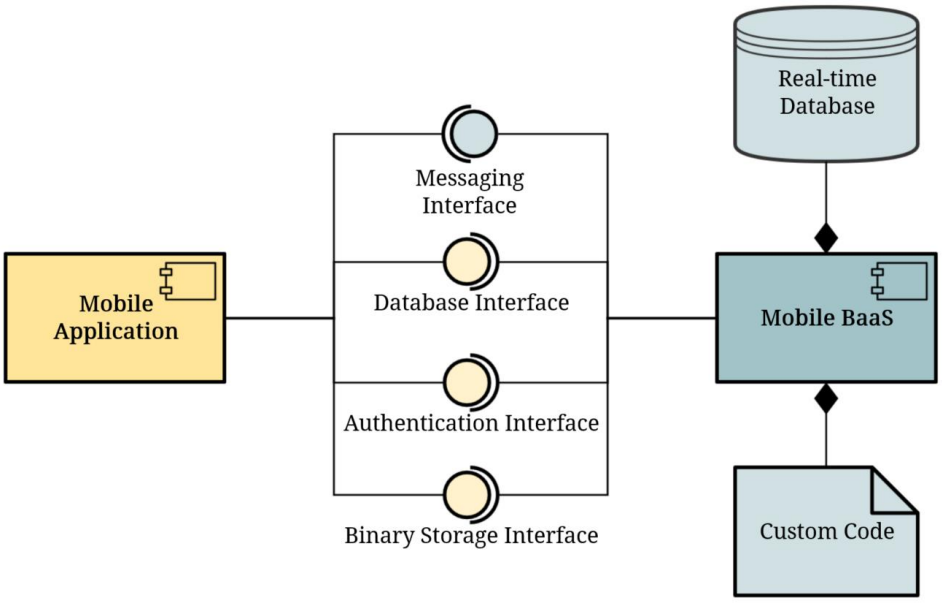


Figure 13 UML component diagram illustrating simplified two-tier architecture for the Kanban board application

## 4.2 Implementation of functional requirements

Due to the extensive utilization of Firebase, a designated NativeScript plugin is employed for its use. It encapsulates the official Firebase SDKs for the respective mobile platforms. As mentioned before, this is a standard approach within NativeScript which is combining the benefit of being able to access any native interface with a proper modularization. Eventually, this results in high nativity as well as a low amount of platform-specific code. Through this, the SDKs are made accessible directly from TypeScript in a platform-agnostic way (52). This setup allows for seamless integration of the cloud services provided by the backend, thus building the foundation for the implementation of several functional requirements described below.

### Presentation of cached data and subsequent synchronization

> *"Firebase apps automatically handle temporary network interruptions. Cached data is available while offline and Firebase resends any writes when network connectivity is restored." (53)*

A specific Angular service is presented with the task of mediating between the services which are handling the application's business logic and the abstracted Firebase SDK. Server requests are resolved asynchronously and will either return freshly retrieved server data or a cached representation of the former. For this to work, the application will cache all retrieved data on the devices disk until a size limitation of 10MB is reached. At this point, any new arriving data

will overwrite the least significant parts of the cached data. Any write operations made during an offline state will also be cached and subsequently performed at a later point when network connectivity is restored.

The process described above will take care of supplying the primary application data, referring to information about boards, lists and cards. However, assets like the images attached to cards (as introduced on p. 30) are handled by retrieving the source URL from the BaaS and passing it to a designated component for their display. Instead of the ordinary NativeScript component for displaying images an enhanced version is used which utilizes native caching libraries. For Android, the Facebook Fresco library is employed (54). This allows for a platform-agnostic image presentation in a web-like fashion whilst leveraging highly performant native image processing capabilities.

### Usage of HTTPS

When using Firebase as a backend provider, the server communication has to be encrypted either way. Initially available, all REST endpoints are disabled for usage with plain HTTP by now. Furthermore, all JavaScript clients are configured to use HTTPS "[…] regardless of the protocol specified in the Firebase database URL" (55). As Firebase is the only web endpoint used within the application, all server communication is consequently protected by proper encryption. A valid HTTPS certificate is provided by the BaaS and communication is secured by cryptographic operations of the respective SDKs (55) (56). However, these are not open source, meaning this process cannot be completely retraced. Only, the usage of an open source component for the HTTP communication is hinted for the Android SDK (57).

Alternatively, the usage of HTTPS could be implemented through a renowned plugin for NativeScript which wraps open source native implementations and is also open source itself (58).

### Provision of proper application metadata

Provision of the necessary metadata for Android applications is done through the Android application manifest. This file is accessible during the development of NativeScript applications for performing platform-specific configurations. However, many cases will not even call for its manual modification. During build time, it is augmented with the most vital metadata by the framework and may not require further treatment for the application to work (59) (49 pp. 20-

22). Among other things, package name and application id, both needed for publishing, are generated from the NativeScript build configuration (60).

During the prototype development, a few adjustments are made to the Android manifest to enable the implementation of all functional requirements. For instance, Android specific intent filters and service definitions (respectively required for deep linking and push notifications, see below) are declared to connect the application code to certain system hooks.

### Display of launch screen

The launch screen is the first view a user will see when starting the Kanban board application. It is displayed to bridge the time gap which occurs when the NativeScript runtime is preparing for executing of the actual application code. In this state access to the NativeScript APIs is obviously severely restricted. Therefore the launch screen has to be configured in a platform-specific manner. A branded launch screen is implemented which displays the application's icon and name in a simplistic manner. This is achieved by defining an appropriate layout with native view components. For Android, a view definition is composed via XML and placed into the application's resource location for the specific platform. During launch the NativeScript framework will pick up the view definition and display it until all preparations are complete. Afterwards the control flow is passed to the Angular application (61) (30 pp. 290-295).

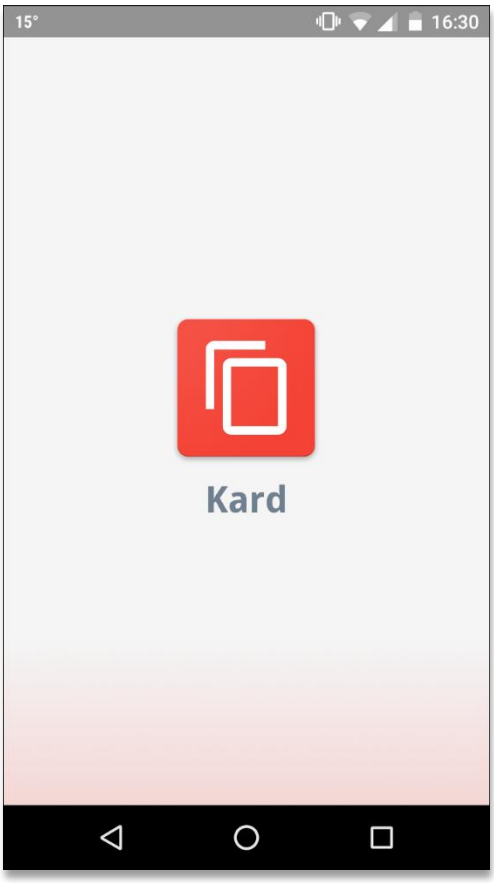

Figure 14 Launch screen for Kanban board application on Android

Regardless of the effort put into the launch screen it is desirable to have it show up as brief as possible. This way the user experience may not be impaired right at the application start. Therefore the startup time is optimized by leveraging Angular's Ahead-Of-Time compilation (AOT) capabilities with the Webpack module bundler (see also appendix B) (62).

### Deep linking support

The apparent use case for a deep linking functionality may be the standardized sharing of individual cards or whole boards through a link. By this, the dynamic contents would be made accessible outside of the application's initial scope. References to specific application parts may be forwarded independently and eventually lead back to their actual representation.

Due to the proximity of the implementations to the ones of ordinary web applications, a convenient gateway is already in place. The Angular component router allows for URL-based navigation within the application itself. Every page of the mobile application is indexed by a specific route. The corresponding components are attached to the routes and will be loaded upon valid navigation (63). Therefore, if such a navigation route is transferrable to the router, the deep linking mechanism may be functioning without further ado. Yet, getting the route to the Angular application, however, requires platform-specific handling of inter-app communications.

For the Android platform, the application's main activity is extended within TypeScript by leveraging NativeScript's proxy mechanism for Java classes (64). Using this in combination with a designated intent filter, incoming Android intents may be intercepted, thus providing access to data passed to the application by the platform. At this point, other applications on the platform are able to invoke application routes by using specific data URLs. These are shaped after the pattern illustrated below.

*Android data URL pattern*

```
<application-scheme>://<route>
```

*Android data URL example linking to specific board*

```
kard://boards/-Khw-Jg-491o1ouI-Qgn
```

Now, for obtaining the ability to map actual web addresses to application routes, another capability of the BaaS is leveraged. Google's standard for Digital Asset Links defines a mechanism for websites to be associated with separate applications (65). Based on this standard, Firebase is able to generate dynamic web links which will eventually invoke the applications already implemented intent handling. As a prerequisite for this mechanism, the application has to be installed on the device which is accessing the link (50). The dynamic links are following the hereinafter illustrated pattern.

*Dynamic link Pattern*

```
https://<application-code>.<application-name>.goo.gl/
?link=<deep-link>&apn=<application-package-name>
```

*Dynamic link example linking to specific board*

```
https://k1234.app.goo.gl/
?link=https://kard.de/boards/-Khw-Jg-491o1ouI-Qgn
&apn=de.mehlhorn.Kard
```

With this setup the deep linking functionality is completed. Hence, any link meeting the described form may lead to the respective application content (see also appendix C). In relation to the definitions for Digital Asset Links the platform or a browser may act as the statement consumer when the user is opening a dynamic link. Afterwards, the statement consumer requests the statement list from the BaaS which is acting as the principal according to the standard. If a valid statement matching the specifications of the link is found, a suitable intent containing an Android data URL will be passed to the application (65). The overall process is again illustrated by Figure 15.

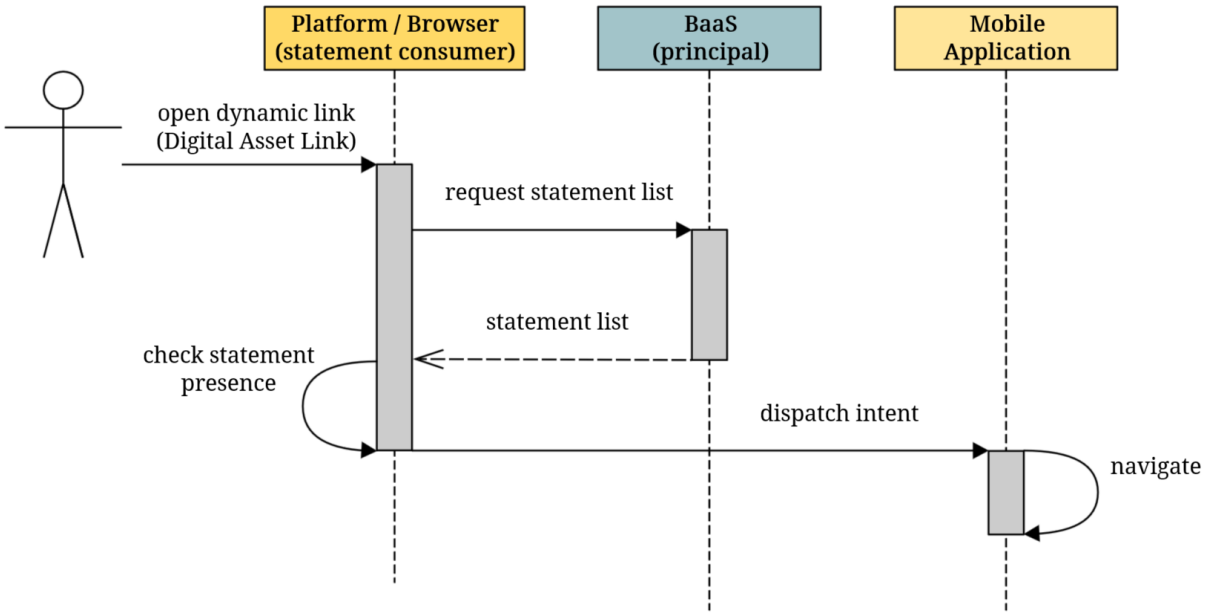


Figure 15 UML sequence diagram illustrating the deep linking mechanism based on Digital Asset Links used for the Kanban board application

### Dispatch of push notifications

Within the Kanban board application the cards are holding crucial information about tasks and their performance. Therefore it may be useful to be informed about any changes made to a specific card. A user may opt-in to notifications

about updates to a card by “watching” it. Watching a card will subscribe the user to the BaaS cloud messaging service and register him for receiving subsequent notifications. This feature also requires extension of the BaaS through custom implementations. Hereby, database triggers are setup which will be activated when any changes to data nodes associated with a card occur. In this case a custom notification is constructed which gets the appropriate data attached for being handled on the mobile client. Afterwards the notification is pushed to all devices subscribed to updates for the specific card. Upon arrival, the notification will appear in the system’s notification tray. When clicked there, the application resumes by displaying the respective card. In case the application is already running in the foreground when the notification arrives, the user will be informed about the update through an alert dialog (see Figure 16).

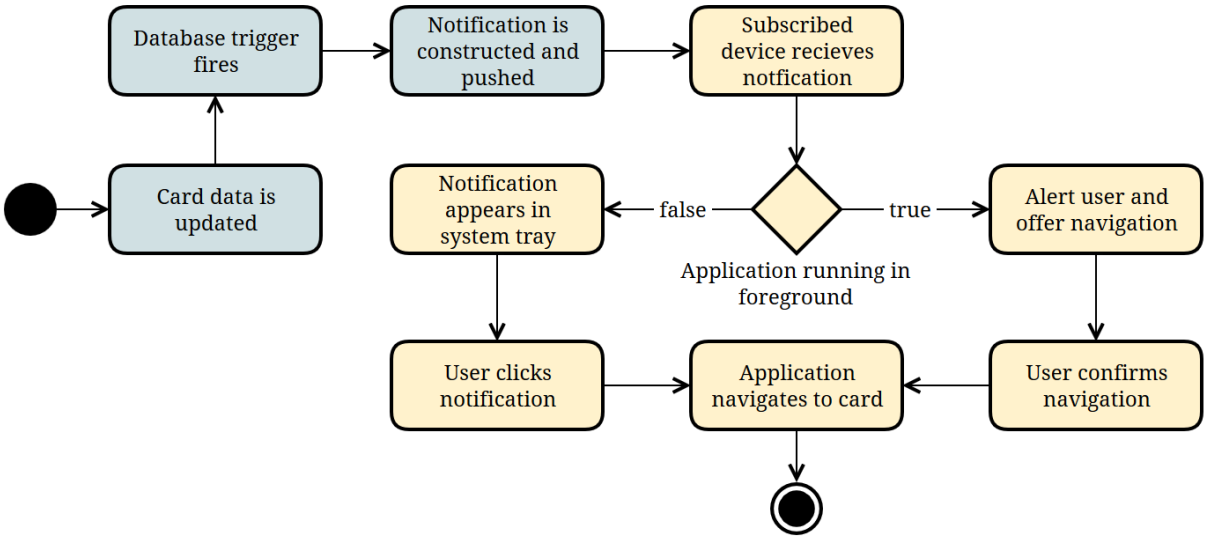


Figure 16 UML activity diagram illustrating the dispatch and handling of push notifications for watched cards

Receiving push notifications while the application is running in the background is made possible by the registration of a native Android service. This service is able to handle platform intents which are produced as a result of a push notification. Such an intent is then parsed by the service and subsequently passed to the application’s NativeScript code.

While the Kanban board application facilitates push notifications by leveraging the Firebase SDK, it is also possible to achieve interoperability with any other cloud messaging service. Additionally, NativeScript itself provides another plugin for generic push notification support, which could be the foundation of such implementations (66).

### Mechanisms for responsiveness

As previously announced, NativeScript applications are not reliant on web-based view mechanisms like the Document Object Model (DOM). Therefore appropriate platform-agnostic counterparts are provided. Different layout containers may be used to organize view elements in a grid or stack them on top of each other, for example. Even a custom implementation of the CSS flexible box layout is available (67 Predefined Layouts). As mentioned earlier, the layout containers can be defined programmatically or in a declarative way. However, in any case, the layout definitions are solely represented by JavaScript components and are therefore completely cross-platform (49 p. 46). Each of these components wraps the access to native layout elements and manages their construction by "[…] measuring and positioning the child views of a Layout container" (67 Layout Process).

NativeScript applications are not working on the DOM, but they may still be styled with an adjusted subset of CSS. The framework works with device-independent pixels (dp) which are mapped to real device pixels depending on the device's pixel density measured in dots per inch (DPI). Consequently, all instructions regarding display sizes or element positions are specified using dp or relative percentage values (68) (30 pp. 51-52).

Indicated in the explanations for displaying the launch screen (see p. 33), the platform-specific ways for responsive resource handling are useable during the application development. Native assets in form of images or view definitions may be supplied directly and thus provide remedy in cases where the framework's limits are reached. This approach is used in the prototype to, for instance, make platform-specific icons available in the application. These are referenced in the view markup simply by their identifier. While building the application for a specific target system, NativeScript will pick up the appropriate asset representation automatically. During the actual execution, the system's innate responsive resource handling will take effect. On Android, this works by deploying multiple assets in different resolutions in the installer called Android Package Kit (APK). Eventually, a particular asset copy matching the device specifications will be displayed (30 pp. 140-143).

A similar mechanism may be used to provide multiple layout markups for a single NativeScript view component. By that, different devices may be served with appropriate view definitions for their specifications (43). Unfortunately, this approach is not utilizable in combination with Angular as it conflicts with

the way how the framework's AOT works (69). Therefore this feature is not implementable for the Kanban board application.

### Mechanisms for progressiveness

Regarding the sole execution of JavaScript code, NativeScript relies on the progressiveness of the respective JavaScript engines of the underlying platforms. Any interface which is not directly related to DOM operations is useable from within NativeScript code. These, just like when they are used for running web applications, always lag behind the official standard in their available feature set. And, just like on the web, this situation is dealt with by leveraging polyfills. These are shims which implement innovative features for older API sets. Because the superset language TypeScript is used for developing the prototype, this procedure is not necessarily needed. As TypeScript code is transpiled[1] to JavaScript code, the target API specification may be declared in the same step. Therefore the generated JavaScript code will be directly compatible with a specified feature set. For the Kanban board application this target feature set resolves to the specification for ECMAScript 5 to guarantee an untroubled execution.

The targeted system API level is in turn crucial for calls to native interfaces. Due to the framework's architecture, NativeScript is able to compile an application against almost any system SDK version. For Android, the necessary metadata for executing an application is generated by iterating over all available Java types. Therefore, the most recent SDK version is seamlessly supported by the framework itself. However, obviously the application code itself also has to be compatible with the version in use (70). In terms of backwards compatibility, the NativeScript Android runtime is able to support Android APIs down to level 10. This is achieved by, among other things, leveraging the Android support libraries (71). Yet, the core cross-platform modules are commonly relying on much more advanced API levels. Due to these restrictions, the prototype can only provide backwards compatibility down to Android API level 17 (72).

[1] source-to-source compilation between programming languages

# 5 Discussion

Hereinafter the results for the assessment of NativeScript are discussed. This is done by going through each criterion of the previously laid down set of criteria and evaluating how the prototype implementations and findings regarding the corresponding functional requirements justify their degree of fulfillment. Furthermore, aspects of efficiency and usefulness of the functions and their implementations may be discussed (see also appendix A).

The prototype application is eventually functioning regardless of network state to the greatest possible extent. A reliable caching concept for every part of the application is implemented. Thus, access to application data including binary assets is ensured. Even outward communication for manipulative operations is, in theory, independent of network availability. However, the meaningfulness of performing any operation at an unspecified later point may be controversial. Here, a concrete concept for dealing with conflicting requests would be needed to allow for a proper multi-user support. Yet, the criterion of network independence may be classified as largely fulfilled as the implemented offline experience is otherwise mainly meaningful. This is further attested by the fact that the offline workflow illustrated in Figure 3 can be evidently retraced for the prototype application.

The criterion of security may also be seen as met by the findings. As any network request is ultimately using HTTPS, the server communication is consequently protected by a credible measure to prevent tampering. Yet, what might be seen as an issue is that the SDKs responsible for the cryptographic operations used in the prototype are not open source. This might be unfavorable from a security perspective. However, it is still possible to leverage HTTPS within NativeScript applications in a cross-platform way whilst relying on solid open source implementations.

With the ability to directly influence the native metadata, NativeScript offers great control over the way the application eventually is deployable. At the same time the framework takes on most of the heavy lifting by generating the most vital metadata completely by itself. This combination allows for producing application packages which are in no way inferior to natively developed ones, consequently enabling convenient publishing. Though the application‘s launch screen is not configurable via abstract view definitions, it is easily integrated through the thought out asset mechanisms and immediately picked up by the

framework. Unfortunately, despite optimization efforts, the application loading time is still lasting unpleasantly long. Yet, overall the criterion of marketability is met as production readiness is achievable without much further ado and the launch screen serves its purpose of furthering the approachability.

At first sight the implementations for fulfilling the criterion of linkability may seem cumbersome. Yet, the corresponding functionality is arguably easier put into practice with NativeScript than with native measures. The Angular framework saves a great deal of work by providing preconfigured application routing. Outside of the mobile application's scope it would be necessary to leverage concepts like Digital Asset Links either way to implement dynamic deep linking. It might not work innately like it is the case for web applications, yet the added value is undeniable. Eventually, the resulting mobile application is almost on par with web applications regarding reachability, thus constituting a satisfying fulfillment of the criterion.

The dispatch of push notifications was performable without any restrictions. Though the prototype implementations are arguably strongly coupled with the BaaS in use, NativeScript applications may be easily integrated with any cloud messaging provider. The corresponding plugins are not conformant with standards like the Push API, but are still able provide the same features. As the use of push notifications was originally restricted to native implementations, this does not come as a surprise. With NativeScript such platform features are simply wrapped into cross-platform modules and consequently offer abstracted handling. However, the prototype is ultimately able to deliver an enhanced user experience and spark anew user interaction unlike it could without push notifications. By implication, the criterion of re-engagement is attainable when building upon NativeScript.

Various mechanisms for building a responsive mobile application seem to be in place. Layout and styling may be applied independent of platform or device specifications. Regarding these aspects, responsive views may be constructed while working at an abstracted development layer. However, assets have to be handled almost entirely with native principles. Having this option may be rated as a useful characteristic in itself, yet, the absence of any platform-agnostic asset handling is definitely a shortcoming. Though, it is not quite predictable how well such an approach would perform eventually, it could probably ease development quite a bit. Furthermore, NativeScript is not able to differentiate between multiple screen sizes when used in combination with Angular. This has

to be also deemed as quite problematic as the application views may need to be broadly adjusted depending on the device's screen specifications. Consequently, the criterion of responsiveness cannot be fully attested. Several beneficial features for ensuring responsive behavior might be in place, nevertheless the mentioned issues prevent fulfilling the criterion in its entirety. Having said this, it also has to be pointed out that the tasks of providing flawless responsive capabilities while offering high nativity and abstraction is anything but simple. With this in mind, NativeScript is actually performing quite well in this area.

Regarding the progressive enhancement of JavaScript code, NativeScript applications are completely similar to PWAs as they are able to use the same approaches for providing backwards compatibility while leveraging innovative language features. In terms of compatibility with the underlying platform NativeScript is always on the edge. The most recent system interfaces are immediately available, but the backwards compatibility has its limits. And these limits might be a deal breaker for certain developments where support for older devices is an important requirement. Still, the criterion is attainable to a far extend. An abstracted development makes it easy to provide filler code and the limited API support seems to be a less technical problem. Therefore NativeScript applications are able to certify very well for progressiveness in this context.

# 6 Conclusion and outlook

PWAs are meant to close the existing gap between the web and native mobile applications. Due to the emergence of sophisticated yet standardized technologies they are able to implement characteristics which eventually allow them to deliver a significantly enhanced user experience. Consequently, PWAs may offer intriguing opportunities by elevating the web to native spheres. Several of the leveraged technologies may be still evolving and it might take some time until they are widely supported, however, with concepts like progressive enhancement this does not constitute a problem. Rather it fits in perfectly with the overall idea. The established characteristics may be implemented incrementally and subsequently improve the experience for everyone in the application's audience.

Meanwhile, NativeScript demonstrates how technologies, previously only used on the web, can be used today to develop highly native applications on a conveniently abstracted development layer. The starting point of PWAs poses interesting environment for looking at NativeScript applications. Though not all characteristics are directly applicable, they still offer a solid foundation for knowing what may be expected from a mobile application. All of these requirements could be practicably implemented or otherwise certified through the presence of certain properties, concepts or features of NativeScript to a far extent. Of course certain issues occurred during the development and shortcomings were identified, yet NativeScript may be able to provide some characteristics earlier than it would be possible on the web. At the same time, however, development can be conducted very much like for the web. Indeed, with the right architecture to ensure proper encapsulation it may even be possible to seamlessly share large parts between a PWA and a NativeScript application.

The differences between both solutions in relation to individual features are rather minor. They both seem to allow for the creation of more than decent user experiences while mitigating the need for cumbersome native developments. However, NativeScript might not be able to offer the same progressive nature as PWAs. The successive elevation of web applications presents innovative ways to think about user funnel optimization. NativeScript applications just cannot compete here as they are inevitably working in the same realms as plain native ones. On the other hand, the framework may allow the use of advanced features on systems which are not yet compatible with the most recent web standards.

Therefore, as often is the case, the individual priorities have to be evaluated when deciding between the approaches as they both seem to offer interesting opportunities but might not fit everyone's needs. It may well be that PWAs eventually gain great general attention in the future, yet, one may implement a NativeScript application today and transform it into a PWA at a later point. This may be accomplished with arguably little effort due to the technical and architectural overlaps.

Looking forward, web technologies, especially JavaScript, seem to grow increasingly powerful and extend to more and more areas of application. As shown, their usage is not restricted to simply making web content slightly more dynamic but rather be the basis for full-featured applications delivering immersive user experiences to various platforms. Furthermore, during the creation of this paper, the third major version of NativeScript was released. With improvements in performance, the elimination of several issues and new features it poses even better abilities to implement the criteria laid down in this paper (73). Hence, the prospects for the future development of web and mobile development are exciting as they seem to converge in certain aspects and conflict in others, consequently sparking progress and remarkable innovations.

In the course of this paper, PWAs and the characteristics of their development could be described in detail after the conditions for the later conducted analysis were defined. Moreover, the thematic backgrounds regarding mobile cross-platform development with NativeScript were provided in order to subsequently prepare the analysis by laying down verifiable criteria. The afterwards derived criteria was comprehensively applied through prototypic implementations and sound research to assess NativeScript and applications developed with the framework. The findings of these examinations were then summarized and weighed accordingly in a discussion. Lastly, the relevant solutions were comparatively debated against the established background resulting in concluding considerations and an appropriate outlook.

# 7 References

1. **Mehlhorn, Nils.** *Modern Cross-Platform Development for Mobile Applications.* Faculty of Media, Hochschule Düsseldorf University of Applied Sciences. Düsseldorf, 2016/2017. Scientific Consolidation. Publication revision.

2. **Google Inc.** Progressive Web Apps. *Google Developers.* [Online] [Cited: March 6, 2016.] https://developers.google.com/web/progressive-web-apps.

3. **Lync, May.** What are Progressive Web Apps? *The Official Ionic Blog.* [Online] May 18, 2016. [Cited: March 6, 2017.] http://blog.ionic.io/what-is-a-progressive-web-app/.

4. **Ater, Tal.** *Building Progressive Web Apps.* Third Early Release. Sebastopol, CA : O'Reilly Media Inc., 2016. ISBN 978-1-491-96158-2.

5. **Google Inc.** Progressive Web App Checklist. *Google Developers.* [Online] Februar 9, 2017. [Cited: March 6, 2017.] https://developers.google.com/web/progressive-web-apps/checklist.

6. **Russel, Alex, Song, Jungkee and Archibald, Jake.** Service Workers. *World Wide Web Consortium.* [Online] June 25, 2015. [Cited: March 9, 2017.] https://www.w3.org/TR/2015/WD-service-workers-20150625/.

7. **Russell, Alex.** Progressive Web Apps: Escaping Tabs Without Losing Our Soul. *Infrequently Noted.* [Online] June 15, 2015. [Cited: March 7, 2017.] https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/.

8. **Google Chrome Developers.** Opening Keynote (Progressive Web App Summit 2016). *YouTube.* [Online] June 22, 2016. [Cited: March 9, 2017.] https://www.youtube.com/watch?v=9Jef9IluQw0.

9. **comScore Inc.** *The 2016 U.S. Mobile App Report.* Virginia, 2016.

10. **Beverloo, Peter, et al.** Push API. *World Wide Web Consortium.* [Online] February 22, 2017. [Cited: March 9, 2017.] https://www.w3.org/TR/2017/WD-push-api-20170222/.

11. **Popescu, Andrei.** Geolocation API Specification 2nd Edition. *Wolrd Wide Web Consortium.* [Online] November 8, 2016. [Cited: March 9, 2017.] https://www.w3.org/TR/2016/REC-geolocation-API-20161108/.

12. **Mills, Chris.** Progressive web apps - App Center. *Mozilla Developer Network.* [Online] March 8, 2016. [Cited: March 8, 2017.] https://developer.mozilla.org/en-US/Apps/Progressive.

13. **Russell, Alex.** What, Exactly, Makes Something A Progressive Web App? *Infrequently Noted.* [Online] September 12, 2016. [Cited: March 9, 2017.] https://infrequently.org/2016/09/what-exactly-makes-something-a-progressive-web-app/.

14. **Google Inc.** Lighthouse. *Google Developers.* [Online] March 1, 2017. [Cited: March 9, 2017.] https://developers.google.com/web/tools/lighthouse/.

15. **Shepherd, Eric, Mills, Chris and sabiwara.** Working Offline - App Center. *Mozilla Developer Network.* [Online] October 26, 2016. [Cited: March 10, 2017.] https://developer.mozilla.org/en-US/Apps/Fundamentals/Offline.

16. **Basques, Kayce.** Why HTTPS Matters. *Google Developers.* [Online] February 9, 2017. [Cited: March 10, 2017.] https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https.

17. **Bahajji, Zineb Ait and Illyes, Gary.** HTTPS as a ranking signal. *Official Google Webmaster Central BLog.* [Online] August 6, 2014. [Cited: March 10, 2017.] https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html.

18. **Hume, Dean Alan.** *Progressive Web Apps.* Manning Early Access Program Version 2. Shelter Island, NY : Manning Publications Co., 2016. ISBN 978-1-617-29458-7.

19. **Caceres, Marcos, et al.** Web App Manifest. *World Wide Web Consortium.* [Online] March 2, 2017. [Cited: March 9, 2017.] https://www.w3.org/TR/2017/WD-appmanifest-20170302/.

20. **Knight, Robert, Mattisson, Jonas and Blackburn, Nathaniel.** Web App Manifest. *Mozilla Developer Network.* [Online] February 26, 2017. [Cited: March 9, 2017.] https://developer.mozilla.org/en-US/docs/Web/Manifest.

21. **Google Inc.** App Manifest. *Android Developers.* [Online] [Cited: March 9, 2017.] https://developer.android.com/guide/topics/manifest/manifest-intro.html.

22. **Gaunt, Matt and Kinlan, Paul.** Web App Install Banners. *Google Developers.* [Online] February 9, 2017. [Cited: March 14, 2016.] https://developers.google.com/web/fundamentals/engage-and-retain/app-install-banners.

23. **Osmani, Addy.** The App Shell Model. *Google Developers.* [Online] February 9, 2017. [Cited: March 17, 2017.] https://developers.google.com/web/fundamentals/architecture/app-shell.

24. **Madrigal, Alexis C.** Dark Social: We Have the Whole History of the Web Wrong. *The Atlantic.* [Online] October 12, 2012. [Cited: March 15, 2017.] https://www.theatlantic.com/technology/archive/2012/10/dark-social-we-have-the-whole-history-of-the-web-wrong/263523/.

25. **Google Inc.** Flipkart triples time-on-site with Progressive Web App. *Google Developers.* [Online] February 9, 2017. [Cited: March 15, 2017.] https://developers.google.com/web/showcase/2016/flipkart.

26. **Nagaram, Amar.** Progressive Web App: A New Way to Experience Mobile. *On The Flip Side.* [Online] November 9, 2015. [Cited: March 15, 2017.] http://tech-blog.flipkart.net/2015/11/progressive-web-app/.

27. **Ross, David, Shepherd, Eric and Mills, Chris.** Responsive design - App Center. *Mozilla Developer Network.* [Online] August 9, 2016. [Cited: March 17, 2017.] https://developer.mozilla.org/en-US/Apps/Progressive/Responsive.

28. **Archibald, Jake.** Is ServiceWorker ready? [Online] February 6, 2017. [Cited: March 17, 2017.] https://jakearchibald.github.io/isserviceworkerready/.

29. **IDC Research Inc.** IDC: Smartphone OS Market Share 2016, 2015. *International Data Corporation.* [Online] 2016. [Cited: March 29, 2017.] http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

30. **Branstein, Mike and Branstein, Nick.** *NativeScript in Action.* Manning Early Access Program Version 10. s.l. : Manning Publications Co., 2017. ISBN 978-1-617-29391-7.

31. **Progress Software Corporation.** Application Workflow. *NativeScript.* [Online] May 13, 2016. [Cited: March 30, 2017.] http://docs.nativescript.org/runtimes/android/advanced-topics/execution-flow.

32. —. What is Android Runtime for NativeScript? *NativeScript.* [Online] [Cited: November 25, 2016.] http://docs.nativescript.org/runtimes/android/overview.

33. **VanToll, TJ.** How NativeScript Works. *Telerik Developer Network.* [Online] February 16, 2015. [Cited: March 30, 2017.] http://developer.telerik.com/featured/nativescript-works.

34. **Walker, Nathan and Green, Brad.** Code Reuse in Angular 2 Native Mobile Apps with NativeScript. *Angular Blog.* [Online] March 30, 2016. [Cited: March 30, 2017.] http://angularjs.blogspot.de/2016/03/code-reuse-in-angular-2-native-mobile.html.

35. **Progress Software Corporation.** Publishing for Android. *NativeScript.* [Online] December 9, 2016. [Cited: April 4, 2017.] https://docs.nativescript.org/publishing/publishing-android-apps.

36. —. Multithreading Model. *NativeScript.* [Online] November 7, 2016. [Cited: April 4, 2017.] https://docs.nativescript.org/core-concepts/multithreading-model.

37. **Google Inc.** Preparing for Release. *Android Studio.* [Online] [Cited: April 4, 2017.] https://developer.android.com/studio/publish/preparing.html.

38. —. Launch screens. *Material design guidelines.* [Online] [Cited: April 4, 2017.] https://material.io/guidelines/patterns/launch-screens.html.

39. —. Enabling Deep Links for App Content. *Android Developers.* [Online] [Cited: April 5, 2017.] https://developer.android.com/training/app-indexing/deep-linking.html.

40. —. Making Your App Content Searchable by Google. *Android Developers.* [Online] [Cited: April 5, 2017.] https://developer.android.com/training/app-indexing/index.html.

41. —. Firebase Cloud Messaging. *Firebase.* [Online] April 3, 2017. [Cited: April 6, 2017.] https://firebase.google.com/docs/cloud-messaging/.

42. —. Notifications. *Android Developers.* [Online] [Cited: April 6, 2017.] https://developer.android.com/guide/topics/ui/notifiers/notifications.html.

43. **Stoychev, Valio.** Supporting Multiple Screen Resolutions in Your NativeScript App. *NativeScript.* [Online] May 4, 2015. [Cited: April 10, 2017.] https://www.nativescript.org/blog/supporting-multiple-screen-resolutions-in-your-nativescript-app.

44. **Google Inc.** Support Library Features Guide. *Android Developers.* [Online] [Cited: 4 25, 2017.] https://developer.android.com/topic/libraries/support-library/features.html.

45. **Anderson, David J.** *Kanban: Evolutionäres Change Management für IT-Organisationen.* [trans.] Arne Roock and Henning Wolf. 1. Heidelberg : dpunkt.verlag, 2011. ISBN 978-3-86491-027-2.

46. **Henry, Alan.** Productivity 101: How to Use Personal Kanban to Visualize Your Work. *Lifehacker.* [Online] February 25, 2015. [Cited: April 26, 2017.] http://lifehacker.com/productivity-101-how-to-use-personal-kanban-to-visuali-1687948640.

47. **Savkin, Victor.** Angular: Why TypeScript? *Angular.* [Online] July 22, 2016. [Cited: April 26, 2017.] https://vsavkin.com/writing-angular-2-in-typescript-1fa77c78d8e8.

48. **Progress Software Corporation.** TypeScript for NativeScript Developers. *NativeScript.* [Online] [Cited: April 26, 2017.] https://www.nativescript.org/using-typescript-with-nativescript-when-developing-mobile-apps.

49. **Anderson, Nathanael J.** *Getting Started with NativeScript.* Birmingham : Packt Publishing Ltd., 2016. ISBN 978-1-78588-865-6.

50. **Google Inc.** Features. *Firebase.* [Online] [Cited: April 27, 2017.] https://firebase.google.com/features/.

51. **Narayanan, Anant.** Where does Firebase fit in your app? *The Firebase Blog.* [Online] March 25, 2013. [Cited: April 27, 2017.] https://firebase.googleblog.com/2013/03/where-does-firebase-fit-in-your-app.html.

52. **Verbruggen, Eddy.** EddyVerbruggen/nativescript-plugin-firebase: NativeScript plugin for Firebase, the leading realtime JSON app platform . *GitHub.* [Online] [Cited: April 28, 2017.] https://github.com/EddyVerbruggen/nativescript-plugin-firebase.

53. **Google Inc.** Enabling Offline Capabilities on Android. *Firebase.* [Online] April 13, 2017. [Cited: April 27, 2017.] https://firebase.google.com/docs/database/android/offline-capabilities.

54. **VideoSpike LLC.** VideoSpike/nativescript-web-image-cache: An image caching library for both Android and iOS that wraps Facebook Fresco and SDWebImageCache. *GitHub.* [Online] April 4, 2017. [Cited: May 2, 2017.] https://github.com/VideoSpike/nativescript-web-image-cache.

55. **Lee, Andrew.** Firebase Will Require SSL Starting February 4th. *The Firebase Blog.* [Online] January 21, 2013. [Cited: April 27, 2017.] https://firebase.googleblog.com/2013/01/firebase-will-require-ssl-starting.html.

56. **Google Inc.** Firebase Hosting. *Firebase.* [Online] April 28, 2017. [Cited: May 3, 2017.] https://firebase.google.com/docs/hosting/.

57. **Firbase Inc.** Open Source. *Firebase.* [Online] [Cited: May 5, 2017.] https://www.firebase.com/terms/oss.html.

58. **getHuman LLC.** gethuman/nativescript-https: Secure HTTP client with SSL pinning for Nativescript - iOS/Android. *GitHub.* [Online] January 5, 2017. [Cited: May 5, 2017.] https://github.com/gethuman/nativescript-https.

59. **Progress Software Corporation.** Infrastructure. *NativeScript.* [Online] July 27, 2016. [Cited: April 4, 2017.] https://docs.nativescript.org/plugins/plugins.

60. —. Publishing for Android. *NativeScript.* [Online] December 9, 2016. [Cited: May 2, 2017.] https://docs.nativescript.org/publishing/publishing-android-apps.

61. —. Creating Launch Screen and App Icons for Android. *NativeScript.* [Online] September 22, 2016. [Cited: May 5, 2017.] https://docs.nativescript.org/publishing/creating-launch-screens-android.

62. —. Bundling Script Code with Webpack. *NativeScript.* [Online] April 11, 2017. [Cited: May 3, 2017.] http://docs.nativescript.org/angular/tooling/bundling-with-webpack.html.

63. **Google Inc.** Routing & Navigation. *Angular.* [Online] [Cited: April 28, 2017.] https://angular.io/docs/ts/latest/guide/router.html.

64. **Progress Software Corporation.** Extending Application and Activity. *NativeScript.* [Online] February 2, 2017. [Cited: May 5, 2017.] https://docs.nativescript.org/angular/runtimes/android/advanced-topics/extend-application-activity.html.

65. **Google Inc.** Google Digital Asset Links. *Google Developers.* [Online] July 14, 2016. [Cited: April 28, 2017.] https://developers.google.com/digital-asset-links/v1/getting-started.

66. **Progress Software Corporation.** NativeScript/push-plugin: Contains the source code for the Push Plugin. *GitHub.* [Online] February 3, 2017. [Cited: May 5, 2017.] https://github.com/NativeScript/push-plugin.

67. —. Layouts. *NativeScript.* [Online] March 17, 2017. [Cited: May 4, 2017.] https://docs.nativescript.org/angular/ui/layouts.html.

68. —. Styling. *NativeScript.* [Online] May 2, 2017. [Cited: May 4, 2017.] https://docs.nativescript.org/angular/ui/styling.html.

69. **Tsonev, Nikolay and Vakrilov, Alexander.** Screen Size Qualifiers not work in tns Angular project, only work in js project. *GitHub.* [Online] April 20, 2017. [Cited: May 4, 2017.] https://github.com/NativeScript/nativescript-angular/issues/404.

70. **Progress Software Corporation.** Android Runtime Requirements. *NativeScript.* [Online] August 2, 2016. [Cited: May 4, 2017.] https://docs.nativescript.org/runtimes/android/requirements.

71. —. NativeScript/android-runtime: Android runtime for NativeScript (based on V8). *GitHub.* [Online] April 24, 2017. [Cited: May 4, 2017.] https://github.com/NativeScript/android-runtime.

72. **Atanasov, Georgi.** Support lower Android API levels. *GitHub.* [Online] September 8, 2015. [Cited: May 4, 2017.] https://github.com/NativeScript/NativeScript/issues/694.

73. —. NativeScript 3.0 Available Today. *NativeScript.* [Online] May 3, 2017. [Cited: May 8, 2017.] https://www.nativescript.org/blog/nativescript-3.0-available-today.

# Appendix

## A. Breakdown of assessment results

The following table shows a breakdown of the results of the NativeScript assessment grouped by criterion. Each criterion is associated with the corresponding prototype implementations or concepts and technologies used of, or in relation with, NativeScript. Lastly, the degree of fulfillment is illustratively noted in an arguably non-uniform manner. These classifications are not claiming general informative value but are rather meant to sum up the actual informed discussion on a criterion.

| CRITERION | IMPLEMENTATIONS / CONCEPTS / TECHNOLOGIES | FULFILLMENT |
|---|---|---|
| *NETWORK INDEPENDENCE* | • Caching of read and write operations via Firebase SDK<br>• Image caching via Facebook Fresco wrapper plugin | fulfilled |
| *SECURITY* | • Encryption of all server communication with HTTPS via Firebase SDK or alternative plugin | fulfilled |
| *MARKETABILITY* | • Modification and merge of Android application manifest<br>• Provision of launch screen via native asset handling<br>• Module bundling and AOT as a startup optimization measure | fulfilled |
| *LINKABILITY* | • Digital Asset Links plus Angular application routing and native Android intent handling | fulfilled |
| *RE-ENGAGEMENT* | • Dispatch of push notifications via Firebase SDK or alternative plugin | fulfilled |

| CRITERION | IMPLEMENTATIONS / CONCEPTS / TECHNOLOGIES | FULFILLMENT |
|---|---|---|
| *RESPONSIVENESS* | • Layout via cross-platform layout containers<br>• Styling via CSS and dp<br>• Native asset handling<br>• Provision of different views via screen size qualifiers (though not useable with Angular) | partially fulfilled |
| *PROGRESSIVENESS* | • JavaScript transpilation or usage of polyfills as shims<br>• Dynamic meta data generation against almost any system API version (though compatibility issues within the framework exist) | largely fulfilled |

## B. Startup optimization data

The following table presents measured startup times for a normal build and one which is optimized with Angular's AOT capabilities through the use of the Webpack module bundler. The provided data refers to the elapsed period between clicking the application icon and the eventual display of the first view after the launch screen. The actual measurements were performed on a Motorola Moto G (2. Generation) running Android 6.0.

Unfortunately, as apparent from the data, the supposed optimization measure seems to have almost no effect or even worsened the startup times. However, these measurements are not claiming a representative nature and the results may be skewed by application specific issues. Nevertheless, the APK size fairly reduced from 24,5 MB down to 15,1 MB. Further optimization steps like lazy loading may improve the startup time more effectively and multiply with the AOT compilation in their effect.

| **LAUNCH NUMBER** | **LAUNCH TYPE** | **STARTUP TIME [s]** | |
|---|---|---|---|
| | | ***Normal build*** | ***AOT build*** |
| *1.* | cold | 8,881 | 8,850 |
| *2.* | lukewarm | 0,515 | 1,838 |
| *3.* | lukewarm | 2,334 | 1,680 |
| *4.* | cold | 9,515 | 9,139 |
| *5.* | lukewarm | 1,704 | 2,562 |
| *6.* | cold | 8,534 | 8,553 |
| *7.* | lukewarm | 0,688 | 0,528 |
| *8.* | cold | 8,390 | 8,355 |
| *9.* | lukewarm | 1,446 | 2,386 |
| *10.* | lukewarm | 0,642 | 0,810 |
| *Avg.* | cold | 8,830 | 8,724 |
| *Avg.* | lukewarm | 1,222 | 1,634 |

## C. Exemplifying code listings for deep linking support

*Route definitions*

The excerpt below illustrates how routes are defined for the Angular application by specifying valid paths and corresponding components for instantiation.

app/app.routing.ts *TypeScript*

```
import { Routes } from "@angular/router";

const routes: Routes = [
    { path: "", redirectTo: "/boards", pathMatch: "full" },
    { path: "boards", component: BoardsComponent },
    { path: "boards/:key", component: BoardDetailComponent },
    { path: "boards/:boardKey/registers/:registerKey/cards/:cardKey",
        component: CardDetailComponent }
];
...
```

*Activity extension and intent handling*

The following listing shows how the main Android activity is extended. By this, access to the native intent handling is gained and any incoming deep links can be eventually exposed as an observable stream of application routes.

app/activity.android.ts *TypeScript*

```
import { ReplaySubject } from "rxjs/ReplaySubject";
import { Observable } from "rxjs/Observable";

const _intentSubject = new ReplaySubject<android.content.Intent>();
const _routeIntentActions = [android.content.Intent.ACTION_VIEW,
                             android.content.Intent.ACTION_MAIN];
// expose routes by mapping intents appropriatly
export const AndroidOnRouteToURL: Observable<string> = _intentSubject
    .filter(i => _routeIntentActions.indexOf(i.getAction()) != -1)
    .map(i => i.getDataString())
    .filter(dataStr => dataStr && dataStr !== "null");

@JavaProxy("de.mehlhorn.Kard.MainActivity")
export class Activity extends android.app.Activity {

    protected onNewIntent(intent: android.content.Intent): void {
        super.onNewIntent(intent);
        _intentSubject.next(intent);
    }
    ...
}
```

*Route handling*

The last listing illustrates how the deep link route triggers a navigation in the Angular application. The main Angular component is eventually able to subscribe to incoming routes in a platform-agnostic way. When a route is emitted, the according subsequent navigation takes place.

app/app.component.ts *TypeScript*

```
import { Component, OnInit, NgZone } from "@angular/core";
import { Observable } from "rxjs/Observable";
import { RouterExtensions } from "nativescript-angular";
import { isAndroid, isIOS } from "platform";

// setup route source for both platforms
let OnRouteToURL: Observable<string>;
if (isAndroid) {
    OnRouteToURL = require("./activity").AndroidOnRouteToURL;
} else if (isIOS) {...}

@Component({selector: "kard-app", templateUrl: "app.component.html"})
export class AppComponent implements OnInit {

    constructor(private _zone: NgZone,
        private _routerExt: RouterExtensions) {...}

    ngOnInit() {
        // subscribe to routing events from both platforms
        OnRouteToURL.subscribe((url) => this._handleRouting(url));
    }

    private _handleRouting(url: string) {
        // in production allowed routes might be limited
        const routePrefix = "://kard.de"
        // assume everything after "://kard.de" is an app route
        const route = url.substr(url.indexOf(routePrefix) +
                        routePrefix.length);

        // do the routing in the Angular zone on next tick
        // to ensure execution in the right context and router is ready
        setTimeout(() => {
            this._zone.run(() => this._routerExt.navigateByUrl(route));
        });
    }
    ...
}
```