José Maria Campos Donato

CAN WEB APPLICATIONS WITH ALL THE RIGHT VITAMINS BE AS RELIABLE AS NATIVE APPLICATIONS?

UNIVERSIDADE D COIMBRA
1 2 9 0

1 2  9 0

UNIVERSIDADE D
COIMBRA

José Maria Campos Donato

CAN WEB APPLICATIONS WITH ALL THE RIGHT VITAMINS BE AS RELIABLE AS NATIVE APPLICATIONS?

Dissertation in the context of the Master in Informatics Security advised by Professor Nuno Antunes and Naghmeh Ivaki and presented to Faculty of Sciences and Technology / Department of Informatics Engineering.

July 2021

Faculty of Sciences and Technology
Department of Informatics Engineering

Can web applications with all the right vitamins be as reliable as native applications?

José Maria Campos Donato

Dissertation in the context of the Master in Informatics Security
advised by Prof. Dr. Nuno Antunes and Dr. Naghmeh Ivaki and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

July 2021

UNIVERSIDADE D
COIMBRA

This work is within the informatics security specialization area and was carried out in the Software and Systems Engineering (SSE) Group of the Centre for Informatics and Systems of the University of Coimbra (CISUC).

This work is partially supported by the project METRICS: *Monitoring and Measuring the Trustworthiness of Critical Cloud Systems* (`POCI-01-0145-FEDER-032504`), co-funded by the Portuguese Foundation for Science and Technology (FCT) and by the *Fundo Europeu de Desenvolvimento Regional* (FEDER) through *Portugal 2020 - Programa Operacional Competitividade e Internacionalização (POCI)*.
It is also supported by the project TalkConnect: *Voice Architecture over Distributed Network* (`POCI-01-0247-FEDER-039676`), co-financed by the ERDF, through Portugal 2020 (PT2020), and by COMPETE2020 and FCT.

This work is being supervised by Professor Nuno Manuel dos Santos Antunes, Assistant Professor at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra and by Doctor Naghmeh Ramezani Ivaki, Researcher at the Centre for Informatics and Systems (CISUC) of the Faculty of Sciences and Technology of the University of Coimbra.

To my mother and my grandpa, everything I do is to make you both proud.

Acknowledgements

I would like to start by thanking Professor Nuno Antunes first for all the professional knowledge, but most importantly for all the personal wisdom. Thank you for the life lessons that you taught me. Dr. Naghmeh Ivaki, thank you for the guidance and uninterrupted motivation.

To my family, Paulo and Severine, Tiz and Ale, João e Sara for keeping me sane during this lonely year. And, of course, to our two beloved dogs that accompanied us over the last years.

To my Aunt Nani, my Grandma Bety, and my cousin Francisco, who are always available for me no matter the circumstances.

To my closest friends, Maria Inês, Barros, Brink, Martinho and Guilhon, for the good laughs and the few but great get-togethers.

Finally, to my kind and beautiful girlfriend and her amazing and caring family. Thank you, Rita, for all the support, nothing would be possible without you. I am sure I could not pass through chapter 1 without your continuous support and love.

This page is intentionally left blank.

Abstract

During the last decade, the gap between native, hybrid, and web applications has been reducing. Push notifications, offline fallback, and other features enabled native-like applications that work directly on the browser. Since they can be accessed from the browser, web applications are not limited to a certain platform, which benefits both end-users and developers. Different types of development tools to produce applications are constantly and rapidly emerging. When engineers need to develop a native or web application, they are overwhelmed by the huge diversity of alternatives and lack the means to choose the solution that best fits their needs.

In this work, we propose a novel framework to assess different development tools according to certain properties such as performance, reliability, and dependability. As it is not feasible to compare the development tools directly, we propose comparing them through representative applications. The framework defines the components and procedures required to define concrete benchmarks.

To demonstrate the applicability of the proposed framework, it was instantiated in a concrete benchmark focused on the performance of entertainment and utility applications developed with JavaScript tools and Kotlin as a native reference. For this, we defined a representative set of features that each application must implement based on an analysis of popular apps. The relevant metrics to characterize performance were identified. For each tool, an application was developed with the defined set of features. A benchmarking campaign was executed, with the help of a supporting tool that automates the functional tests and collects the metrics. The campaign results were analyzed to compare the applications and the development tools that produced them.

The results show that the framework can be used to assess and compare the development tools. We observed that even though Ionic uses more Memory and CPU than the native applications, it was the fastest to complete the tests. The observed differences between Expo and React Native are not significant, meaning that Expo is able to ease development and extend the cross-platform development without compromising performance. Finally, the results confirmed that web applications are already a competitive alternative in most mobile application scenarios.

Keywords

JavaScript Frameworks, Benchmarking, Android, Mobile applications, Native applications, Web applications

This page is intentionally left blank.

Resumo

Ao longo da última década, as diferenças entre as aplicações nativas, híbridas e web têm vindo a diminuir. Notificações, usabilidade mesmo sem conexão à internet, e outras funcionalidades permitiram a existência de aplicações que funcionam diretamente através do browser como se de nativas se tratasse. Dado que são acedidas pelo browser, estas não ficam limitadas a uma certa plataforma, o que constitui uma vantagem tanto para os utilizadores como para programadores. Para além disso, as ferramentas para desenvolver aplicações web e nativas têm vindo a aumentar exponencialmente e são hoje muito diversas. Consequentemente, os desenvolvedores sentem-se sufocados com tal diversidade e não têm meios para avaliar qual a melhorar solução para as suas necessidades.

Neste trabalho é proposta uma framework para avaliar e comparar diferentes ferramentas de desenvolvimento, de acordo com certas propriedades, tais como performance, e confiabilidade. Dado que não é viável comparar as ferramentas diretamente, propomos compará-las através de aplicações representativas. Esta framework define um conjunto de components e procedimentos necessários para definir benchmarks concretos.

Para demonstrar a aplicabilidade da framework, ela foi instanciada num benchmark concreto focado na performance de aplicações de utilidade e entertenimento desenvolvidas com ferramentas JavaScript e Kotlin como referência nativa. Definimos um conjunto representativo de funcionalidades que cada aplicação deve implementar com base na análise de aplicações populares. Identificámos métricas relevantes para definir a performance. Para cada ferramenta, uma aplicação foi desenvolvida com o conjunto de funcionalidades. Uma campanha de benchmark foi executada com o suporte de uma ferramenta para automatizar testes funcionais e recolher as métricas. Os resultados desta campanha foram analisados para comparar as aplicações e as ferramentas que as produziram.

Os resultados mostram que a framework pode de facto ser utilizada para avaliar e comparar diferentes ferramentas de desenvolvimento. Observámos que Ionic apesar de utilizar mais recursos que as aplicações nativas, foi a mais rápida a completar os testes. As diferenças observadas entre Expo e React Native não são significantes o que demonstra que o Expo pode facilitar o desenvolvimento e suportar ainda mais plataformas sem comprometer a performance. Finalmente, os resultados confirmam que as aplicações web são já uma alternativa competitiva na maior parte dos cenários.

Palavras-Chave

Frameworks de JavaScript, Benchmarking, Android, Aplicações móveis, Aplicações nativas, Applicações web

This page is intentionally left blank.

Contents

List of Figures

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

During the last decades, smartphones have improved, and their usage came along. In 2019, there was an estimate of 3.5 billion smartphone users [1], which translates to around 45% of the world's total population, with even higher percentages in some countries (e.g., 60% in China or 79% in USA [2]). This massive adoption leads to continuous improvement of mobile devices (e.g., regarding their computation power) and, consequently, increased the requirements for mobile applications. As the need for this type of application and its complexity are increasing, a grown number of development tools were created and are still being created to support developers in building new, high quality, and fast mobile applications. Multiple development tools surged and still keep surging because each author came forward to solve the same problem (i.e., how to produce applications for mobile devices or other devices) with different solutions.

Three main types of applications can be developed for mobile devices:

- **Native applications**: run natively after being installed (e.g., through application stores). They are typically the primary choice for engineers because they are fast and work offline. However, when supporting multiple platforms is vital, development and maintenance become time-consuming as they are tied to the target platform [3].
- **Web applications**: run directly on a browser, and thus, they can run on all devices using the same codebase. However, they are not always able to achieve the same performance as native applications or provide offline functionalities, but these problems are reducing and Progressive Web Applications (PWAs) try to minimize it [4].
- **Hybrid applications**: run natively, but they are usually built using web technologies and packed into native containers. They enable the development of native applications for multiple platforms, sometimes at the cost of performance [5].

An example of the increasing adoption of the web applications where native ones usually dominate is the last announcement from Microsoft. In 24th June of this year, Microsoft officially announced that Windows 11 Store will include "Win32, UWP, PWA, and now, Android apps" [6]. This means that Progressive Web Applications, a specific type of web application, will now be available to install from the same starting point as other more common native applications in the Windows environment. In the mobile environment, a

similar situation happened in 2020 where "Microsoft Worked with Google to Bring PWAs to the Play Store" [7]. Although the Windows environment is still different from mobile environments (e.g., Android or iOS), we are observing a huge adoption of web applications that may replace native ones in various situations.

When developing an application for mobile devices, developers need to choose an appropriate approach and select an adequate development tool (also called framework) to implement it. The challenge is that there are countless options (e.g., React Native or Kotlin for native, React.js or Svelte for web). In the JavaScript ecosystem, this number is constantly increasing, causing "JavaScript Fatigue" [8]. Although it pushes technologies forward, developers are overwhelmed with options and do not always know how to choose the tool for their use cases. Thus, **there is a clear need for techniques and tools that allow assessing different properties of mobile applications' development tools**, both the ones currently available and the ones that are yet to appear, to help developers decide which are the best solutions based on their requirements.

Several works in the literature evaluated both web and/or native applications (detailed in Section 2.5). However, all of them had at least one of the following limitations: i) a lack of representative features (i.e., the implemented features by the applications under test were few and/or were not features that are expected in mobile applications); ii) a lack of representative metrics (i.e., the metrics gathered had no impact on the end-user's day-to-day usage); iii) only compared few development tools from the same environment (e.g., they did not compare native applications against web applications).

This study proposes a **new framework for assessing different mobile application development tools** from different environments considering several properties, such as performance, reliability, and security. As it is not feasible to compare the development tools directly, we propose implementing applications with a representative set of features. These applications are then executed and submitted to an automated testing procedure during which metrics are gathered to be used for the comparison of the applications, consequently allowing to indirectly compare the development tools (or frameworks).

The framework defines the required components and procedures that should be followed for the development of fair and useful benchmarks. The user must define the target of the benchmark in terms of the application domain of interest, types of development tools to be adopted and quality attributes of interest. This target is decisive for specify the **feature selection criteria**, the **development tool selection criteria** and the **metric selection criteria**.

Based on these criteria, the user must identify a representative list of features, the development tools and the metrics. The applications are implemented with the selected development tools and must support the representative list of features. These applications are then submitted to an automated testing procedure during which the selected metrics are gathered to be used for the comparison of the applications, and that also allow to compare the development tools indirectly.

To demonstrate the proposed framework, we designed a concrete benchmark to evaluate and compare multiple popular development tools, used to produce applications for mobile devices, in terms of performance. For this, we defined a set of features (i.e., our reference app specification) that users expect to find in a mobile application based on the results of

the analysis of the most popular applications from Android Store. The following features are examples that the applications we developed support: i) access to the camera, ii) geolocation, iii) fetching items from a database to display, iv) infinite carousel, v) operations around a table, vi) fetching from the cache (more on the reference app specification/features on Section 4.1.1). Then, we selected a set of popular and recent development tools that are widely used, including: i) React.js, ii) Svelte, iii) Next.js, iv) Gatsby, v) preact, vi) Ionic, vii) Expo, viii) React Native, and ix) Kotlin (see Section 4.1.2). Each tool was adopted to develop one application with the reference set of features, resulting in a set of similar applications in terms of functionality, although developed with different tools. We selected several key performance metrics, including RAM consumption, response times, CPU usage, and application size, to be measured before/during the tests (see Section 4.1.3).

All the applications *must* pass through a set of functionality tests specific to the implemented features and automated with the help of Appium framework (see Section 4.2.2). All the applications were developed in-house, but the benchmark is designed to welcome and incorporate future improvements. This is an open-source project that will be open to the contribution of the community, inspired in Techempower Benchmarks [9]. Following the preparation phase, the applications enter the benchmark campaign, where they are executed over an Android device, and submitted to different workloads while gathering the measurements.

The experimental results **show that our framework can be used effectively to compare different development tools**. Between the web applications, tools such as preact and Next.js that are built with performance as the main requirement show promising results. We also observed that, in general, the performance of the development tools differs when different features of applications are executed (e.g., Kotlin uses more CPU and memory when interacting with native features such as camera, geolocation or the file system). This helps developers to choose appropriate development tool depending on the use case to implement. Another interesting observation is that some development tools resulted into development of fast applications, but instead consume more memory and CPU (e.g., implementation with Ionic presented the fastest response times but consumed more memory and CPU than native applications). This is particularly relevant in the environment (i.e., mobile devices), in which reduction of energy consumption is of high importance. In situations where there are few resources, Expo, Kotlin, or React Native should be used as they are more efficient than the competitors.

The results also showed a difference observed between Expo and React Native that is not significant, which means that Expo can be a viable alternative to produce React Native applications without compromising their performance. Regarding web applications, they are already a viable alternative to the other types of applications, providing faster response times than the native applications at the cost of more memory and CPU usage. Finally, in between the web applications, React-based tools are faster and more efficient than Svelte and between the React-based, Next.js and preact presented the best results in both resources efficiency and response times, but the differences are minimal.

As future work, we aim to extend the framework to support more development tools, perform related experiments and keep publishing the results. Moreover, although in our experiment we focused on performance, we believe that our solution can be extended to other properties such as security, thus, we aim to define appropriate measures and

necessary workload (or attack load) to explore several properties. We also believe that the framework could be extended in the future to target other properties more related to the developer experience when using such development tools. During our experiment, in addition to performance we also used some metrics to compares the tools in terms developer experience such as lines of code, dependencies number, etc. (more details on Section 4.1.3). Moreover, in the future, the framework could be extended to compare the learning curve of each development tool, or the time to develop a certain application using them.

1.1 Contributions

The main contributions of this work can be summarized as follows:

- **A framework to evaluate and compare mobile applications' development tools called Savery**. This contribution addresses the main issue stated before: engineers feel overwhelmed when they need to choose a development tool for their use cases. Savery details steps to define benchmarks that have specific targets in terms of applications and quality attributes. It considers quality attributes such as performance, reliability and security, and is prepared to be extensible and open to the community (detailed in Chapter 3).
- **A benchmark to evaluate and compare development tools that can produce applications for mobile devices, in terms of performance.** The framework proposed can only be a good contribution if it is useful for the community. Hence, another objective was to prove that the framework can indeed be used to compare different development tools. To achieve this, we instantiated the framework and conducted a performance benchmark that compared applications with the same set of representative features implemented using seven different development tools. This includes several supporting tools developed (detailed in Chapter 4).
- **A benchmark campaign that compared some of the most popular development tools commonly used to produce applications for mobile devices** (discussed and presented in Chapter 5). Our end goal with this study was to compare popular development tools capable of producing applications for mobile devices. With the results of the benchmark we were able to compare the different applications (and, subsequently, the development tools that produced them) in terms of performance. We were also able to conclude whether web applications are already an alternative to the native applications and in which situations (detailed in Chapter 5).
- **Implementation of different tools to ease usage of the assessment framework**. The main challenges of building such benchmarking environment is to keep its API simple (to be easily used by users to evaluate their applications and also to easily add new development tools for evaluation) but at the same time sufficiently complete (to ensure an effective evaluation and comparison). We developed tools to support the benchmark (detailed in Section 4.2.3).

An additional output of this study is the source code of the applications implemented with nine different popular development tools. To conduct the benchmark, we had to

implement the same application with the same set of features using different development tools. Therefore, we also provide the source code for these implementations using nine different tools. These implementations can serve as starting point for engineers that may want to start using a certain development tool supported by our benchmark.

All the outputs of this study are open to the community in the following website and repository:

- `https://savery.dei.uc.pt`
- `https://github.com/jose-donato/savery`

1.2 Dissertation Structure

The remainder of this document is organized as follows:

Chapter 2 provides an overview of web and native applications and a summary about benchmarking concepts (mainly regarding software systems). Alongside these explanations, this chapter also presents the related work.

Chapter 3 presents the main contribution of this work: the framework for the assessment of mobile development tools called Savery. This chapter is divided into four different sections, matching the four phases of the framework.

Chapter 4 focuses on the performance benchmark conducted to instantiate the proposed framework. It goes through the different phases and outlines how we implemented the processes from the framework in detail.

Chapter 5 presents and discusses the results of the benchmark conducted. We break this chapter into different sections to analyze the applications considering certain aspects (e.g., CPU usage, memory consumption). The threats to the validity of the presented results are presented in the last section of this chapter.

Chapter 6 concludes this work and outlines the main lessons learned throughout the study as well as the possible future work that can be done.

This page is intentionally left blank.

Chapter 2

Background and Related Work

This chapter introduces some important concepts regarding web applications, mobile applications and the main differences between them, which are required to understand the following chapters. The need for applications for mobile devices is increasing exponentially. This has led to the appearance of many development tools to support developers in creating new, high-quality, and fast applications. In general, these tools are used to create two groups of applications: web applications (see Section 2.1) and native applications (see Section 2.2). Their history has some similarities that will be explained in the following sections.

After explaining these two types of applications for mobile devices, a brief overview of benchmarking of software systems is provided in Section 2.4. Finally, studies and publications related to the assessment of these applications are presented in Section 2.5.

2.1 Web Applications

Starting with web applications: in 1989, Tim Berners-Lee proposed the World Wide Web (WWW) that has evolved exponentially ever since. They started as completely static websites where each interaction with the website would result in a request to the web server [10], and in the current time, we have web applications that pose a serious alternative to native ones. The proof of this is that, as said before, both Microsoft and Google are setting web applications, namely Progressive Web Applications, at the same starting point as native ones in their native stores (i.e., Microsoft Store and Google Play Store) [6, 7].

At the current time, developers can choose between two different architectures when developing web applications [11]:

- **Traditional Web Applications** (also called Multi-Page Applications or Web 2.0): HTML pages are pre-rendered on the server and are highly coupled with a remote web server. Upon request, the server usually returns a page [12]. In this type of application, each transition between different pages triggers a browser reload and a request to the server. Examples of development tools that implement such applications are Ruby on Rails or Django.

- **Single Page Applications** (SPAs): applications that are able to rewrite the page contents. This type of application enables shifting the computation load from the server-side to the client-side devices due to the continuous increase of application requirements. They are designed to be highly decoupled from the server. Each transition does not trigger a browser reload and does not make an additional request to the server. Examples of tools that produce Single Page Applications are React.js, Angular, or Vue.js [13].

To better understand how these different approaches appeared, we first need to define what the **Document Object Model** (DOM) is. According to Mozilla documentation [14], the DOM is the "data representation of the objects that comprise the structure and content of a document on the web". In simple words, it is a tree that contains all the elements that are displayed on the devices' screen. Also, a good analogy from [15] explains that "HTML is a text, the DOM is an in-memory representation of this text".

At first, web applications had no complex requirements (prior Web 2.0). The major requirement (and normally simple to achieve) was only to display content on the devices' screen. Since there was no interactivity, there was no need to manipulate the DOM. At this time, the servers were doing all the work, and the end devices were only responsible for displaying the pages and files returned from the servers.

As user requirements increased, applications became more complex and interactive. Between 1999 and 2002, asynchronous JavaScript and XML (i.e., AJAX) appeared and Rich Internet Applications (i.e., RIA) were born [10]. The web applications shifted to perform requests after getting the server's response, which allowed end-users to submit forms and add comments to give some examples [12]. This added more complexity to web applications, but still far from what is possible in the current days.

When some interactivity was added, JS helpers libraries such as jQuery abstracted interacting with the DOM, and web applications became more responsive.

Web application requirements have increased exponentially during the following years, as well as the processing power of client-side devices. Thus, **Single Page Applications (SPAs)** are born in 2010 to meet these requirements. The first development tools that could implement SPAs were Backbone.js and Angular.js. This was only possible because of the constant improvement of end devices (i.e., more precisely, mobile devices), allowing the shift of the computation from the servers to the end devices.

Instead of performing requests to the server on every interaction, a Single Page Application will receive a considerable JavaScript bundle from the server on the first request. This bundle is then responsible for displaying the page's content on the devices' screen and switching it as soon as the users navigate through the application. This resulted in faster and more responsive applications as it avoids extra trips to the server to grab content for subsequent pages (that may not differ a lot from the previous ones) [13].

Single Page Applications kept evolving during the last decade and reached a point where their differences with native applications can be minimal. This can easily be observed when comparing TikTok website[1] with its native applications[2].

[1] tiktok website: https://tiktok.com

[2] tiktok android application: https://play.google.com/store/apps/details?id=com.zhiliaoapp.musically

Fig. 2.1 displays the differences between Traditional Web Applications and Single Page Applications.

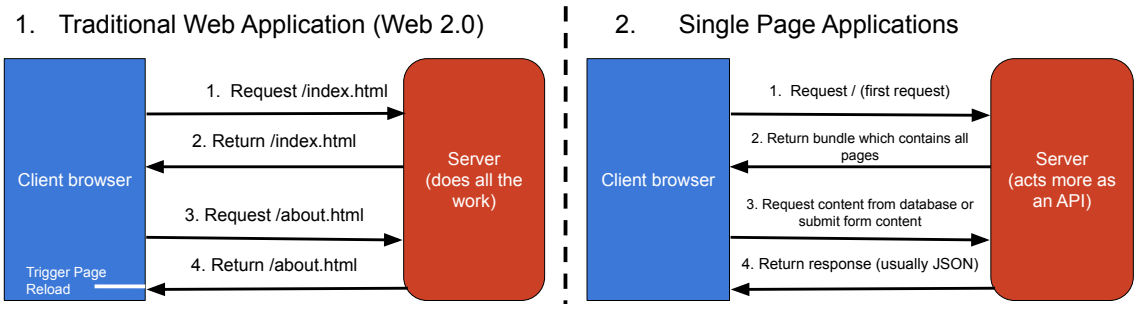


Figure 2.1: Comparison between web applications and single page applications

In Traditional Web Applications, with each request, the user receives a different HTML page (e.g., when accessing `http://example.com/index.html` index.html page is returned; when accessing `https://example.com/about.html` about.html page is returned) [10] and every time a new page is received, the browser page reloads. In this scenario, the server does all the work. Some pages may require content from the database or other sources, and the server will grab that content and insert it in the HTML before sending it to the client (of course, the client waits all this time with a blank page in the browser).

Contrary to this approach, in a SPA, with the first request to the server, a bundle is returned, containing all the application pages. When navigating through the application, instead of going back to the server for each page, the bundle is responsible for manipulating the DOM to show the desired content. Navigating through pages does not trigger a page reload when using a SPA. In the cases where the application needs a database or other server-side content, the client can send subsequent HTTP requests, but the user is never presented with a blank page because this content is normally only a fraction of the whole page (i.e., the developers of Single Page Applications can display loading spinners, skeletons, or other UI techniques while fetching content from the server never leaving the client with a blank page).

It is essential to understand that this is only possible because of the computing power of mobile devices and the existence of technologies like the DOM. With JavaScript, development tools can manipulate the DOM to show the contents they desire. Although "manipulating the DOM is the heart of modern and interactive web", this operation is expensive and can be slow [16]. SPA development tools, i.e., tools that are able to produce Single Page Applications, differ on *how* they address this problem (i.e., interaction and manipulation of DOM). The most common ways to solve this problem are listed below:

- **Virtual DOM**: **React.js** created virtual DOM (and later used by Vue.js as well). In this approach, there is a corresponding virtual node for every DOM node, which is a lightweight copy of the real one. The virtual nodes have the exact same properties but no power to change what is on the screen (which is what slows real DOM). Manipulating the virtual DOM is much faster than the real DOM because nothing gets drawn to the screen. After a certain change occurs, a process called _diffing happens. The new virtual DOM is compared to the previous one, and the changed

objects are marked. Then, React updates only these changed objects in the real DOM (only at this point, the end-user can visually see the changes on the screen). However, this process can be memory-heavy in some situations since React needs to store all the different virtual DOM [16].

- **Real DOM**: **Svelte** (a framework that is starting to gain some popularity) claims that the overhead in Virtual DOM comes from diffing and that users should not need to wait for comparison process (between virtual DOM) to see the changes on the screen. Therefore, they have solved the DOM manipulation problem by interacting directly with the real DOM (without any virtual DOM) [17].
- **Incremental DOM**: Instead of directly interacting with the real DOM, **Angular** uses something they called incremental DOM. In this approach, only one virtual DOM exists, and the tool transverses the DOM tree to look for changes (if so, update them to the actual DOM). In the virtual DOM, memory is allocated to the whole tree, while in the incremental DOM, memory is only allocated when DOM nodes change. It reduces the bundle size and memory footprint seen in development tools that use multiple virtual DOM [18].

Initially, Search Engine Optimization (SEO) was a problem in Single Page Applications. At that time, they were only rendered on the client-side (i.e., the bundle was returned to the client and rendered only when it reached the users' devices) instead of being rendered on the server-side (e.g., Ruby on Rails and Django), where pages are pre-rendered before leaving the server. This resulted in bad SEO results, i.e., it was extremely hard for crawlers (e.g., Google) to grab information about the applications and, therefore, for them to appear in Google's first results [19]. Also, rendering only the client-side can impact the performance in bad performance when the bundle sent to the client is huge because the browser needs to parse it. Later, Single Page Applications development tools were able to optimize their SEO results and, other meta-frameworks appeared (i.e., frameworks on top of other frameworks were created). This was the case of **Next.js** which is used to build server-side React.js applications [20] or Nuxt.js to build Vue applications. The appearance of these development tools allowed to split the bundle returned to the client and render some parts of the pages on the server before sending it to the client.

Single Page Applications allowed the web to become more interactive and, therefore, a better experience for end-users [21].

The key difference is that sometimes web applications are not able to achieve the same performance as mobile applications or provide offline functionalities, although these problems are reducing and Progressive Web Applications try to minimize it [4]. Web applications that could behave as native applications were first introduced by Steve Jobs in 2007 [22]. However, this was quickly forgotten by the Apple team. Later, in 2015, Alex Russell from Google brought back this concept and named it **Progressive Web Applications (PWA)** [23].

Some web development tools may facilitate achieving this, but any web application type mentioned before can be a PWA as long as it fulfills the following three requirements [24]:

- **Service Worker**: the brain behind Progressive Web Applications.

A service worker acts similar to a proxy (i.e., a benign man in the middle) that can intercept and cache requests. This functionality is displayed in the diagram 2.2 and can be useful to provide offline functionalities or intercepting requests when the connection is slow to later retry when it becomes stable, using devices' cache. Service Workers are event-driven and run on a separate JavaScript thread in the background which makes push notifications also possible [25]. A brief explanation on how a service worker may fit in a web application is provided in Fig. 2.2.

- **Web Manifest**: normally a JavaScript Object Notation (or JSON) file that contains several key value pairs to describe important application details (e.g., how it looks when installed) such as the author, version, icons, description, etc.
- **HTTPS**: the web application must be served over SSL, i.e., an encrypted connection (unless it is localhost - exception when developing the application).

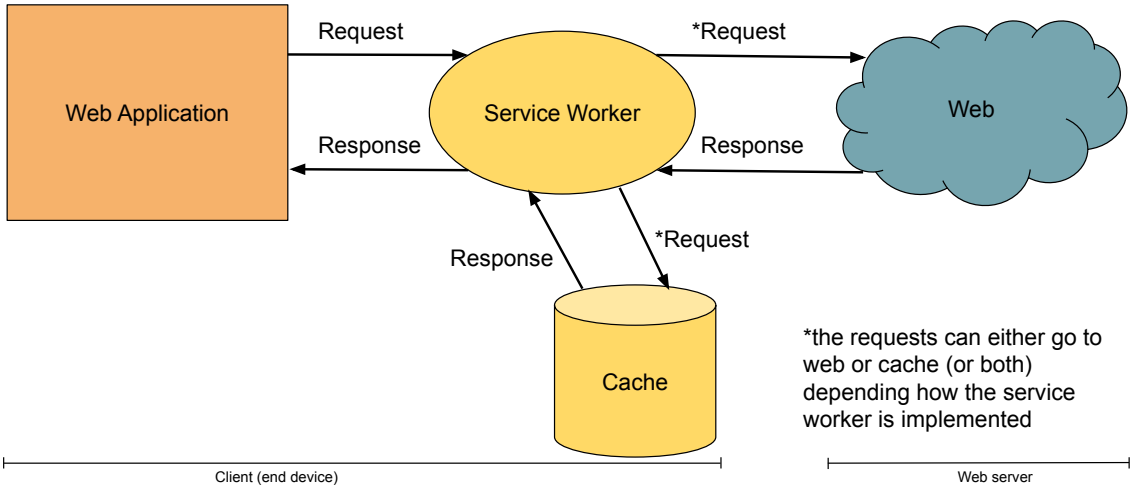


Figure 2.2: How Service Workers fit in a Web application

The shell of a web application is the part of the application that will never change. This part is the perfect example of assets that can be cached by *service workers*, which allow them to be rendered almost instantly, avoiding extra trips to the server [26].

Another important feature of PWAs is that they can be installed in any device (including mobile devices). When a user installs a PWA, it can be directly access from the home screen and the browser UI (i.e., URL bar, favorites, etc.) can optionally be opt-out - only the application will appear in the screen making a web application a much more native-like experience.

Even though it is a recent technology, companies are quickly adopting it, and many Progressive Web Applications are already being used in production [27].

2.2 Native Applications

Native applications differ from web applications in the sense that they are typically tied to a specific platform (e.g., Android or iOS), whereas web applications can run on any

platform with a web browser.

Native applications run natively after being installed (e.g., through application stores). They are typically the primary choice for engineers to build an application for mobile devices, as they are fast and work offline. However, when the ability to support multiple platforms is vital, development and maintenance become time-consuming and expensive as they are tied to the target platform[3] (e.g., Swift to produce an application for iOS devices, Kotlin to produce the same application for Android devices).

This leads up to the appearance of cross-platform tools (or CPTs), i.e., development tools that allow to build native applications for multiple platforms from a single codebase (i.e., from one source code), sometimes at the cost of performance [3]. Hybrid tools are also a subset of CPT, and they wrap web applications into native views (e.g., Ionic).

Hence, the tools to develop native applications can also be separated into different categories:

- **Native Tools**: to develop mobile applications for particular platforms (e.g., Android or iOS). The languages used are usually only supported by those particular platforms (e.g., Kotlin for Android or Swift for iOS). Users/companies normally face a major challenge: to reach all end-users, several applications should be implemented (one per platform), which is expensive and time-consuming [28].
- **Cross-Platform Tools**: include the same codebase that produce applications for multiple platforms [3].
  - **Hybrid tools**: normally websites packed into containers (i.e. web views) that emulate software behavior for every platform (e.g., Ionic) [29]. Allow developers with a web background to build native applications for multiple platforms (e.g., Android and iOS).
  - **Native-based tools**: output native applications but come from one single codebase (e.g., Flutter, React Native) [30]. These type of tools try to achieve the performance of fully native tools while not being tied to a particular platform.

As we have seen so far, both web applications and native applications suffered considerable changes in the last decades. However, web applications sometimes are not able to achieve the same performance as native applications or provide great offline functionalities. Although these problems are reducing and Progressive Web Applications try to minimize it [4]. We will now see some of the most popular development tools to develop these types of applications.

2.3 Development Tools

A brief overview of some technologies that exist to develop web applications, native applications, and PWAs was introduced in 2016 [5]. However, it is already outdated. The following sections will enumerate development tools capable of producing the different types of applications for mobile devices.

2.3.1 Web development tools

Tools to produce Traditional Web Applications

Django is a development tool to produce Traditional Web Applications. It includes an ORM for database transactions and was one of the most popular tools when Traditional Web Applications used to dominate. However, it is currently losing market share since Single Page Applications are replacing Traditional Web Applications. **Ruby on Rails** or **Laravel** are other examples of popular tools to produce Traditional Web Applications.

Tools to produce Single Page Applications

Angular was one of the first frameworks that appeared to develop Single Page Applications [31]. It is slowly becoming outdated and companies are ditching it for React.js, Vue or Svelte.

React.js is currently the most popular development tool to produce SPAs. Instead of interacting with the Real DOM, React uses an abstraction called virtual DOM [32].

Vue.js author tried to grab the best parts from Angular and build a development tool capable of producing fast applications without Angular overhead [33]. It also uses an abstraction to interact with the DOM.

Svelte is another tool to build web applications. This tool interacts with the real DOM and claims that abstractions like the ones used by React.js or Vue add overhead [17]. "Svelte is a compiler that knows at build time how things could change in your app, rather than waiting to do the work at run time" [34]. Since Svelte compiles everything at build time, it does not need to bundle the whole library code to make computations at runtime as React and many other development tools do. Svelte is starting to be adopted by the JavaScript community, and it is already being used by big companies like Microsoft, Amazon, NY Times.

Preact is a minimalistic version of React with only the needed features to produce fast and simple web applications. It is starting to be used by huge companies to build small applications. It is basically React.js with lower overhead [35].

Gatsby is a meta-framework, i.e., a framework built on top of other framework. It is built on top of React and is usually used to build static websites by major companies. However, it is currently losing market share to Next.js [36].

Next.js is another meta-framework. Claims to be a scalable, performant, and production-ready library built on top of React [37]. It is used to build both static and dynamic web applications. It is backed by a company called Vercel and is already being extensively used by popular companies (e.g., Apple, Google), and expanding exponentially.

Blitz.js is a recent framework built on top of Next.js that aims to build full stack web applications [38]. It is gaining the attention of the developers that were used to build Traditional Web Applications since it is inspired by the same principles of Ruby on Rails and Django (i.e., tools to build Traditional Web Applications). **RedwoodJS** is another JavaScript library built on top of React.js that tries to achieve the same as Blitz.js [38].

SvelteKit is a framework that tries to facilitate the creation of Svelte applications ready for production [39]. SvelteKit is similar to Next.js, but for Svelte.

Nuxt.js is a framework to build Vue applications. Next.js was inspired in Nuxt.js [40].

In 28th June of this year, **SolidJS** officially launched their first stable version. It is another development tool to build web applications. While it has a syntax similar to React.js, the authors of this tool decided to interact with the Real DOM instead of abstracting it [41]. SolidJS follows a similar approach to svelte in that it also works as a compiler.

We are already observing the birth of different development tools that aim to be the successor of Single Page Applications. An example of such is **Marko.js**. Marko.js tries to reimagine web applications and instead of sending a huge bundle that the client needs to parse, Marko streams the contents of the application based as soon as they are ready [42]. With this approach, the users do not need to wait until the browser parses the bundle and can see content almost instantly. **Astro** is another library that follows a similar approach and tries to serve the applications with as minimal JavaScript as possible [43].

As we can observe, the web is a constant fast-growing environment, with new development tools launching and current ones releasing new versions every month. We will now switch to the native environment, and we will see that although the pace is slightly slower, it is still a considerable fast-changing environment.

2.3.2 Native development tools

Java (Android) used to be the main choice when developing fully native Android applications. It was vastly used because of its performance and being the official and primary way of developing Android applications.

Kotlin is replacing the previous tool. Kotlin is currently used by 60% of professional Android developers[44]. Can produce fast and efficient native applications. Pinterest, Trello, or Evernote are examples of Android applications made with Kotlin.

Tools to produce Cross Platform Applications

Ionic is a development tool to produce hybrid cross-platform applications [45]. It uses web view native containers to render web applications in multiple platforms using web technologies (e.g., with React). Capacitor plugins can then be used to platform-specific native APIs from the web view environment. It is extremely popular because it allows building applications that run natively on several platforms using web technologies known by lot of developers (e.g., React.js).

React Native is a cross-platform technology to build native applications for both iOS and Android [30]. React Native has a syntax similar to React.js, which enables developers with a web background to build native applications and is becoming the reference to build cross-platform native applications. Examples of applications built with this technology are Airbnb, Coinbase, and Twitter.

Expo is a framework built on top of React Native to build applications for several platforms

from the same codebase [46]. Expo introduces one more level of abstraction and with the help of react-native-web [3] can produce output to the web in addition to iOS and Android platforms.

Flutter is a popular development tool created by Google to produce native cross-platform applications from the same codebase [47]. Uses its own components and render mechanism, allowing for better performance than other cross-platform frameworks. It is already being used by big companies such as Alibaba, Google, and Toyota.

Throughout the previous sections, we tried to sum up the most popular development tools to build applications for mobile devices (i.e., web, hybrid and native applications). Of course, we could not cover all the development tools that exist. This summary shows the exponential grown of these environments and although it keeps pushing the technology forward, engineers may have a hard time when choosing a tool one for their use cases. Therefore, it is crucial to assess them and figure out which one is suitable for each use case. Benchmarking is a possible solution to achieve this. The following section will go through important concepts regarding benchmarking and why it can help to rank different types of software according to certain properties.

2.4 Benchmarking Concepts

Benchmark can be defined as "standard tools that allow evaluating and comparing different systems or components according to specific characteristics (performance, dependability, security, etc)" [48, 49].

This allows customers and vendors or to assess and compare different products to ease the process of selecting a product or to help to improve the products.

In order to make a fair and meaningful comparison, a benchmark must be highly representative: i) the conditions and setup in which the benchmark is performed must be representative of realistic scenarios, ii) the properties (e.g., performance, security) of the product that are the target of the benchmark must be representative of the main functional and non-functional requirements of the product, and finally iii) the metrics chosen for the evaluation must be representative of the properties under assessment [50].

When conducting a benchmark, four main components are key:

- **System Under Benchmark (or SUB)**: refers to the product that will be assessed in the benchmark (e.g., in our experiment they are the selected development tools).
- **Rules/procedure**: specify what needs to be followed during the benchmark campaign. The procedure must be easy to follow because the benchmark needs to be easily re-executed.
- **Workload**: refers to applications input values, which determine the type of operations that should be executed during the benchmark. The main associated challenge is workload representativeness.

[3] https://github.com/necolas/react-native-web

- **Measures (or metrics)**: refers to measurement tools to be used for evaluation of the SUB's characteristics (e.g., page load time). Measures determine what data and how to be collected for calculating the value of the metrics.

A benchmark can target several properties. However, in our study, we are only concentrating on two primary ones:

- **Performance**: focus on a specific domain and compares different systems (e.g., databases or operating systems). Commonly used by vendors to promote their products (marketing). The problem with this type of benchmark is that during the tests it is assumed that there are no errors and everything always works. In our experiment, we focused on performance and measured several metrics such as CPU and RAM usage, response times (see Section 4.1.3).
- **Dependability**: focus on whether a system reacts well in the presence of problems. Dependability integrates the following attributes: Availability, Reliability, Safety, Confidentiality, Integrity and Maintainability [51]. Includes performance benchmarking and fills its problem. Faults are inserted to see how the system reacts. The faults inserted do not take into account the malicious behaviors that result from the exploitation of some vulnerability in a system. This means that besides the workload present on performance benchmarking we also have a fault load [51]. Although our experiment did not include dependability, we believe it can easily be extended to support so (see Chapter 3).

The results of the benchmark enable comparisons between the systems under benchmark according to the properties that we target.

Chapter 4 explains the experiment we conducted and how we used these benchmarking concepts to compare different development tools capable of producing web applications in terms of performance.

2.5 Evaluation of Web and Native Applications

This section presents several studies that assessed web and or native applications for mobile devices. Several works in the literature were found that focus on comparing different mobile applications [3, 45] or different web applications [9, 52]. Some studies compare mobile and web applications [53, 54]. However, a lack of representativeness was observed in the chosen features of the applications under test (e.g., [54] only considers two features: geolocation and camera). Also, sometimes, the metrics used were not sufficient to effectively and precisely evaluate and compare different development tools against each other (e.g., the same study [54] does not consider the application package size, battery consumption, or other important metrics in mobile devices).

2.5.1 Benchmarking web applications

In 1998, a "self-configuring, scalable benchmark that generates a server benchmark load based on actual server loads" was presented [55]. In other words, they performed bench-

marking tests (heavily based on server load, throughput and response time as opposed to the existing benchmarks at the time that only focused on throughput and nonexistence of faults) on traditional web servers. Despite the paper being from the last century, it contains several phrases that could be applied to the current times, such as: "we need realistic and meaningful ways of measuring their performance" or "Current server benchmarks do not capture the wide variation". They claim that benchmarks of the time did not take into account that faults occur, and they are right: few current benchmarks do. This paper makes a fundamental claim: "although people are impossible to model (the end-users of a software), we cannot just stop benchmarking it". This phrase means that even if engineers cannot predict all user actions, it is still important to assess how different systems behave after a subset of common user actions.

In 1999, a study was published comparing different schools' and professional organizations' websites [56]. They reveal interesting metrics for the time being, such as "how fast does the main page of the website load" or "does the site provide text-based navigation for users who may have a slow connection or poor graphics ability" (the latter also considering faults - slow connection). This reveals that applications must not leave users apart who have fewer conditions. At the time, websites were already "much more than simple graphically-oriented hypertext" and were "becoming key entry points for customers and clients of an organization". Also, they provided an interesting explanation of why they performed this benchmark: "discover the'best practices" of organizations related to this study's sponsor" in order to improve their own website.

Around a decade later, in 2008, a benchmark of Web 2.0 was published [12]. With the introduction of Web 2.0, new benchmarks were necessary. They presented a set of automation tools to generate load and measure the websites' performance. They developed two similar applications in both PHP and Ruby on Rails that provided the same functionalities. Among other works, we inspired ourselves in this approach to develop our assessment framework. These applications could also be applied to the same workload and deployed in different deployment environments. They discovered that the bottleneck in these applications is the persistence tier (i.e., accesses to the database or file system). This was the first study that aimed to create a benchmark for the Web 2.0.

Similar to the previous work, in 2011, a study provided an architecture for benchmarking frameworks to develop the Web 2.0 applications (e.g., Ruby on Rails or PHP) [57]. These frameworks emerged to simplify the web application development and, as said before, needed to be benchmarked. They claimed that traditional web benchmarks such as RUBiS or TPC-W only took into account simple web applications that generated two to six requests to fetch its whole page content. At that time, [Ebay.com](http://ebay.com/) and [Amazon.com](http://amazon.com/) required 28 and 141 browser-interactions to fully fetch the index page, which reveals the lack of representativeness of such benchmarks. These new technologies brought an entire new level of complexity in terms of interactions with the web-browser that needed to be assessed. Therefore, they proposed a modulated architecture that can be used to benchmark the traditional way to develop applications (i.e., Web 2.0).

The study [58] compares a Progressive Web Application (PWA) with a Regular Web Application (RWA). They used the same template for both applications and implemented the PWA features in one of them. This resulted in two identical applications: one that had

PWA capabilities (e.g., work offline) and the other that had not. As expected, since content can be served from the cache, they observed that on repeated visits speed index the PWA outperformed the RWA. They also made another claims that although interesting, we have some thoughts about:

- "Https connection for PWA is slowing down all of the PWA's performance metrics on the first visit": Of course, encryption introduce overhead and is always slower than no encryption. However, using HTTPS is crucial, and it should be impossible to produce web applications without it. This process should be applied to any web application (regardless of whether it is RWA or PWA).
- "Memory consumption on PWA increased more than 2 times the size of RWA": This problem can happen due to poor caching policies (e.g., caching all the application assets of course will increase memory consumption, but in most times it is not necessary).
- "Hence this paper will cover unique features of PWA that a general web programmer might be unfamiliar with and compare the result with regular web site to see if it worth it to build/rebuild your web site with PWA features and requirements": It is always important to consider converting a Regular Web Application into a Progressive Web Application to improve the user experience. As stated before, ideally, a PWA is just a RWA with more performance, accessibility and security. Also, it is not necessary to rebuild the entire application to convert it into a PWA, this can be an *ad-hoc* process.

In 2017, the energy efficiency in Progressive Web Applications was studied [25]. As we saw before, service workers are the brains behind these type of applications. They define service worker as "a set of APIs that allows developers to programmatically cache and preload assets and data, manage a JavaScript module running in its own thread and providing generic entry points for event-driven background processing (e.g. reaction to the receiving of a push notification)". Because service workers run on the background, they have a cost. In this study, the impact of service workers in an application in terms of energy efficiency was assessed. They studied seven different real PWAs running "with and without service workers, under different network conditions (2G and WiFi), and on different mobile devices (i.e., low-end and high-end)". They concluded that although in most situations the service workers have not a big impact in the energy, developers must be careful on how to implement them.

TechEmpower benchmark [9] is a project currently maintained (at the time of writing). It measures the performance of multiple operations (e.g., JSON serialization, database queries, the return of those queries to the HTML response that is sent to the client, etc.) between multiple traditional web applications (web 2.0). This benchmark is considered a standard, but it only focuses on the traditional web applications and does not take into account the modern web: Single Page Applications.

Web frameworks [59] is a benchmark that aims to test also Traditional Web Applications as TechEmpower benchmark [9] but more limited. It is also currently maintained (at the time of writing) and only focus on three HTTP calls in different frameworks. There are no database calls or any kind of computation, and everything always goes well in the tests

(there are no faults inserted). Therefore, the workload is not representative. As the first one, it only focuses on traditional web applications.

JS-Repaint-Perfs [60] is a benchmark from 2016, and it is not currently maintained. It tries to measure the repaint rates of JavaScript libraries to create Single Page Applications. This benchmark uses random computations (using the Math module) to simulate queries that are far from representative, but since the purpose is only to simulate the repaint-rates of each library it could be enough. Also, the tests have no faults (i.e., everything always behaves as expected).

js-framework-benchmark [52] is another web application benchmark. It is currently maintained (at the time of writing) and consists of multiple operations around an HTML table. This HTML table are presented in applications developed using different SPA Frameworks. This was the first benchmark we found that aims to assess different Single Page Applications. However, the features in the application tested are not representative of what a SPA is capable of (i.e., CRUD - create, read, update and delete - operations around a table may be one use case of an SPA, but it is not the only one).

`Realworld` [61] defines a Reference Application Specification which is a clone of Medium, a blog application, and implements it with different development tools. It is a really popular tool and counts with contributions from more than 50 developers. This project covers a lot of tools and contains some performance comparisons between the development tools. However, it restricts the focus to only a simple blog application which we believe it is not a representative application that explores the functionalities of mobile devices (e.g., camera, geolocation).

2.5.2 Benchmarking native applications

Alongside the web, there are a lot of work in the mobile environment and its comparison to the web - primarily against PWAs. In general, all studies provide great theory explanations of the frameworks under study but lack when it comes to benchmark them. The studies that we found that perform some kind of benchmarking end up lacking on representativeness (the same problem in Web frameworks [59] and in js-framework-benchmark [52] benchmarks presented before) resulting in simple applications that cover few use cases.

In 2019, a study that focused on Flutter, a framework to develop multiple applications from the same codebase, and compares it to Apache Cordova (nowadays called Ionic) and Native (e.g., iOS with Swift and Android with Java) was published [45]. They developed the applications in those languages. These applications features were: create and edit tasks. Metrics such as source code lines; number of files; dependencies; application package size and installed size; RAM usage, startup and view transition timed. Tokei to perform a static analysis between the applications. They concluded that both native applications (i.e., Android and iOS) resulted in three times the lines of code of both flutter and ionic; in terms of files: native combined had 36, flutter 16 and ionic 27; regarding dependencies native combined had 10, flutter 7 and ionic 36. Regarding performance profiling and application characteristics, they concluded that:

- "Flutter versions were the fastest and most responsive versions while the Cordova/Ionic

versions were slow to start"

- "the overhead of the cross-platform frameworks has a noticeable impact on memory requirements"
- "flutter had the highest values (installation size of 41.29 MB on Android and 74.00 MB on iOS) and ionic the lowest (with 2.69 MB on Android and 7.90 on iOS)"

A performance analysis of a fully functional mobile application implemented with platform ten **cross-platform tools** (or CPT, term that we use several times in our work described in the beginning of this chapter) and native for Android, iOS and Windows Phone operating systems was published in 2016 [3]. As we talked before, maintenance is a big problem in native development, which lead to the appearance of cross-platform tools. The latter use web containers (i.e., web views) which makes it easier for people with web background but many times at cost of performance. They felt that this assessment was needed because there are many cross-platform tools. The application was developed by experts from each tool but since it is from 2016 some technologies used are already outdated and followed the PropertyCross specification [62]. However, the technologies chosen were justified and representative (i.e., they chose technologies that used different programming languages, which can be a key factor for a company). The application was tested in both high-end and low-end iOS and Android devices, and they are factory reset and updated to the most recent operating systems available at the time. The metrics collected (and tools to collected) were the following:

- Response Time: launch time of application (from tapping the app icon to display the main screen); time to load new page (favorite page used because it was the only one that did not require internet access); time to return to the previous page. Measuring tool was DDMS in Android (using difference between the timestamps) and instruments tool in iOS.
- CPU Usage: overall CPU usage of application and in two specific actions. Measuring tool was ADB in Android and Instruments tool in iOS.
- Memory Usage: Memory allocated when: the application fully launches; after visiting each page in a specific order; after each page is visited. These values were calculated while navigating through the app in a specified order. Measuring tool was ADB in Android and instruments tool in iOS.
- Disk Usage: space taken by the installed app; size of the APK for Android and IPA for iOS. No measuring tool was needed because such values are available without any.

Battery usage was not considered, and they claimed that "impact caused by overhead of CPT is minimal".

They concluded that the disk space used were less in native technologies than in CPT because they do not need web view and runtime packed in the application size. They also concluded that the frameworks that depended on JavaScript were the most CPU intensive and with the slowest launch times, but achieved similar navigating response times to native.

2.5.3 Benchmarking web and native applications

A comparison between a Native Android Application (NAA) and a Progressive Web Application (PWA) was published in 2017 [54]. They developed the same app in both native Android with Java and PWA with React.js. They only focused on two features: geolocation and camera, and metrics collected were its response times. They recognized that they tested few features and more APIs must be assessed (e.g., push notifications, startup time of applications, file access, etc.). Their results revealed that geolocation in PWA was significantly faster - "This result shows that applications that relies heavily on maps could be developed as a PWA instead of a NAA, making the development process faster since two platforms can use the exactly same code".

Another comparison between applications developed using different frameworks was published in 2017 [53]. They developed the same application using both ionic, react-native, and React.js (for the PWA). However, both the metrics and the features of the application were not representative. The metrics used were:

- Size of installation
- Launch time
- Time from app-icon tap to toolbar render

According to their results, the progressive web application only lost to the other two in the third metric, winning the other by far. They also claimed that: "Progressive web apps enable the best of both approaches, where end-users can easily experience an application through their web browser, then choose to install it via an "Add to Home screen" banner prompted".

Another study dived into Flutter [47]. Provides an explanation about the differences between Native development and two hybrid technologies - React Native and Flutter. They claimed that for native applications "platforms expose exclusive high-level APIs (Application Program Interface) that are used to implement the user interface, I/O (Input-Output) operations, and other features" resulting in "a tailored look and feel of applications on each platform that most users have become accustomed to". This is completely true, however, since with React Native, we can use the native widgets, and Flutter provides the material widgets, we can also develop applications that users are accustomed to with those two other frameworks.

An article that explains the different caching strategies work and how PWA provides offline capabilities was published [63]. It contains tests related to the cache. Using blazemeter to test the same application made as an Android app and PWA, they discovered that the PWA's average response was three times faster. After analyzing their tests, in the discussion, they reveal that the PWA performance of the same app is better than its Android version. Also, as expected, they reveal that the caching process increases the performance of a PWA.

The study containing a brief comparison between React Native, native applications, and PWAs was published in 2018 [64]. They made inquiries about user experience regard-

ing each application and its features. It contains a table containing several features and whether a tool supports them or not.

2.5.4 Summary

The Table 2.1 contains a brief summary of all related work dicussed in the previous sections.

Table 2.1: Related work summary

ID	Study Type	Description	Contributions	Limitations
BW1 [55]	Benchmarking Web	Benchmark traditional servers	"although people are impossible to model, we cannot stop benchmarking"	Outdated
BW2 [56]	Benchmarking Web	Performance of different websites	"discover the best practices" to improve their website	Outdated
BW3 [12]	Benchmarking Web	Benchmarking traditional web applications	multiple applications developed with web frameworks (assess apps to assess the frameworks)	Outdated
BW4 [57]	Benchmarking Web	Benchmark traditional web	traditional web benchmarks simple web applications	Outdated
BW5 [58]	Benchmarking Web	Web Application vs PWA	repeated visits speed index the PWA outperformed the RWA	wrong practices when developing apps
BW6 [25]	Benchmarking Web	Assess Energy Efficiency of Service Workers	Service workers have not a big impact in the energy	-
BW7 [9]	Benchmarking Web	Benchmark traditional web frameworks	Well-structured, maintained	do not consider SPA
BW8 [59]	Benchmarking Web	Benchmark traditional web frameworks	Maintained	Workload not representative; do not consider SPA
BW9 [60]	Benchmarking Web	Repaint rates of SPAs	Focus on SPAs	Outdated and not maintained
BW10 [52]	Benchmarking Web	Benchmark of different SPAs frameworks	Focus on SPAs; well structured	Workload not representative; Faultload not included
BW11 [61]	Benchmarking Web	Benchmark of different SPAs and Traditional web frameworks	Focus both on SPAs and TWAs; Well-structured; Specification well defined	Features presented in application are not representative in mobile devices (simple blog app)
BN1 [45]	Benchmarking Native	Flutter vs Ionic vs Native	Good metrics;Flutter the fastest;Ionic the slowest to start;Frameworks impact on memory and size	Workload not representative
BN2 [3]	Benchmarking Native	Performance analysis of several CPT	Representative metrics. JS Frameworks are CPU intensive	Battery usage not considered; no faultload
BN3 [47]	Benchmarking Native	Flutter vs RN	-	-
BNW1 [53]	Benchmarking Native and Web	PWA vs Native App	PWA smaller and with faster launch times;Lost in navigation time	Workload and metrics not representative
BNW2 [54]	Benchmarking Native and Web	PWA vs Native	Geolocation+Camera faster in PWA	Workload not representative
BNW3 [63]	Benchmarking Native and Web	PWA vs Native	Caching increases the performance of a PWA (three times faster)	-
NW1 [64]	Native and Web	Inquiries about PWA vs React Native vs Native	-	-
NW2 [65]	Native and Web	UX comparison between Native vs PWA	-	-
NW3 [5]	Native and Web	Technologies to develop different apps	-	-
NW4 [66]	Mobile and Web	PWA explanation	-	-
NW5 [27]	PWA Stats	Native and Web	Website with PWA statistics	-

As observed throughout this section, the related studies lacked when selecting the features to implement the applications under assessment or which measures to gather when the applications were being assessed. Also, there is a lack of studies that compared applications from different environments (e.g., hybrid, native, and web). This is crucial because the differences between these different types of applications are narrowing, and there is a vast offer of development tools to develop them. In the following chapter, we will explain how Savery addresses these problems.

Chapter 3

SAVERY Framework

Developers have, nowadays, countless solutions to support them in developing mobile applications. However, they lack ways that allow them to select which are the best development tools for their specific case study. Existing studies comparing different applications for mobile devices lack in choosing suitable features or metrics.

In this chapter, we present **SAVERY, a framework for the assessment and comparison development tools capable of producing applications for mobile devices**. Comparing development tools is a challenging proposition, as it is not feasible to effectively compare them directly. Instead, the usefulness of comparing them is to learn which one helps to create better (e.g., faster, lighter, with less overhead) applications or helps to create applications with less effort or knowledge from the developers. We argue that it is possible to make this comparison through the output of these tools, i.e., the developed applications. With this approach, one can evaluate and compare several properties such as performance and dependability (or even security) of the resulting applications.

As recommended in the best benchmarking practices, the assessment and comparison of different solutions must be performed in a fair and useful way [48]. Thus, this framework carefully lays out the relevant components and procedures to develop benchmarks that can support this type of evaluation in a fair, effective, and representative way.

The framework was designed considering several quality attributes in mind, such as performance, reliability, dependability. It was also designed to be extensible, so that in the future it can support the assessment of security of the developed apps and even other properties that are very relevant for development tools: the learning curve and the speed or ease of development. The framework is prepared to be open source and to receive the contributions of the community, both in extending it and providing new implementations of the applications.

Fig. 3.1 depicts the principal components and the procedure to be followed to build such framework. It details the required steps to be taken for the definition of the benchmarks that have specific targets in terms of applications and quality attributes. As we can see, there are four key phases, which are discussed in the following sections.

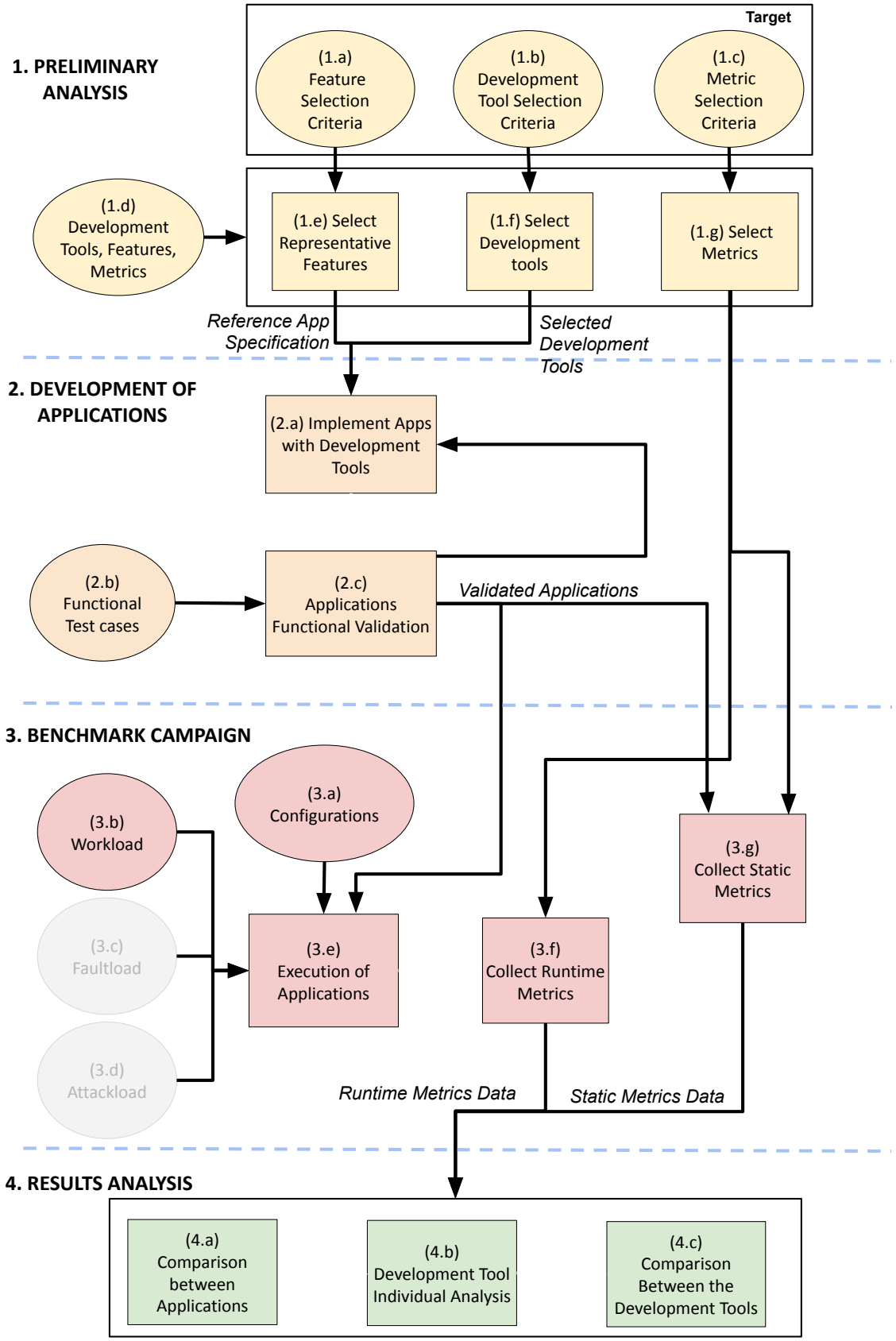


Figure 3.1: Overview of the components and steps defined in the SAVERY Framework.

3.1 Preliminary Analysis

This phase defines the **target of the benchmark** in terms of the application domain of interest, types of development tools to be adopted and quality attributes of interest. This target is decisive for specify the **feature selection criteria**, the **development tool selection criteria** and the **metric selection criteria**.

Based on this criteria, we need to analyze and select a representative list of features of mobile applications. This represents the set of features that the applications under test should support and should correspond to classes of applications that might be of interest of groups of developers. For this, it is necessary to study the most common features in the mobile applications for the domain of interest defined as target for the evaluation to be conducted. The result is a **reference app specification**, which is a specification that details which functionalities should be implemented in each application and the respective interactions, and a set of functional tests to those functionalities.

3.1.1 Reference App Specification

Based on these criteria, we need to analyze and select a representative list of features of mobile applications. This represents the set of features that the applications under test should support, and should correspond to classes of applications that might be of interest to groups of developers. For this, it is necessary to study the most common features in mobile applications for the domain of interest defined as the target for the evaluation to be conducted. The result is a **reference app specification**, which is a specification that details which functionalities should be implemented in each application and the respective interactions, and a set of functional tests to those functionalities.

3.1.2 Development Tools

Next, we need to analyze the existing development tools for mobile applications to select a representative list of them (i.e., commonly used and recommended by the community). There are only two requirements that restrict the available development tools:

1. They must be able to output applications for mobile devices, which can either be native, web, or hybrid;

2. They need to provide the ability to identify each element of the application with a unique ID.

   This is crucial because we propose that the applications are treated as black boxes with the help of the functional testing (more details in Section 3.2.3). Therefore, assigning unique identifiers to each application element is the only way to identify, and perform successful operations with them (e.g., grab an input field and click them or verify if an image is presented on the screen). For example, several automation libraries can identify elements in web applications if they have an unique ID assigned to them:

```
<button
    id="UNIQUEBUTTON"
    >
    Automation library should be able to click me!
</button>
```

Although our framework defines few requirements for this, it will also depend on the library used to perform the functional tests. In the experiment conducted, an extensive analysis, we found that Appium and WebdriverIO can be great matches for this framework (more details in Section 4.2.3).

3.1.3 Metrics

Finally, we need to select adequate metrics to collect. These metrics depend on the quality attributes that we are interested, and will help to provide insightful comparisons between each development tool.

They will be collected during the third phase when executing the benchmarking campaign.

For this, we propose two main types of metrics:

- Static metrics: metrics that are collected before running the campaign tests. Includes, for example, application size, line of codes, dependencies number (and dependencies number only for development), and build times. This type is more focused on providing insights regarding the developer experience rather than the user experience.

- Runtime metrics: metrics that are collected during the execution of the applications (i.e., during the execution of benchmark campaign test cases). Depending on the benchmark focus, it may include, for example, response times, CPU usage, and RAM consumption.

The outputs of this phase (i.e., selected tools, reference app specification, and selected metrics) are used in the following phases.

3.2 Development of Applications

In the second phase, **Development of Applications**, we prepare the applications for the benchmarking campaign (i.e., third phase). This phase is responsible for developing the applications following the reference app specification and using the selected development tools; both defined in the previous phase. After implementing the applications, they need to pass through a validation phase with the help of functional testing. This process aims to verify if the applications are correctly implemented or not. It is also in this phase that auxiliary tools (e.g., to help the benchmark execution, to analyze the results) should be developed.

This phase outputs the validated applications prepared to be assessed in the third phase and possible auxiliary tools to be used in the following phases.

3.2.1 Auxiliary Tools

During the preparation phase, all the auxiliary tools to aid the main processes of the framework (e.g., functional validation, execution of the applications, analysis of the results) must be built. The auxiliary tools can be multiple:

- A tool to send the functional tests to each application during the validation process and report which parts of the application are incorrectly implemented.
- A tool to send the workload during the benchmark campaign while gathering measurements.
- A tool to analyze the results (i.e., the measurements gathered) from the benchmark campaign and generate different tables and charts for analysis.

3.2.2 Application Development

Ideally, applications should be developed by engineers that are fluent in each specific development tool to make sure that the implemented application adopts the corresponding best practices. When this is not possible, bias problems should be mitigated by reviewing and improving the implementations or having multiple versions for the same application. In the cases that the benchmark is of interest to a broad community, it is recommended to be open to their respective contributions. For example, in our experiment, we run into a bug in one of the implementations that could be easily avoided if the applications were peer-reviewed before moving to the validation process. Instead, we only detected the error in the validation phase delaying the validation of the applications (for more info, check Section 4.2.2).

This process represented in Fig. 3.2) receives the Reference Application Specification and the set of development tools selected from the previous phase, and outputs the applications implemented.

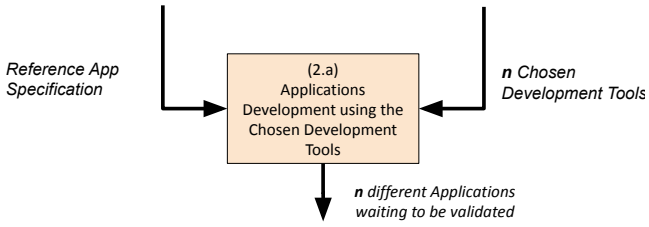


Figure 3.2: Steps for the development of the applications.

During the process, the applications will be implemented following the reference app specification and using the development tools selected. It is important to note that the application elements must have unique identifiers (these identifiers are defined in the Reference App Specification) to allow the functional tests to perform operations around the application elements.

The output of this process will be the different applications implemented but yet to be validated. They may return to this process again if they fail to pass the validation phase.

3.2.3 Functional Validation

After all the applications are implemented, we need to confirm that they have no issues before proceeding to the benchmarking campaign. Therefore, all the applications will go through a validation phase. This process is presented in Fig. 3.3. It receives the different applications implemented from the previous process and set of functional tests. If there are no issues with the implementations, they are free to proceed to the campaign phase. Otherwise, they will return to the Application Development process (Sec. 3.2.2) to fix the existing issues.

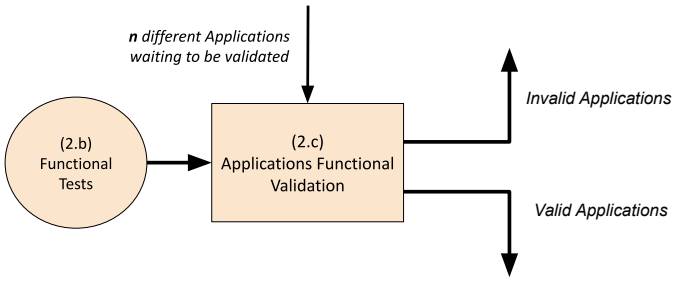


Figure 3.3: Steps for the development of the applications.

We propose that the implementations are validated with functional testing without inserting any custom code as some testing libraries do (see more in Sec. 4.2.3). This is done because of the following main reasons:

- To keep the tests as **non-intrusive** as possible.
- Since this framework targets development tools that output applications for different environments (e.g., web, native, or hybrid), we want to develop tests that can be reused across the different implementations and in different phases (e.g., preparation and campaign).

So, given that the applications can be either native, hybrid, or web, we propose that the functional tests are implemented using a technology compatible with all these environments (more on how we choose the technology for our benchmark in Sec. 4.1.2). Otherwise, we would need to use a different technology for each environment, which would result in redundant tests with the possibility of introducing more bugs unnecessarily.

By treating the applications as black boxes, the functional tests will be non-intrusive, and we bring the applications under test closer to what happens with real-world applications without adding any custom code to each implementation to perform the tests.

One validated application means that it is correctly implemented and follows the Reference App Specification defined in the first phase. If this happens, all the components in the application under validation are presented, and each component presents its functionalities working (e.g., Camera component is presented application and renders a camera feed and button to capture images from the camera feed). If an implementation presents issues, it will need to return to the development process to fix the issues before returning to the validation process.

When all the implementations are successfully validated, they can proceed to the campaign phase and are ready to start the benchmark.

3.3 Benchmark Campaign

The benchmarking campaign phase is responsible for executing the benchmark and gathering the measurements. As inputs, it receives the validated applications from the second phase and the selected metrics from the first phase.

After respecting certain configurations, a workload will be submitted to the validated applications. At the same, the defined measurements are gathered according to the previously defined runtime metrics. It is also in this phase that the applications are analyzed in order to collect the static metrics.

Attackload and Faultload in Fig. 3.1 and in Fig. 3.4 are grey-shaded because we did not include them in the experiment we conducted. However, future benchmarks may include them since our framework can be extended to support these two types of loads.

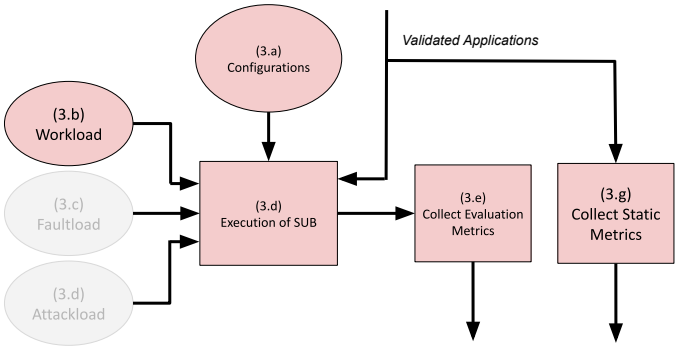


Figure 3.4: Steps for the execution of the applications implementing the reference specification.

3.3.1 Configurations

The configurations are the set of rules that we must respect before starting each execution. The following list enumerates examples of such configurations:

- If the applications under test are web-based, they must always be served under an encrypted connection using HTTPS.
- The remote server that serves the web applications must isolate them from each other to have fair comparisons.
- Permissions are not granted to any application. This is done to emulate what happens in real-world scenarios. The workload must include a warm-up period to perform this type of operation (e.g., accept the required permissions before the main tests start).

- The target device (e.g., iOS or Android device) that is running the application cannot have background processes running. Before starting each campaign, the device must also be restarted to assure that all the processes are cleaned and the previous tests have no impact on the subsequent ones.

When the framework is instantiated, rules can be added or removed depending on the systems that are being assessed.

3.3.2 Workload

A benchmark **workload** refers to the applications' input values, which determine the type of operations that should be executed during the benchmark. The main associated challenge is workload representativeness. The workload should emulate what an end-user of the application would do and comprise a set of functional tests that cover all the application functionalities in one or more sequences.

The process of sending the workload to the applications under test should be automated to ease the reproducibility of the benchmark, i.e., to facilitate the execution of the campaign several times. We recommend that each application is tested 30 times.

3.3.3 Measurements Gathering

The process of gathering measurements has some requirements:

- Must be reproducible, i.e., easy to execute since the campaign must be executed several times.
- Must be non-intrusive: the goal is to minimize the impact of this process on the applications to have transparent results.
- Must be fair to all types of applications under test, i.e., can not be more expensive to collect metrics in Web applications than in Native ones (e.g., gathering the CPU usage cannot be more expensive when assessing web applications than native ones).

When in need of an external library to count, for example, the lines of code of the applications, one should seek, when possible, libraries that support all the applications under test, i.e., avoid using different libraries to collect the same metric.

Also, this process should be as simple and naive as possible to keep the campaign complexity low. For example, if all the applications are JavaScript-based, to collect the dependencies number, this process can be as simple as analyzing the `package.json` file. When possible, tools that are already available should be used to the detriment of adding new libraries (thus, increasing the campaign complexity). For example, if the target device is an Android one, ADB commands can be used to query important metrics such as CPU usage, RAM consumption, battery levels, among others (identical situation when the target is an iOS device).

3.4 Result Analysis

Finally, the fourth phase, **Result analysis** concludes our framework. In this last phase, all the measurements are received from the previous phase.These measurements will allow the evaluation of each application. Consequently, this analysis will provide comparisons between the different development tools that produced the applications. Fig. 3.5 presents the proposed folder structure to save the results in a well-organized manner.

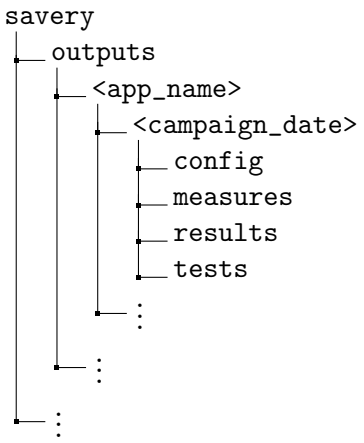


Figure 3.5: Proposed folder structure to save Campaign results

Each campaign execution should output the following data:

- `config`: configuration used for the current campaign;
- `measures`: list of measurements gathered during the campaign execution (e.g., CPU usages, RAM consumption);
- `results`: additional information about the campaign (e.g., start-up duration);
- `tests`: tests information (e.g., outcome, duration, timestamps);

This section concludes the presentation of our framework. As we saw, with four equally important phases, SAVERY tries to fill the gap that currently exists when assessing different tools that exist to produce applications for mobile devices. In the following chapter, we described the benchmark that instantiated to demonstrate the applicability of the proposed framework.

This page is intentionally left blank.

Chapter 4

Performance Benchmark of Mobile Development Tools

The framework presented in the previous chapter guides the definition of concrete benchmarks for a target that consists of application domain of interest, types of development tools being considered and quality attributes of interest. It leaves to the user the freedom to define the benchmarks according to their interests, and therefore, more useful. However, as many users have similar interests, a way to increase the adoption of the framework is to accompany it with concrete benchmarks for targets that are of general interest.

In this chapter we present an instantiation of the SAVERY framework in a **concrete benchmark for a target of entertainment and utility applications, development tools that are popular in the JavaScript community, and with a focus on performance attributes**. For this, it was necessary to study the popular entertainment and utility applications to learn what are the most common and representative features to be considered. To be evaluated, we selected `9` widely used mobile application development tools and considered metrics of response time and resource consumption.

We developed `9` mobile applications with the defined set of features, each one implemented by one of the selected development tool. These applications were validated according to the set of functional tests defined for the specified functionalities, and then were subject to the benchmarking campaign while measurements were gathered for posterior analysis. The in-house development of all applications may introduce a bias related to our specific programming abilities, but this affects similarly all the alternatives, and that was mitigated by following a reduced yet solid set of the best development practices. This problem will be further mitigated due to the open nature of the benchmark, with the support of the community that will suggest changes or propose new implementations.

The benchmark and the related materials, including documentation, sources of the applications developed for the experiments, and the results of the experiments are available at:

- `https://savery.dei.uc.pt`

The benchmark instantiates the components and the four main phases of the SAVERY framework presented in Chapter 3. The ensuing sections present in detail the three first phases, while the phase of *Result Analysis*, will be presented and discussed in Chapter 5.

4.1 Preliminary Analysis

4.1.1 Reference Application Specification

Considering entertainment and utility applications, we leave out of the scope of our study other types of applications such as games, payment, and chat. In order to understand which features the reference application specification should support, we analyzed the most popular applications and their primary features. Table 4.1 provides the overview of our analysis of several sources, including [67, 68, 69, 70]. As we can observe, that all the applications rely somehow on native features such as camera access, geolocation, and notifications.

Table 4.1: Popular applications, download count (in Google Play Store) and their features

App Name	**Download Count**	**Features**
WhatsApp	1-5 billion	audio call, camera, carousel, chat, login, notifications, video call
Facebook	1-5 billion	camera, carousel, chat, content from external database, live feed, login, notifications, reviews/comments
Facebook Messenger	1-5 billion	audio call, camera, carousel, chat, login, notifications, video call
Instagram	1-5 billion	camera, carousel, display images/videos from web/cache, display posts from database, live videos, login, notifications, reviews/comments
Snapchat	500m - 1	billion,camera, chat, login, notifications, short videos
UC Browser	500m - 1	billion,browser
Uber	100-500 m	geolocation, login, maps, notifications
Youtube	1-5 billion	display images/videos from web/cache, notifications, video player
Netflix	100-500 m	display images/videos from web/cache, login, payments, notifications, video player
SHAREit	500m - 1	billion,file access, share files over the web
Bitmoji	100-500 m	camera, file access
Google Search	1-5 billion	content from external database
Google Maps	1-5 billion	audio player, geolocation, maps
Amazon	100-500 m	content from external database, deliveries, payments, reviews/comments
Twitter	0.5-1billion	camera, carousel, content from external database, notifications, reviews/comments
Pinterest	0.5-1 billion	camera, carousel, content from external database, notifications, reviews/comments
Google Pay	1-5 billion	payments
Shazam	0.5+ billion	content from external database, display images/videos from web/cache, microphone
Spotify	1+ billion	audio player, content from external database, login
ZOOM	0.5+ billion	audio call, camera, chat, video call, screen sharing
Cash App		login, payments
Google Meet	0.1+ billion	audio call, camera, chat, video call, screen sharing
Microsoft Teams	0.1+ billion	audio call, camera, chat, video call, screen sharing
TikTok	1+ billion	camera, carousel, display images/videos from web/cache, display posts from database, live videos, login, notifications, reviews/comments
Telegram	0.5+ billion	audio call, camera, carousel, chat, login, notifications, video call

A large part of these apps require login to access the main features of the application. Also, most of the applications rely on fetching content (e.g., images or videos) from an external source. Given these observations, we designed the structure presented in Fig. 4.1.

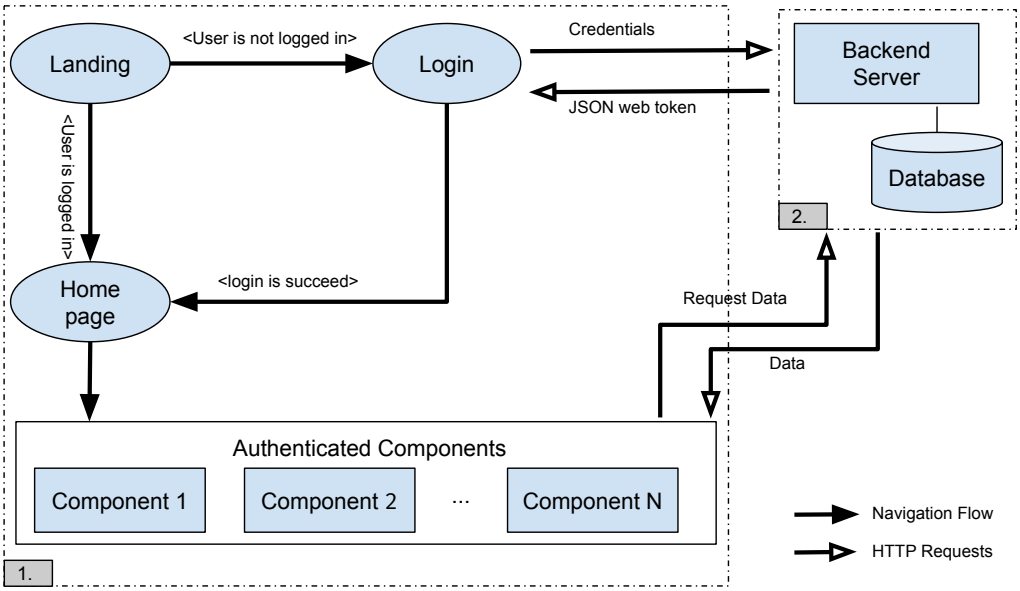


Figure 4.1: Overview of the main functionalities and navigation of the application.

According to this architecture, there are two main components, described below:

1. Mobile Application: application that will be developed in the Preparation Phase. According to our analysis, the application flow presented is visible in the majority of the most popular applications in the Android Store. It divides the application in two parts: the authenticated and not authenticated. The authenticated parts are only accessible after the user successfully logs in, while the other pages are accessible if the user is not authenticated.

2. Backend Server: appears to support the mobile application because some application components may require extra functionalities that cannot be done in client-side (i.e., in the mobile application itself). For example, the authentication process (i.e., verifying if the user credentials are correct).

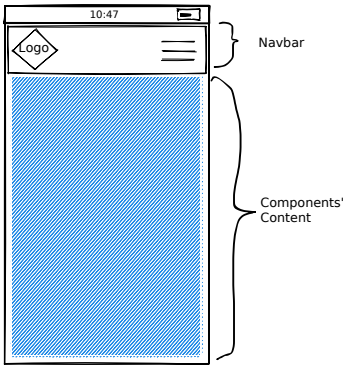
Mobile Application

When first visiting the application, the user is presented with a landing page. In this page, there is a button to request the required permissions that may be necessary inside the application. There is also another button to navigate to the Login page. Once the user navigates to the login, is presented with a form where he can insert his credentials (i.e., a username and a password). After submitting the credentials, an HTTP request is sent to the backend server that will then return a unique token in case the credentials are correct, and the user will be redirected to the homepage. This authentication process is visible in almost any popular mobile application nowadays. Once the user is authenticated, a Homepage is presented, and now, the user can access other eight carefully thought components that aim to cover the majority of the features presented in the Table 4.1.

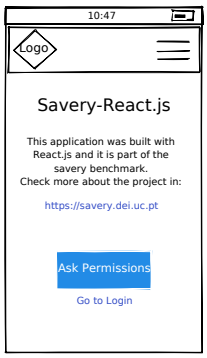
These components may also require interaction with the backend server, which is done via HTTP requests. The list of authenticated components, i.e., components that are only presented once the user is logged in, is presented below:

- **Homepage**: renders a paragraph that tracks whether the user has given the permissions or not and a table with the components that the user can access. These components should be the remaining items of this list.
- **Camera**: renders the live camera feed. There is also the possibility to capture an image by clicking a button which will be rendered below. This is a native feature seen in almost all applications.
- **Geolocation**: renders user geolocation. This is also a native feature that is becoming very used in popular applications. According to [71], 30% of study respondents think that localized information and position-based information are crucial features.
- **File Access**: when a button is clicked, a file picker is displayed to select an image from the file system. The image will be later rendered below. This aims to cover the applications that render images after being uploaded.
- **Notifications**: after clicking a button, a local native notification is sent to the device. Notifications are crucial to all applications to engage their users.
- **Feed**: when the page mounts, several random posts are requested to our backend server, which contains a database with multiple posts. When the posts are received, they are displayed below, each containing text and an image. This page also has an input field to request more posts. When the submit button is pressed, the number of posts will be requested to our backend server and displayed below. It aims to emulate the fetching from external sources seen in popular applications, used in Techempower [9] and considered a key in surveys [71]. In a survey, 60% of the respondents consider that mobile apps that communicate with backend information are key [71].
- **Carousel**: when the page mounts, random images are requested and rendered to the screen, and as the user scrolls through the page, more images are requested and rendered simulating an infinite scroll.
- **Background Sync**: this component tracks the user connection and fetches content from cache. It also contains two buttons: i) one to fetch an image from the cache; ii) another to fetch a random image from the web. After the buttons are pressed, the images are present below to test the app's offline capabilities.
- **Expensive Operation**: contains several buttons to perform expensive operations in a table, such as creating 100 and 1000 rows, swapping or deleting rows. Inspired in another popular benchmark, it aims to test how the application reacts when submitted to expensive operations [52].

Besides the authenticated components presented above, the application also has two unauthenticated components:



(a) Application Skeleton



(b) Landing page component

Figure 4.2: Example of application layout

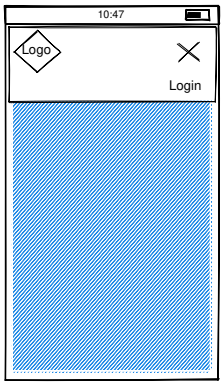


Figure 4.3: Authenticated Navigation Bar

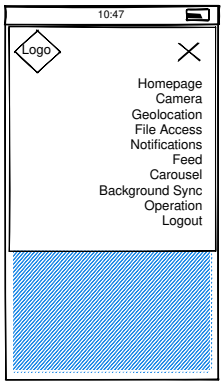


Figure 4.4: Unauthenticated Navigation Bar

- **Landing**: when the application opens, a Landing page is presented. It contains a small description about the technology used to build the application, a button to ask the required permissions, and a link to navigate to the login page.
- **Login**: renders a form that contains two text input fields and a button. The input fields are for the username and password, and the button to make a request to the backend server with those credentials.

Fig. 4.2 presents the layout that each implementation should follow. Fig. 4.2a the application skeleton. Each application page must always have a navigation bar with the application logo (to identify and distinguish each implementation, we used the logo of each development tool), and a button to open a bar that contains links to all the application components. The navigation bar should display different navigation links depending on whether the user is authenticated or not. The authenticated navigation bar is presented in Fig. 4.3 and the unauthenticated in Fig. 4.4.

Each application element must have unique identifiers to facilitate the process of functional testing. These unique IDs are listed in the Appendix A.

Table 4.7 presents the functional tests associated with each application component. Appendix B lists the screenshots of each application component of one of the implementations.

Finally, Table 4.2 presents a brief summary of the components previously-defined.

Table 4.2: Summary of components defined in the Reference App Specification

Component	**Requires Auth**	**Description**
Landing	No	First page of the application
Login	No	Page where user can authenticate
Homepage	Yes	Display the components available
Camera	Yes	Render live feed, capture photo and display it
Geolocation	Yes	Display user location
File Access	Yes	Upload image from file system and display it
Notifications	Yes	Send native notifications
Feed	Yes	Fetch posts from external sources and display them
Carousel	Yes	Load images infinitely as user scrolls
Background Sync	Yes	Track user connection and render images from cache and web
Operation	Yes	Perform expensive operations

We could have bootstrapped the Reference App Specification from an outside project like Realworld [61] or PropertyCross [62]. Although this projects were inspirations for our study, we did used them for our Reference App Specification because at least one of the following reasons: i) the primary focus of specification was not concerned on mobile devices; ii) if we opted to the applications from these specifications, less customization was possible; iii) when we found out about `Realworld`, we were already midway through our study, and already have defined the Reference App Specification was presented. These problems could be surpassed if we restricted the benchmark to only target blog applications. However, we wanted to target more application types for mobile devices. Hence, we opt to define our own Reference Application Specification from the scratch.

Backend Server

Some features in the Feed or Login components require a backend server. Hence, the Reference App Specification also requires that a backend that all the implementations can make requests to. It must serve two endpoints:

- `/api/login`: listens for POST requests that contain a JSON object in the body. The object must have a username and a password. This endpoint must check if this user exists in a database (this database should be implemented) and if the password is correct. If so, a unique token called JSON Web Token (or JWT for short) is returned.
- `/api/news`: listens for POST requests that contain a JSON object in the body. The object must contain a `numberOfNews` which is an integer indicating the number of posts that this endpoint should return. The endpoint should search the database for random posts and return them.

4.1.2 Selection of development tools

We selected nine different development tools to use in implementing applications to be tested. The following list enumerates the chosen development tools and their relevance for the benchmark:

- Tools selected to produce **web applications**:
  - **React.js**: currently the most popular tool to produce Single page Applications.
  - **Preact**: selected to assess whether its low overhead is verified in comparison to the other react-based tools.
  - **Next.js**: selected to confirm if it is currently the best react-based tool to build performant web apps.
  - **Gatsby**: was popular but losing market share for Next.js, and thus the comparison between them is important.
  - **Svelte**: selected to compare against React-based tools and compare the differences between interacting with the Real (Svelte) or a Virtual (React) DOM.
- Tools selected to produce **hybrid applications**:
  - **Ionic and Capacitor** (previously called Cordova): selected because produces cross-platform hybrid applications.
- Tools selected to produce **native applications**:
  - **Kotlin**: selected to be the reference of the benchmark due to its native nature.
  - **React Native**: selected because produces cross-platform native applications.
  - **Expo**: selected to compare if the abstractions introduced by Expo have a substantial impact compared to plain React Native.

All the development tools selected provide a way to identify the application elements with unique identifiers. To build the functional tests, we used an automation library called WebdriverIO [72]. This library to identify the application elements, needs unique identifiers. For web applications, WebdriverIO can easily identify the element if it has assigned a unique id:

```
<input
    id="Login_Input_Username"
    type="text"
    placeholder="Username"
    name="username" required />
```

However, this may depend on the environment or from tool to tool. For WebdriverIO to successfully identify a React Native or Expo application element, we would need to use the React Native accessibility labels:

```
<TextInput
  accessibilityLabel="Login_Input_Username"
  placeholder="Username"
  value="username" />
```

For Kotlin, we would need to assign `contentDescription` label to the element:

```
<EditText
    android:contentDescription="Login_Input_Username"
    android:id="@+id/Login_Input_Username"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
/>
```

All the implementations source code are open and available at

- `https://github.com/jose-donato/savery`

Table 4.3 presents a brief summary of the development tools selected.

Table 4.3: Summary of the development tools selected to implement the applications

Application Type	**Development Tool**	**Relevance**
Web	React.js	Most popular tool to produce SPA
	Preact	vs React.js (Normal React vs Low overhead)
	Next.js	Becoming the standard to build performant SPA
	Gatsby	vs Next.js
	Svelte	vs React.js (Real vs Virtual DOM)
Hybrid	Ionic	Popular tool to produce hybrid apps
Native	Kotlin	Reference tool to build native Android apps
	React Native	Popular tool to build cross-platform apps
	Expo	vs React Native (abstractions vs no abstractions)

4.1.3 Metrics

As proposed in our framework, we selected two different types of metrics: static and runtime metrics.

Static Metrics

For the static metrics, the benchmark includes application size, line of codes, dependencies number (and dependencies number only for development), and build times. Except for application size, the static metrics help to assess the developer experience of each development tool.

Although the device's storage has increased significantly in recent years, in some low-end and even medium-end devices, small **Application size** is still crucial, and it was

measured in KB. **Lines of code** (or LOC) can help to estimate the maintainability of the software produced. **Dependencies number** is the number of packages that will be used in the production build of the application. As this number increases, the final package will also increase. **Development Dependencies number** is the number of dependencies that are used during development. **Build Time** is the duration (in milliseconds) that each development tool takes to build the production version of the application. Some deployment platforms charge more depending on the build duration. Hence, it is important to keep this value as minimal as possible.

Runtime Metrics

For the runtime metrics, the benchmark contains three metrics: response times, CPU usage, and RAM consumption. These runtime metrics are the most important type to assess the performance of the applications, and we inspired by the following great study [3]. We also believe that they are the best metrics to assess the performance of this type of applications (i.e., applications for mobile devices).

CPU usage is the percentage of CPU used by each application during a certain interval [3]. "CPU intensive applications may negatively impact other processes running on the device, decreasing user experience" [3]. An higher CPU usages will have an impact on the battery duration.

RAM consumption is the amount of memory allocated by the system in MB. Devices can see their performance "degraded if a high percentage of the available RAM is allocated" [3].

Response Times are the most important metrics to mobile application users. This is the total duration in ms that some action took to happen. Studies show that "53% of mobile users abandon sites that take over 3 seconds to load" [73]. If an application response times are high, it will degrade the user experience and may result in the user never using the application again.

Once measured, all the metrics presented have one thing in common: the smaller their value, the better. In Section 4.3.4 we explained how each metric was collected during the Benchmark Campaign. Table 4.4 presents a summary of the selected metrics.

Table 4.4: Summary of the selected metrics

Type	Name	Unit
Static	Application size	-
	Lines of code	-
	Dependencies number	-
	Dev dependencies number	-
	Build time	milliseconds
Runtime	CPU usage	%
	RAM consumption	MB
	Response times	milliseconds

4.2 Development of Applications

In the second phase, we developed 9 applications following the previously defined Reference App Specification, each one using one of the different selected development tools. The development of these applications took an estimated duration of `150-200` hours, i.e. approximately `1-1.5 persons*month`.

The rest of the section is structured as follows. In Section 4.2.1 describes the process of building the applications. Section 4.2.2 explains how we used functional testing to validate the applications. Section 4.2.3 outlines the tools that we developed to aid the campaign and validation process.

4.2.1 Implementing the applications

All the applications were developed in-house. Although this might introduce some bias in the evaluation, it will be mitigated when the project is open to the community, and engineers are able to improve or fix the existing implementations. This will also make possible the addition of other implementations relative to new development tools.

Styling Process

To style the web and hybrid applications, we used a styling solution named tailwindcss for all web applications and Ionic. With this approach, we reduce the time wasted styling the applications (because all the styling is reused) and reduce the possible bugs when implementing this process in each application. Since tailwindcss is not supported on native applications (at least officially), we opt to not use any additional library to style the elements on those applications: for the React Native and Expo applications (native), we used the default styling solutions (through the style React Native prop); for the Kotlin implementation (also native), we used the popular Material Design library from Google.

Configuration Details for Development Tools

We used Craco to configure and create our React.js application. Although create-react-app is the most common tool to build React.js applications, it does not allow many custom configurations. For example, at the time of writing create-react-app did not allow customizing the service worker (and this was needed to implement our reference app specification).

For the react-based implementations (i.e., react.js, next.js, gatsby, preact) we shared a lot of code since there are a lot of similarities between these development tools, and we tried to share as much react components as possible to introduce the less amount of bugs.

To implement the svelte application, we used SvelteKit. Even though SvelteKit was at an early stage at the time of writing, this tool is officially supported by the team behind Svelte, and we did not encounter any barriers when using it.

We built the Ionic application after the React.js app. We started with this application and added the needed configurations to make it a valid Ionic project. Ionic documentation was

followed to achieve this [74].

To build the React Native application, we did not use any Expo-related libraries. Although these libraries are usually good abstractions and fit well with Expo projects, we tried to develop the React Native application without anything related to Expo to better assess the implications that it may have.

On the other hand, in the Expo application, we used several libraries provided by Expo ecosystem which are easy to install in an Expo project and do not require any configurations in Android Native files.

The other applications (i.e., the applications built with Next.js, preact, and Gatsby) were built using their official command-line helpers.

Application Details

Regarding authentication, the implementations present a login screen that contains two fields for the username and password. Once the user types in these fields and clicks the submit button, an HTTP will be made to the backend login endpoint. Then, the server will check whether the submitted credentials are correct or not. If so, a unique token (a JSON Web Token or JWT, more on this) will be retrieved. In the native and hybrid implementations, this unique token is saved in an encrypted store in the device for subsequent accesses to the application to avoid the need of logging in again when the application is restarted. Nowadays, JWT authentication is very common, and this token must be stored securely because an attacker can impersonate a victim in case he gets his hands on the victim's token. In the web applications, we stored the token in the browser local storage, which is unencrypted. Although in most scenarios this is a bad practice because browsers' local storage is not encrypted, we opt to do it to avoid over complexing the web applications compared to the others. More details on how to achieve secure authentication systems on web applications can be found in[75].

For the operation, carousel, or feed component, where there was a need for adding several elements to the screen, we could have used specific libraries that virtualize the elements (i.e., only render them when they are visible on the screen) to make it more performant. However, we opt not to do it in the web applications and keep the applications as simple and with the least number of third-party libraries. As we will see in the following Chapter, the web applications behaved well without these solutions. On the other hand, in the validation phase, the native applications failed to pass the tests where a lot of elements are added to the screen (e.g., in the Operation component) and the applications crashed. A GitHub issue confirmed our assumption that rendering numerous elements can result in sluggish and even crash a React Native application [76]. Therefore, this use case needs to be accounted for and implemented carefully. React Native team published resources on how to solve this problem [77]. To fix this, we virtualized the elements using `FlatList` on React Native/Expo applications and using `RecyclerView` on Kotlin.

In the future, there is a possibility to add new application versions to the benchmark to compare how these virtualized solutions impact the performance of the web applications.

Screenshots of one of the implementations can be found in Appendix B and a live imple-

mentation can be found in the following URL:

- `nextjs.vitamin-server.dei.uc.pt`

Due to time-constraints and some possible lack of knowledge about each development tool specific from us, we do not exclude the fact that some implementations may have some problems and not be fully optimized. To mitigate this we tested all the applications extensively with functional testing in order to find issues, fix them and make sure that all the applications behave similarly (i.e., follow the previously defined App Reference Spec). Of course, functional testing cannot understand if the implementations are fully optimized. Therefore, we will also open the project to the community. This way, the community can both fix possible issues with the existing implementations and introduce new implementations using different development tools.

In the Section 4.2.2, we will go through how the applications were validated with the help of functional testing to prevent that applications with issues proceed to the campaign phase.

Backend Details

Since our reference application spec defined some components that needed external services (e.g., retrieving posts, authentication with username and password), we developed a simple auxiliary backend with fastify. This backend serves two different endpoints (defined in Sec. 4.1.1) that our implementations will then consume for authentication process and the feed component. When these endpoints are called, the backend communicates with a database with an Object-Relational Mapping (ORM) called Prisma and retrieves the desired data to the implementations. The database contains two tables: i) users table: used to store several users (username and password); ii) news table: used to store several posts that contain a title, image, description, date, URL.

We tried to keep the backend service as simple as possible since it is not the focus of the benchmark.

4.2.2 Functional Testing

The functional testing comprises a set of tests to check whether the applications are correctly implemented or not. The tests that we defined try to emulate what an end-user of our application would do when navigating through all of its components.

The functional tests navigate through all the application pages in a certain order without any human-interaction. They check if all application elements have the correct identifiers and that each element works as expected (application elements and their requirements listed in Appendix A). A high-level description of the flow followed to execute the functional tests is presented below:

1. Visit landing page and check if all landing page elements are presented;
    (a) Click button to request permissions;

   (b) Accept all the required permissions.
   (c) Press login link to be redirected to the login page;
2. Visit login page and check if all login page elements are presented;
   (a) Insert correct login credentials;
   (b) Press login button to be redirected to the home page;
3. Visit homepage and check if all homepage elements are presented;
4. Open navbar to check if all the page links are presented;
5. Visit each component by accessing them in the navbar;
   (a) Check if each component elements are presented and if they work as defined in the Reference App Specification;
6. After all the components are visited, open navbar, press logout button and check if the user is redirected to the landing page.

To implement these tests we developed a **Testing and Measurement Tool** that uses Appium [78] and WebdriverIO [72] to automate this process (more details in Section 4.2.3).

All the source code to implement the functional tests is available at the following URL:

- `https://github.com/jose-donato/savery`

If all the tests pass, the application under validation is considered valid and can proceed to the campaign phase.

4.2.3 Auxiliary tools

In this section, we will explain the tools that were developed to support the benchmark.

Testing and Measurement Tool

To support the validation and campaign phases, we built a tool called **Testing and Measurement Tool**. This tool is responsible for creating and submitting the workload to the applications that will be running in the target device. Although in the validation phase, measurements are not gathered, this tool also implements this process. This tool is used in `Preparation Phase` to validate the applications with functional testing and in `Campaign Phase` to execute the benchmark and gather the measurements.

An interesting challenge posed in front of ourselves because in our benchmark we are focusing on several types of applications: web, hybrid and native. We needed to select technologies that helped to test any of these applications. Given this, we only had two options:

- Use different technologies: one for testing each application type.

- Use technologies compatible with all application types.

The first option would result in redundant tests, since there was a need to implement the same set of tests multiple times: one for each environment. This would increase the complexity of the benchmark and would be more time-consuming. Hence, we decided that we were going to opt for the first option. But another challenge appeared: as it happens for development tools, there are countless libraries to build automation tests.

To solve this, we analyzed the most popular automation libraries and compared them. First, we defined several requirements that the ideal automation library should meet:

- **Support different popular mobile operating systems** (i.e., Android and iOS). Although in this benchmark we focused only in one Android device, we developed this tool with the vision that it would be re-used for another rounds that may support iOS in the future.
- **Share tests between multiple application types**, i.e., the tests can be-used to target different types of applications.
- **Support writing the tests in JavaScript/TypeScript**. JavaScript remains programming language most used through the last three years [79], and does not seem that its adoption will stop anytime soon. Since this project will be open to the community, write the tool in this language is the natural option. Also, it is the programming language we are the most familiar with.
- Should be **free** because the benchmark campaign includes dozens of executions for each application.
- Should be **Open Source**. Open technologies are prone to more scrutiny than closed ones. We are focusing on technologies that have been extensively used and scrutinized by the community. We are also focusing on technologies that are currently maintained due to the rapid evolution of the systems under test (applications for mobile devices).
- Must be **highly customizable** (e.g., support scroll operations, verify that items are displayed, clicking in elements or in screen positions).
- Should **not be cloud-based**. This normally means that we have less control and, hence, less customization is possible.
- **Cannot be intrusive**. The tool should treat the applications as black-boxes and not insert custom code in the application under test.

Table 4.5 presents the results of our extensive analysis from multiple sources [80, 81, 82, 83, 84, 85, 86, 87, 88].

We found three main candidates that satisfy the previously-mentioned requirements and can be used to build the Testing and Measurement Tool: Appium, Selenium, Selendroid, and TestProject.

We end up choosing Appium over the others because of several reasons: i) Appium supports multiple platforms (e.g., Android, iOS) as opposed to Selenium that only supports

Table 4.5: Comparison of Popular Automation and Testing Libraries

Name	Open Source	Active	Cross-platform	Cloud-based	Language Drivers	Agnostic	Price	Customization
Appium	Yes	Yes	All mobile apps	No	Ruby,Java,JavaScript, PHP,Python	Yes	Free	High
Selenium	Yes	Yes	Only web apps	No	Java,C#,Perl,Python, JavaScript,PHP,Ruby	Yes	Free	High
monkey-runner	Yes	Yes	Only native android apps	No	Java/Kotlin	Yes	Free	Low
Firebase Test Lab	No	-	Yes	-	-	Yes	Freemium	Low
Robotium	Yes	No	Native and hybrid Android apps	No	Java	No	Free	Medium
Calabash	Yes	Yes	Native/Hybrid iOS and Android apps	No	Cucumber (natural language)	Yes	Medium	
Cypress	Yes	Yes	Only web applications on desktop environments	No	JavaScript	Yes	Free	High
Kobiton	No	-	All mobile apps	Yes	Multiple	Yes	Paid	High
XCTest	Yes	Yes	Only iOS native apps	No	Swift	No	Free	Medium
TestProject	Yes	Yes	All apps for multiple platforms	Yes	Multiple	Yes	Free	Medium
Apptim	no	no	no	no	no	no	no	no
Perfecto	No	-	All apps for multiple platforms	Yes	Multiple	Yes	Paid	High
Selendroid	Yes	No	All Android apps	No	Multiple	Yes	Free	High
KIF	Yes	Yes	Native iOS apps	No	Swift	Yes	Free	Low
Katalon	no	no	All apps for multiple devices	Yes	no	Yes	Freemium	no
TestComplete	No	-	Several apps for multiple devices	Yes	-	-	Paid	High
ios-driver	Yes	No	All iOS apps (web, hybrid native)	No	-	Yes	Free	High
21labs	No	-	All native Android and iOS apps	Yes	DND	Yes	Freemium	Low
Xamarin.UITest	No	-	Native iOS and Android apps	No	C#	No	Free	Medium
Espresso	Yes	-	Android native apps	No	Java/Kotlin	No	Free	Medium
Ui Automator	-	-	Android native apps	No	Java/Kotlin	Yes	Free	High
BrowserStack	No	-	All apps for multiple platforms	Yes	Multiple	yes	Paid	High

web environment, and although we are only targeting Android in our benchmark, our framework can be easily extended to more platforms and future benchmarks may include other devices; ii) Appium can automate tests for multiple environments (e.g., web, native, and hybrid); iii) Appium allows us to write the tests with WebdriverIO, a library that has Node.js bindings (a language we are very familiar with) and re-use these tests across multiple environments; iv) Selendroid development ceased activity while Appium is maintained currently maintained; v) TestProject is not used by the community as much as Appium (TestProject JavaScript Sdk [1] has only 48 weekly downloads on NPM while WebdriverIO, Appium JavaScript driver, has 861,348 weekly downloads at the time of writing [2]); vi) Appium has a client-server architecture that fits perfectly with the Testing and Measurement Tool.

It is important to note that Appium abstracts other libraries that appear in Table 4.5 (e.g., UiAutomator to communicate with Android applications, XCUITest to communicate with iOS applications). This exactly what we were looking for and makes possible that from the same set of tests we can automate different types of applications (i.e., native, hybrid and web) in different types of devices (e.g., Android, iOS).

Appium has a server-client architecture that fits exactly our requirements. We developed a client called Testing and Measurement Tool (or TMT) in TypeScript. TMT contains

[1] https://www.npmjs.com/package/@tpio/javascript-opensdk
[2] https://www.npmjs.com/package/webdriverio

several tests built with the help of WebdriverIO. WebdriverIO has JavaScript bindings and allowed us to build the tests using this language that we are used to. During the tests' execution, they are sent to Appium server, which is responsible for forwarding them to the devices under test and receiving the results (i.e., reporting the test results back to the Testing and Measurement Tool). In addition to WebDriverIO, we used Mocha and Chai to help structure our tests.

Since our objective is to make all the contributions of this study open-source (including the Testing and Measurement Tool), we implemented the tool using TypeScript [89], a superset of JavaScript, designed to be more scalable and easier to maintain.

The functional tests simulate several human-like operations. In the following list, we enumerate some of these operations and code snippets on how we achieved them:

1. Clicking buttons: in the following example, we can observe the necessary code to click a button. The code is straightforward, `grabber` is an asynchronous function that receives the application type (e.g., web, hybrid or native), the WebDriverIO client and the unique identifier that we want to interact with. Then, we wait for WebdriverIO to find this element (if timeout exceeds this test will fail), finally WebdriverIO client will send the instruction to click this button.

```
async function () {
    const Login_Button_Submit = await grabber(appType, webdriverIOClient,
        "Login_Button_Submit")
    await Login_Button_Submit.waitForExist({ timeout: timeout })
    await Login_Button_Submit.click()
}
```

2. Submitting text to input fields: the approach to input fields is similar, we first select the input fields we want to type and then `setValue` function provided by WebdriverIO is used to send the text to the input fields. *hideKeyboard* is another function provided by WebdriverIO that can be used to hide the device keyboard after typing in the input fields.

```
async function () {
    const Login_Input_Username = await grabber(appType, webdriverIOClient,
        "Login_Input_Username")
    await Login_Input_Username.waitForExist({ timeout: timeout })
    const Login_Input_Password = await grabber(appType, webdriverIOClient,
        "Login_Input_Password")
    await Login_Input_Password.waitForExist({ timeout: timeout })
    await Login_Input_Username.setValue("user1")
    await Login_Input_Password.setValue("pass1")
    await webdriverIOClient.hideKeyboard()
}
```

3. Scrolling through lists: to perform the scroll operation, we need to define an array with different operations. If we want to perform a vertical scroll, we simulate the presses that a user would do in real a device. First, we simulate the finger pressing in bottom of the screen (y: 1500), then wait and move the finger to the top of the screen (y: 10). Finally, the finger is released with the "release" action. `touchPerform` WebdriverIO function is then called with this array to perform the scroll operation.

If we want to run the scroll multiple times, we can wrap this function in a for loop. In the following example, the scroll operation will be performed three times with 3.5 seconds of interval.

```
async function () {
    const scroll = [{
        action: 'press',
        options: {
            x: 0,
            y: 1500
        },
    },
    {
        action: 'wait', options: {
            ms: 300,
        }
    },
    {
        action: 'moveTo',
        options: {
            x: 0,
            y: 10
        },
    }, {
        action: 'release',
    }]

    for (let i = 0; i < 3; i++) {
        await webdriverIOClient.touchPerform(scroll);
        await webdriverIOClient.pause(3500)
    }
}
```

4. Open notifications bar: opening notifications bar is easily achieved with WebdriverIO. First we can its function `openNotifications`, then *pause* will wait for 2 seconds and, finally, calling `pressKeyCode` with 4 as an argument (Android Key Code to click the back button) will close the notifications bar and return to the application.

```
async function () {
    await webdriverIOClient.openNotifications()
    await webdriverIOClient.pause(2000)
    await webdriverIOClient.pressKeyCode(4)
}
```

In the `Preparation Phase`, this tool is only responsible for validating each application components.

However, in the `Campaign Phase`, TMT has some more responsibilities. While the tests are running, this tool also sends commands via the Android Debug Bridge (or ADB) periodically to the connected Android device under test to measure the CPU usage and RAM consumption. All these measurements and results are then saved into JSON and SQLite files for further analysis. We followed the proposed folder structure by our framework in Section 3.4 to organize the outputs of the Benchmark Campaign.

In more detail, while the automation tests are being sent to the target device in the main thread, and, we used a library called threads.js to spawn another thread responsible for sending the ADB commands referred in Section 4.3.4 to gather the measurements with an interval of 200 milliseconds.

There are two different ways to interact with the Testing and Measurement Tool:

- **Command-line interface**: to facilitate the usage of the tool, we developed a single command-line interface. Although it is currently more limited, it eases the process of trying the tool. Fig. 4.5 presents an image of this interface.
- **Configuration file**: allows more customization of what the tool should do. In addition to all the options presented in the command-line interface, in the configuration file, we can define: the order in which the components are tested, if the device should reboot after each execution, customize the interval between each measurement gathering, which components to test, among others. Appendix C presents this configuration file in more detail.

Figure 4.5: Testing and Measurement Tool command-line interface

The Testing and Measurement Tool needs to be running on a device with a valid Android SDK installation to be compatible with Appium (more details on the requirements in the project repository).

Although we did not include iOS devices in our experiment, the stack that we used to build the Testing and Measurement Tool turns possible to support such devices in the future. However, some additional work must be done since the current measurements gathering process uses ADB, which is unavailable in iOS devices.

Analysis Tool

We also developed a simple web application that analyzs the results from the benchmark campaign and generates tables and charts for further analysis. We used `Next.js`, `better_sqlite3` [3] to read the benchmark campaign results, `danfo-js` [4] to process and structure this data and `apexcharts` [5] to generate the charts. Some charts presented in Chapter 5 came from the analysis tool (e.g., Fig. 5.1).

[3] https://github.com/JoshuaWise/better-sqlite3
[4] https://danfo.jsdata.org/
[5] https://apexcharts.com/

The source code for this tool is in the following URL:

- `https://github.com/jose-donato/savery`

4.3 Benchmark Campaign

In the third phase, after a set of configurations are respected, the Measurement and Testing Tool presented in the previous section sends three different workloads. At the same time, the tool collects the previously defined metrics. We will start by explaining how the setup for the benchmark campaign is structured. Then, we will through the configurations required for each campaign and, finally, how the measurements were gathered.

4.3.1 Setup

The campaign setup includes three main devices:

- **Device running Testing and Measurement Tool**: this device is responsible for running Appium and the TMT. After the connection between the target device and this one is established, and the configurations are respected (more on configurations in Section 4.3.2), the benchmark campaign is ready to start. In this experiment, the device was a virtual machine running Lubuntu, a lightweight Linux distro. The device has a valid Android SDK installation which is required in order to use ADB. ADB is the technology used gather the measurements in the Android device (more on this in Section 4.3.4).
- **Target device**: where the applications under test will run. This device must be connected to the TMT via USB. In our experiment, we used a medium-end Android device from 2016 (Honor 8). The device has the following specs: 4 GB RAM; a medium-end CPU (an Octa-core with 4x2.3 GHz Cortex-A72 & 4x1.8 GHz Cortex A53); GPU Mali-T880 MP4; and Android version 7.
- **Remote server**: serves the applications and the backend that the applications may require. This server was deployed on a machine with 8 GB of RAM and high-speed network transfers to prevent it from becoming a bottleneck of the benchmark. All the implementations and the backend were deployed with the help of Docker [90] for isolation between the applications and Nginx Proxy Manager [91]. For each web application, a Nginx web server was used to serve the respective production build files. For the native and hybrid applications, another Nginx web server was used to serve the android APKs. The target device and TMT will make requests to this server during the campaign. The backend server is also running on this server.

A campaign follows the flow described in Fig. 4.6 and is enumerated below:

1. Testing and Measurement Tool (TMT) initiates the benchmark by requesting the application to the remote server.

2. TMT receives the application (e.g., `next.js` application) and starts the connection to the target device that will run the application requested.

3. TMT connects to the target device and sends the application to the target device. If it is a web application, target device will open it in the Chrome browser. Otherwise, it will install the application (i.e., the Android device will install the native APK). Once the application is loaded and started, TMT will start sending the workload (workloads defined in Section 4.3.3). This workload contains a Warm-up period to prepare the application (e.g., accept the required permissions). After the Warm-up period finishes, the TMT will send the main tests and start querying periodically the runtime metrics (i.e., CPU usage, RAM consumption and response times). TMT saves the results to a SQLite database as it receives these values.

4. During the tests' execution, target device may communicate with the backend that is being served also in the remote server (e.g., to the login process, or to fetch some content from the database).

5. TMT finishes sending the workload, saves the results, closes the connection to the target device, and cleans its usage (cleaning the Chrome browser data in case the application was a web application, uninstalling the application otherwise).

The benchmark campaign must run a considerable amount of times for each application to reduce/eliminate the effect of random errors. As we will see in the Section 4.3.3, the campaign was executed 60 times in each application with two different workloads. The experimental procedure followed is illustrated in Fig. 4.7. As we can observe, each application must be executed several times and each execution contains three different phases that happen sequentially:

1. **Initialization**: Application is installed and opened in the device.

2. **Workload**: The tests are executed (more details on the workloads in Section 4.3.3).

3. **Cleanup**: Application is closed and uninstalled from the device (in case of web applications the browser data is cleaned).

In the following section, we will see the set of configurations that must always be respected before each execution.

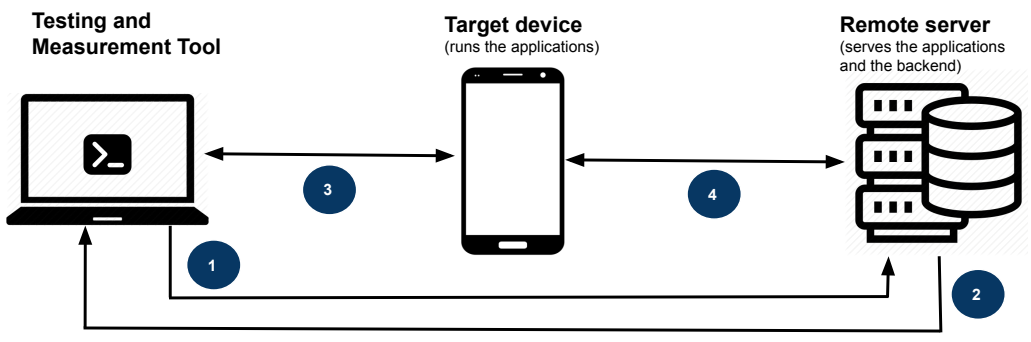


Figure 4.6: Proposed Benchmark Campaign flow

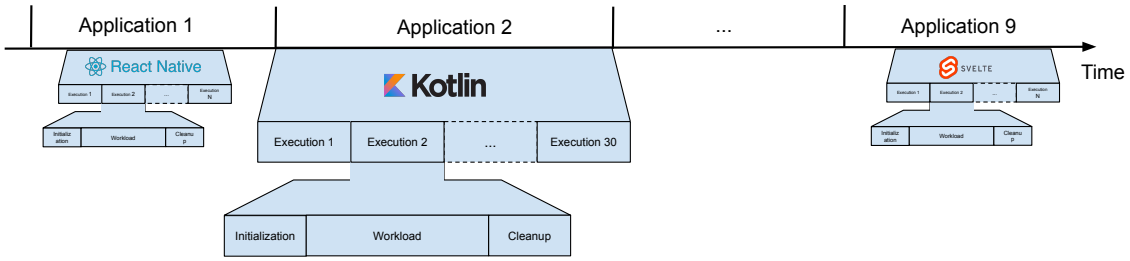


Figure 4.7: Experimental procedure for the benchmarking campaign

4.3.2 Configurations

Before starting each execution, the following configurations must be always respected:

- All the web applications and the backend server must be served with HTTPS. Nginx Proxy Manager [91] facilitates this process.
- Applications must be isolated between each other in the remote server to have fair comparisons.
- Permissions are not granted to any application to emulate what happens with real-world. The workload contains a Warm-up period that accepts the required permissions with automated functional tests.
- Target device must be rebooted before each execution to guarantee that the RAM is released, and the device is cleaned. Also, after the device is rebooted we must until the CPU usage and RAM consumption are close to 0%.
- The target device cannot be running any other applications.
- Must have recent image downloaded.
- Target device must have a valid and fast internet connection.
- An image must be recently downloaded because the File Access component requires a recently downloaded image in the file system.
- After the execution finishes, the application must be fully removed. Appium capability `fullReset` was used to achieve this [6].

4.3.3 Workload

The benchmark workload follows the reference app specification previously defined. The main objective of the workload is to go through the application and test all the components defined in Section 4.1.1. This process is fully automated, i.e., does not require any human interaction satisfying the reproducible benchmark requirement.

For our workload we have defined two groups of tests:

[6] Appium capabilities: https://appium.io/docs/en/writing-running-appium/caps/

- **Complete action tests**: all application components are tested subsequently with a small interval in between.

- **Single-action tests**: perform single-actions in the application, i.e., visit only one component of the application. Sometimes, users do not open applications and use all their features but only one or a few.

Before each group of tests starts, a Warm-up Period is required. Once each group starts, the tests will go through the application components and assess their functionalities. It is also at this moment that It is important to note that the functional tests associated with each component are both the same in Single-action or in Complete action test groups. In total there are 49 component functional tests, and they are listed in Table 4.7.

Warm-up Period

Warm-up Period is responsible for initializing and preparing the application for the subsequent tests. It starts by accepting the permissions needed (e.g., camera, geolocation) in the landing page, then proceeds to log in and visit each page without testing its functionalities just to cache its contents in order for all the applications to have the same starting point (i.e., web, native and hybrid apps). After visiting each component, this process finishes when the logout button is clicked and the application is at the Landing page again. At this point, the application is ready to start the primary tests (either the Individual or the Complete group of tests). The list automated tests from Warm-up period is presented in Table 4.6.

Table 4.6: Automation test suite for warm-up period.

https://github.com/jose-donato/savery/blob/main/tmt/src/tests/pages/warmup.page.ts

ID	Description
t-warmup-landing-0	Landing page is presented
t-warmup-landing-2	Click ask permissions button; Accept permissions (2 in mobile/hybrid applications, 3 in web applications); Button should be replaced with text "Permissions granted"
t-warmup-landing-2	Clicking login button should redirect to Login page
t-warmup-login-1	Login page should be presented
t-warmup-login-2	Insert username and password in respective input fields
t-warmup-login-3	Clicking submit button should redirect to Home page
t-warmup-bgSync-0	Navigate to Background Sync component
t-warmup-fileaccess-0	Navigate to File Access component
t-warmup-fileaccess-1	Click upload image button; Accept storage permission in web applications (native applications do not require this permission);
t-warmup-feed-0	Navigate to Feed component
t-warmup-geolocation-0	Navigate to Geolocation component
t-warmup-notifications-0	Navigate to notifications component
t-warmup-camera-0	Navigate to Camera component
t-warmup-carousel-0	Navigate to Carousel component
t-warmup-operation-0	Navigate to Operation component
t-warmup-logout-0	Clicking logout button should redirect to Landing page

Complete Action Tests

In this group of tests, it is important to define the sequence that the automated tests must follow, i.e., the order of visiting the components and testing its functionalities. Since some components are more expensive than others (e.g., Operation component renders a lot of elements to the screen whereas Geolocation only renders one-paragraph of text), they may influence the subsequent components. But it is important to test the application as a whole because there are users that may use all the features from one application subsequently. To mitigate this, we have done two things: i) defined two different orders for this group; ii) defined another group of tests (Single-action tests) that aim to test the components individually and isolate them.

The two orders we have defined to test the components are the following:

`01` Landing →Login →Homepage →Operation →Camera →Carousel →Notifications →Feed →Geolocation →File Access →Background Sync →Logout

`02` Landing →Login →Homepage →Background Sync →File Access →Feed →Geolocation →Notification →Camera →Carousel →Operation →Logout

The flow for the Complete-action tests is the following:

1. Install (in case of native and hybrid applications) and open application under test (handled by Appium);
2. Run `Warm-up` automated tests;
3. Run only one of `01` or `02` automated tests;
4. Uninstall application under test (in case of native and hybrid applications) and clean data (handled by Appium);

Single-action Tests

The Single-action Tests try to assess each authenticated component individually. Instead of testing all the application components subsequently, Single-action only tests a feature in each execution. The order to run the components is arbitrary, since the application is removed and cleaned after each execution (an in between each component). Therefore, the order should not matter in this scenario.

1. for `currentComponent` in `listOfComponents`: (i.e., loop through list of authenticated components)
   (a) Install (in case of native and hybrid applications) and open application under test (handled by Appium)
   (b) Run `Warm-up` automated tests (since we are visiting each component individually, the warm-up does not need to visit each application authenticated component but only the one under test)

(c) Log in

(d) Run `currentComponent` automated tests

(e) Logout

(f) Uninstall application under test (in case of native and hybrid applications) and clean data (handled by Appium)

The Table 4.7 enumerates all the automation tests: their IDs and a brief description of the tasks performed in each test. The tests that only wait an interval to complete are presented to increase the representativeness of the workload. Focusing on one example, `t-camera-2` waits for 2 seconds when the camera feed is displayed. Without this test, the execution would proceed instantly to the next test after the camera feed is rendered: taking a picture from the camera feed. For example, if this application was Snapchat or Instagram, the user would not click instantly to capture a photo. First, would pose for the camera, check if everything was fine and only after that would capture the photo. Hence, the interval aims to achieve similar effect.

Unfortunately, due to time-constraints, we only executed the Complete Action Tests in our experiment. We executed 30 times each order, i.e., we executed 30 times Complete Action Tests with the `O1` plus 30 times Complete Action Tests with `O2`. Chapter 5 presents and discusses the results of the 60 executions.

4.3.4 Measurements Gathering

To measure the **size** of web applications, we used the service lighthouse-metrics.com which provides the total transferred resources size. For native and hybrid applications, this value is just the size of the final Android Application Pack (or APK). The `lines of code` were measured with the help of cloc [92], a great tool to count physical lines of source code in many programming languages (supported Svelte as opposed to other tools that we analyzed). The `dependencies number` was calculated by analyzing the `package.json` file. To compute the `build times`, we used the `time` command from Unix. For native and hybrid applications, an additional step was needed. Since these applications need to be built using Android Studio, this duration also needed to be accounted for. Considering Expo uses their cloud-based system to build the APKs, the duration did come from this system instead of Android Studio.

Runtime metrics are trickier to collect, and to respect the framework requirements, we developed a solution that was non-intrusive and had minimal impact on the benchmark.

To collect the **CPU usage**, our Measurement and Testing Tool sends the `adb top` command every 200 ms during the execution of the tests via USB to the Android device. This command returns the CPU Usage of the package provided in percentage:

```
adb shell top -n 1 | grep 'packageName' | head -1 | awk '{print $5;}'
```

After analyzing the duration of each test and concluded that 200 would be small enough to intersect each test and provide the CPU usage during the execution of the test.In web applications, the package considered is the mobile browser since it is where this type of applications is running (e.g. Google Chrome, hence the package was `com.android.chrome`).

Table 4.7: Automation Tests Suites per Component

https://github.com/jose-donato/savery/blob/main/tmt/src/tests

Page	**ID**	**Description**
Landing	t-landing-0	Landing page should be presented when application launches
	t-landing-1	Clicking ask permissions button should render text "Permissions Granted"
	t-landing-2	Clicking login button should redirect to Login page
Login	t-login-1	Login page should be presented
	t-login-2	Insert username and password in respective input fields
	t-login-3	Clicking submit button should redirect to Home page
Home	t-home-1	Home page should be presented
	t-home-2	Table with available components should render below
	t-home-3	Paragraph on top should say "All permissions granted"
	t-home-4	Toggling menu button should open application Navigation Bar
	t-home-5	Toggling menu button again should close application Navigation Bar
Background Sync	t-bgSync-0	Navigate to Background Sync component
	t-bgSync-1	Top paragraph should say user is online
	t-bgSync-2	Toggle Wi-Fi; Wait until top paragraph says user is offline
	t-bgSync-3	Click get image from cache button should render image from cache below
	t-bgSync-4	Wait 2 seconds to observe cached image
	t-bgSync-5	Toggle Wi-Fi; Wait until top paragraph says user is online
	t-bgSync-6	Click get image from web button should render image from web below
	t-bgSync-7	Wait 2 seconds to observe random image
Camera	t-camera-0	Navigate to Camera component
	t-camera-1	Camera feed is presented
	t-camera-2	Wait 2 seconds to observe camera feed
	t-camera-3	Clicking take photo button should render image from camera feed below
	t-camera-4	Wait 2 seconds to observe image taken
	t-camera-5	Toggling switch button should close camera feed
Carousel	t-carousel-0	Navigate to Carousel component
	t-carousel-1	First image from vertical carousel is presented
	t-carousel-2	Scroll and wait two seconds three times (when bottom is reached more images should be requested and rendered below)
Feed	t-feed-0	Navigate to Feed component
	t-feed-1	10 posts should render when page mounts
	t-feed-2	Change input field to 20
	t-feed-3	Click button to request 20 another posts
	t-feed-4	20 another posts should be loaded below
	t-feed-5	Scroll and wait two seconds two times
File Access	t-fileaccess-0	Navigate to File Access component
	t-fileaccess-1	Click upload button; Select image from native file system
	t-fileaccess-2	Selected image should render below
	t-fileaccess-3	Wait 2 seconds to observe image selected
Geolocation	t-geolocation-0	Navigate to Geolocation component
	t-geolocation-1	Paragraph with user geolocation should be rendered when page mounts
Notifications	t-notifications-0	Navigate to Notifications component
	t-notifications-1	Click local notification button; Open notification bar; Notification should appear; Close notification bar
Operation	t-operation-0	Navigate to Operation component
	t-operation-1	Clicking create small button should render list with 100 rows
	t-operation-2	Clicking clear button should remove list of rows
	t-operation-3	Clicking create big button should render list with 1000 rows
	t-operation-4	Clicking swap rows should swap list of rows
	t-operation-5	Scroll to check rendered rows
Logout	t-logout-0	Clicking logout button should redirect to Landing page

In the native or hybrid application, the package considered is the respective APK package name. All implementations and respective package names are listed in Table 4.8.

Regarding **RAM Consumption**, the process had to be different. We started collecting the memory a similar process to collecting the CPU usage. The only difference was the

Table 4.8: Package names for gathering measurements of each implementation

Type	Name	Package
Web	React.js	com.android.chrome
	Preact	com.android.chrome
	Next.js	com.android.chrome
	Gatsby	com.android.chrome
	Svelte	com.android.chrome
Hybrid	Ionic	io.ionic.starter
Native	Kotlin	com.savery.kotlin
	React Native	com.react_native
	Expo	com.temp.expo

ADB command:

```
adb shell dumpsys meminfo packageName | grep TOTAL | head -1 | awk '{print $2;}'
```

When we first analyzed the memory consumption by the web applications, we observed that they all consumed the same amount of memory: around 100MB. This value is low for this kind of applications and we quickly realized that we were not collecting all the memory that was being used by Chrome. It turns out that the browser uses three different processes: i) `com.android.chrome`; ii) `com.android.chrome:privileged_process0`; iii) `com.android.chrome:sandboxed_process1`. At this point, we were only measuring the first process. We tried to call the `adb dumpsys` command for the three packages but with this approach, the web applications were presenting visible lag and sometimes ended up crashing. We quickly realized system was running out of memory because of all these adb commands (1 for CPU and 3 for RAM) being called every 200 milliseconds. It is important to note that CPU was not a problem since the top command returns all three packages and we could sum the CPU usage of the three processes. On the other hand, the memory ADB command only returned the value for one of the processes. Performing 4 requests every 200 milliseconds for web applications and only 2 requests for other types of applications was completly unfair to the former.

To solve this, instead of measuring the memory consumed by each package, we measured the memory consumed by the whole system. Therefore, we used the following command to gather the memory used by the system in MB every 200 milliseconds.

```
adb shell cat /proc/meminfo
```

Finally, the **response times**, in our study, correspond to the test duration in ms. We opt to not include code to collect these values directly in the applications source code because, as said before, we treated all the applications as a black box. Therefore, we use the duration of each test to compare the different applications in terms of how much time does it take for each application to complete a task. TMT measured this value with the help of the `perf_hooks` module from Node.js [93]. In all the tests, TMT ticks the instant immediately before the test starts and immediately after it ends. The subtraction between these two values provided the test duration in milliseconds.

Chapter 5

Results and Discussion

As explained in the previous chapter, we implemented nine applications with the same set of features by using different popular development tools. Then, with automated functional testing, the implemented applications were validated. Once all the applications were validated, a benchmark campaign was conducted.

This chapter presents the results of 60 executions of Complete-action Tests of each application in two different orders (i.e., 30 executions with `01` and 30 executions with `02`). This chapter is divided into two different parts: i) the first four sections analyze the runtime measurements and tries to assess which applications are **faster** (in terms of response times), and more **efficient** (in terms of CPU and memory usage); ii) Section 5.5 analyzes the static measurements. The goal of static measurements is to provide some insights about the **maintainability** (comparing lines of code, dependencies number, etc.) and **developer experience** (comparing the build times) of each tool.

These experiments have as objectives to **demonstrate the applicability of the proposed framework**, i.e., that it could be used in practice to compare different development tools, and to **analyze and compare the performance of different applications**, and thus understand which development tools is best suited for each use case.

The complete results are included, together with the remaining materials of this work in:

- `https://savery.dei.uc.pt`

Table 5.1 enumerates the identifiers of each application presented in the charts and presents a brief summary of the selected metrics (further detailed in Section 4.1.3).

Table 5.1: List of Application types and identifiers and summary of the selected metrics

Type	Name	Identifier
Hybrid	Ionic	h_ionic
Native	Kotlin	n_kotlin
	React Native	n_react_native
	Expo	n_expo
Web	React.js	w_reactjs
	Preact	w_preact
	Next.js	w_nextjs
	Gatsby	w_gatsby
	Svelte	w_svelte

Type	Name	Unit
Static	Application size	-
	Lines of code	-
	Dependencies number	-
	Dev dependencies number	-
	Build time	milliseconds
Runtime	CPU used by each application package	%
	RAM consumed by the system	MB
	Response times	milliseconds

5.1 Overall results

This section presents an analysis of the results obtained during the benchmarking campaign. We present and analyze the average value of the obtained results in terms of total tests duration (ms), CPU Usage (%), and RAM Consumption (MB). As explained before, each experiment was repeated 60 times for each application, and therefore the results correspond to the average of all the measures gathered throughout the tests. Table 5.2 present an overview of the results, which are discussed below.

Table 5.2: Average (and max) results

App ID	Avg total test duration	Avg duration (Max)	Avg CPU (Max)	Avg RAM (Max)
h_ionic	120708.15	1855.73 (13965.30)	6.93 (90)	2088.24 (2979.58)
n_expo	148148.16	2405.48 (13465.44)	5.81 (89)	1952.24 (2243.23)
n_kotlin	136799.97	2177.46 (16770.84)	5.50 (86)	1943.84 (2621.90)
n_react_native	147009.25	2382.49 (13423.80)	5.65 (103)	1941.76 (2245.21)
w_gatsby	134379.05	2130.14 (13354.86)	9.09 (85)	2108.38 (2958.99)
w_nextjs	130841.10	2057.95 (13378.70)	9.00 (82)	2101.60 (2885.66)
w_preact	132625.58	2092.96 (13371.45)	9.01 (74)	2121.36 (2854.06)
w_reactjs	133216.95	2106.54 (13386.19)	9.16 (92)	2095.66 (2801.26)
w_svelte	137299.84	2188.56 (13739.12)	11.12 (79)	2123.47 (2652.79)

The **average tests' duration** is depicted in Fig. 5.1, Surprisingly Ionic, the hybrid application, presented the fastest average response times with all web applications as close competitors. This was not expected because, typically, cross-platform tools come with the penalty of performance. However, our results showed the opposite for Ionic, which, according to Table 5.2, was 1.133 times faster than Kotlin (a difference of around 16 seconds), a native Android application, to complete all tests from the campaign. Comparing to Next.js, the fastest web application, Ionic was 1.084 times faster. Regarding the web applications, the slowest, on average, to complete all the tests was Svelte, and the fastest was Next.js. However, this difference was around 6.5 seconds. This difference is even lower if we only consider React-based web applications: 3.5 seconds. The slowest React-based application to complete all the tests was Gatsby. Next.js and preact achieving the lowest response times between web applications was expected since both development tools are built with performance as the primary requirement.

This difference is more significant if we compare the native and hybrid applications: around 26.3 seconds on average between Ionic and React Native. We also highlight that the React Native application was only 1.1 seconds faster than Expo, which shows that if we only consider response times, the abstractions introduced by Expo (which bring many advantages) are worth it since the response times are so similar.

The results for **CPU Usage** are depicted in Fig. 5.2, with the numbers available in Table 5.2. As expected, the React-based web applications behaved similarly (again with Next.js, 9.00, and preact, 9.01 as the more efficient), which was expected since they are built on top of React. On the other hand, we can observe that Svelte was far more expensive in terms of CPU consuming over 1.214 times more than React.js, proving that interacting

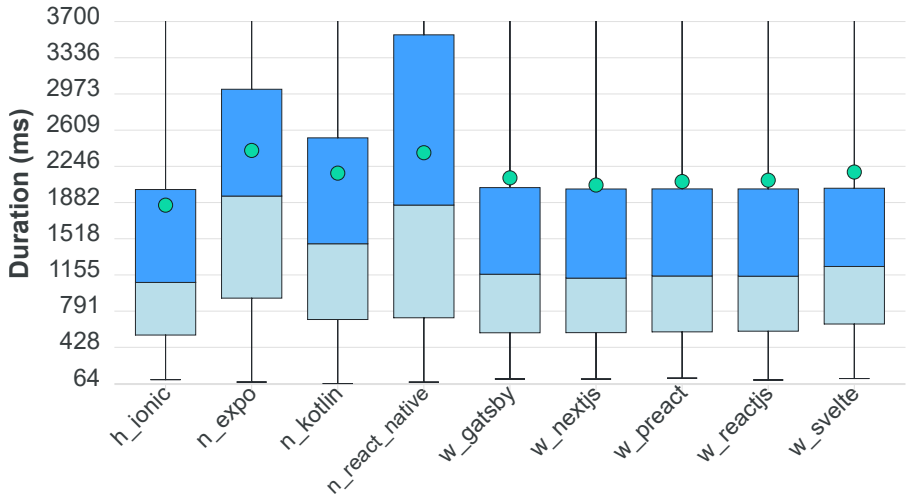


Figure 5.1: Results Tests Duration (ms) per application.

with the Real DOM is an expensive operation, while, React approach of abstracting this process is more efficient.

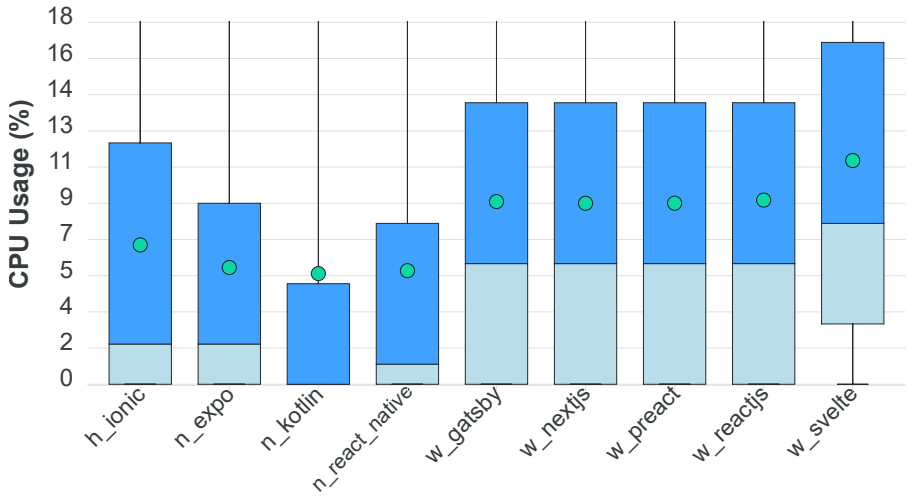


Figure 5.2: Results for CPU Usage (%) per application.

Between the native applications, as expected, Kotlin used the most negligible percentage of CPU and used around 1.260 times fewer than Ionic, the most expensive native tool, and 2.022 times fewer Svelte, the slowest overall. Again, Expo and React Native presented similar results favoring, again, React Native.

Regarding **Memory Consumption**, assuming that the only application consuming relevant memory in the system was the application under test (and we ensured this with our set of configurations, cf. Section 4.3.2), web applications were again the applications that used more resources. When running the web applications tested, the system used similar amounts of memory. Also, close to CPU usage results, Ionic presented similar usages to the web applications, which was expected since this development tool uses web technologies under the hood.

In general, the results showed that Ionic is a fast solution if we can afford to use more

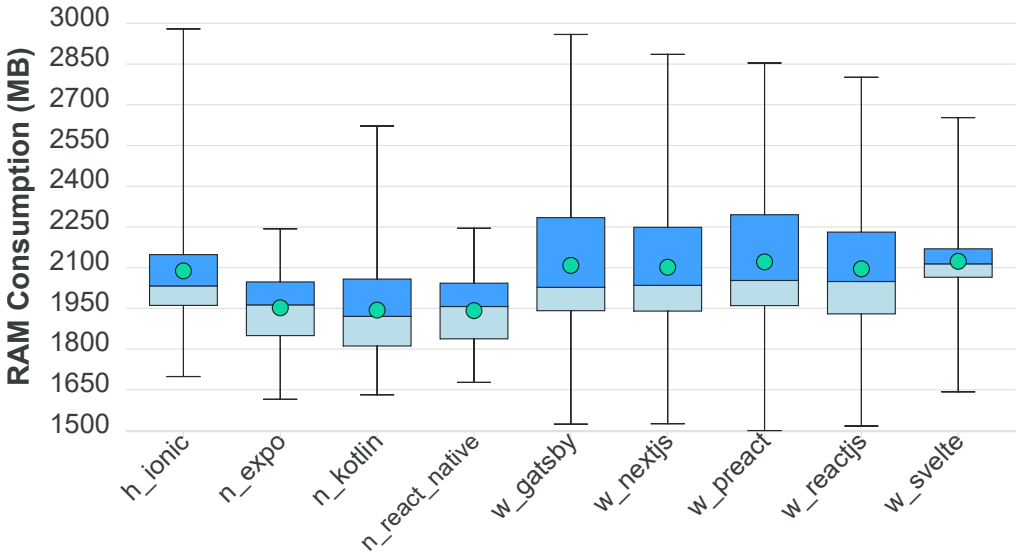


Figure 5.3: Results for RAM Consumption (MB) per application.

CPU and memory. Like Ionic, web applications remain an interesting option, providing fast response time while consuming more resources. If we do not have resource constraints, web applications are a viable alternative to other types of applications. Also, the difference between Expo and React Native is not significant, which means that Expo can be a viable alternative to produce React Native applications with great additions (e.g., producing applications for more platforms than iOS and Android). Finally, for the use cases that our campaign covered, React-based tools are faster and more efficient than Svelte. Between the React-based, Next.js and preact presented the best results in both resources efficiency and response times, but the differences are minimal. In situations where there are few resources, Expo, Kotlin, or React Native should be used as they are more efficient than the competitors.

5.2 Test Duration

This section analyzes the response time (or test duration) results during the benchmarking campaign. We analyze the time (ms) that each application took to complete the tests. Since the experiment was repeated 60 times for each application, the presented duration is an average value of all the executions. Table 4.7 enumerates the list of tests (i.e., their identifiers and descriptions). There are in total 49 tests (presented in Table 4.7), and Fig. 5.4 presents the average time taken to complete each test per application. We primarily focused on the tests covering the scenarios where web applications usually fall behind compared to native ones to answer the dissertation title.

In tests that involve interacting with elements (e.g., typing in an input field) such as `t-login-2` and `t-feed-2`, the applications that use web technologies (web applications and ionic) were faster than React Native, Expo, or Kotlin (native applications). Amongst the applications that use web technologies, we can also observe that Svelte is slightly slower in this use case. This suggests that web applications and Ionic are more viable for situations when interacting with input elements is necessary (e.g., code editors, note-taking apps).

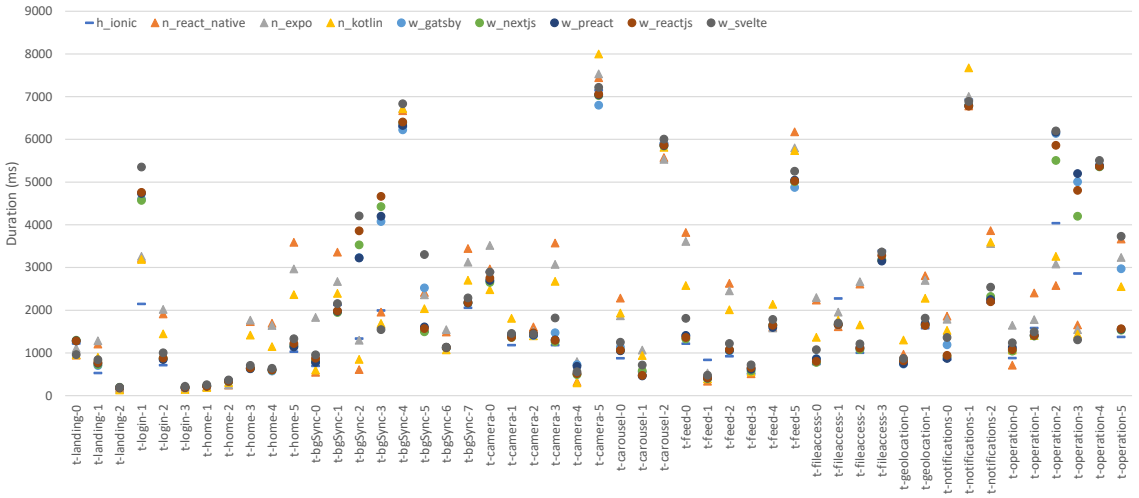


Figure 5.4: Average duration (ms) per test for each application.

Toggling the navigation bar (also called drawer in native environments) occurs in tests `t-home-4` and `t-home-5`. In these tests, web-based technologies presented the fastest results again. This was expected since the other applications add by default an animation that delays the appearance and removal of the drawer.

`t-bgSync-2`, `t-bgSync-3`, and `t-bgSync-5` are three tests that aim to assess the offline capabilities of each application. Although all web applications took more time (around two more seconds) than native and hybrid applications to realize that they had no internet connection (`t-bgSync-2`), some of them (i.e., preact, Next.js and React.js) were faster than all the others to realize that they were connected to the internet again (`t-bgSync-5`). As for retrieving an image from the cache (`t-bgSync-3`), native and hybrid tools performed again way better than all web applications except for Svelte. We can conclude that although the web applications tested can provide offline functionalities (e.g., retrieving content from cache, reacting to the lack of internet), native and hybrid applications are still better for this use case.

Tests that used native features such as Geolocation (`t-geolocation-1`), Camera (`t-camera-1` and `t-camera-3`) or accessing the file system (`t-fileaccess-2`)

Rendering an image from the file system was faster in all web-based applications. Also, similar results are observed when accessing the geolocation of the device. Regarding the camera usage, while accessing its feed was similar in all applications, rendering an image from the camera feed to the screen was also faster in web-based technologies except for Svelte. This shows that web applications (and Ionic) are already a fast alternative to native applications when we need to access native features such as the ones mentioned.

Regarding loading content from outside sources (`t-bgSync-6`, `t-feed-1`, `t-feed-4`), and `t-carousel-1`), all the applications take similar time. We highlight the test `t-feed-1` where Next.js excels between the web-based applications because of Incremental Static Regeneration. This feature provided by Next.js allows the user to be always presented with instant content (more detailed in the article [94]).

Finally, we highlight the operations tests where hundreds of items are added to the screen (`t-operation-1` and `t-operation-4`), swapped (`t-operation-2`) and removed

(`t-operation-3`). It is odd to observe that the web-based technologies behaved similarly to the native applications (Kotlin, React Native, and Expo) in the tests that the elements are added to the screen because, as opposed to the latter, the former implementations did not virtualize the elements (i.e., only render the elements on the screen and loading others only when they are visible). If the results are close while web-based applications do not virtualize the elements, if they use virtualized solutions (e.g., react-virtualized [95] for React-based), we think the results would favor them. The benefits of virtualizing elements are visible on `t-operation-2` and `t-operation-3` when swapping or removing these elements. In these tests, native applications that virtualize the elements presented faster results since they only need to bring to memory the visible elements to the screen. Therefore, virtualizing solutions should always be considered for use cases where elements are added to the screen. In addition, although Ionic does not virtualize the elements, it shows faster results which suggest that it handles these use cases better than web applications.

5.3 CPU Usage

This section presents an analysis of CPU used by the application under assessment in each test. We analyze the CPU that each application used during the tests. Since the experiment was repeated 60 times for each application, the presented CPU is an average value of all the executions. Fig. 5.5 presents the average CPU usage in percentage per test for each application.

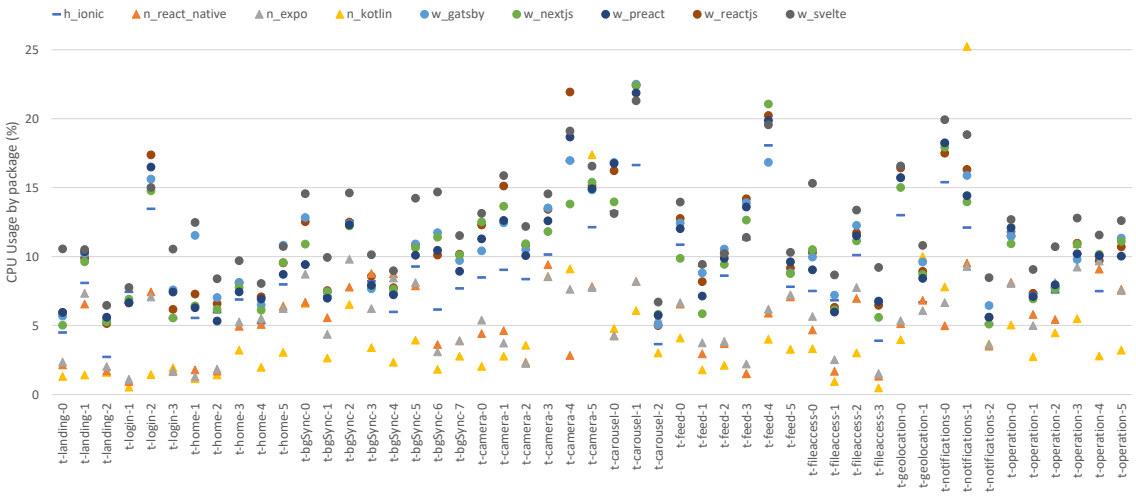


Figure 5.5: CPU Usage by package (%) per application and test

Overall, it is clearly visible in Fig. 5.5 that native applications (marked with triangles) used less CPU than all the other applications throughout the tests. Also, Ionic (marked with a dash) uses less than web applications (marked with circles).

Regarding native applications, Kotlin presents as a clear winner in terms of CPU usage. Kotlin is more efficient throughout all the tests, except when accessing native features (e.g., `t-camera-5`, `t-notifications-1`, `t-geolocation-1`). In two of these tests, Kotlin is the application that uses more CPU. We can conclude that we should avoid using Kotlin when accessing native features is required (if we are concerned with the CPU requirements and, consequently, the devices' battery). On the other hand, Expo presents slightly higher CPU

usages than React Native. This was expected due to the abstractions and little overhead introduced by Expo. However, the differences are minimal. We also highlight that although Kotlin presents better results than these two applications, in the use cases related to native features (and overall in all scenarios tested), React Native and Expo the CPU usage remains low and to some degree constant when compared to the others. Therefore, for intensive applications that may require accessing native features, these cross-platform applications are an efficient a solution.

If we compare Ionic with, for example, Preact (representing the web applications) we can observe that Ionic always uses fewer CPU than the web applications in all the use cases tested. We can conclude that in scenarios with CPU-restrictions, we can transform a web application into a native one using Ionic, and it does not impact the CPU usage. In fact, the application would use slightly less CPU.

Regarding the web applications between themselves, we can observe that all of them behave similarly throughout the tests, except for Svelte. As said before, Svelte interacts with the Real DOM and this operation proves to be more expensive than interacting with a virtual DOM as react-based applications do. Hence, in scenarios where there are more limited CPUs, React.js alternatives should be selected over Svelte.

5.4 RAM Consumption

This section presents an analysis of the memory allocated by the system during the benchmarking campaign. We present and analyze the RAM Consumption (MB) for each test per application. As explained before, each experiment was repeated 60 times for each application, and therefore the results presented correspond to the average of all the memory gathered throughout the tests. Fig. 5.6 presents the total memory (RAM) used by the system.

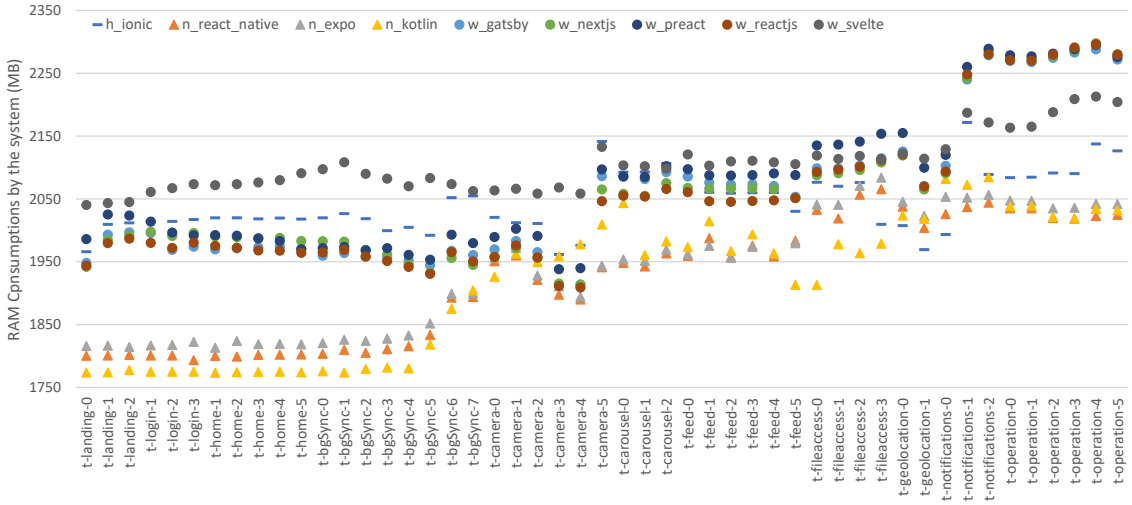


Figure 5.6: RAM Consumption by the system (MB) per application and test

The memory consumption chart is similar to the CPU usage chart in that native applications always allocate less resources (in this case memory) than the competitors. Overall,

since the results are similar, the conclusions regarding memory consumption are also similar to the ones described in the last section. It is clear that the native applications consume way fewer memory than the web-based applications across all the tests. We highlight again that in scenarios with memory or CPU constraints, all the native applications are better alternatives. In between native applications, Kotlin has a slight advantage over React Native and Expo except in the scenarios that interact with native features (e.g., geolocation, file access or camera).

Between the web-based applications, the tests that required more memory are the ones that involve adding a lot of elements to the screen (e.g., carousel or feed). In these scenarios, using a lot of memory may degrade the devices' performance [3] which is visible if we cross the memory results with the test duration presented in Section 5.2. This also visually observed during the tests' execution (i.e., the web applications were more sluggish in these tests). On the other hand, the experience was far more fluid in the three native applications. This is explainable if we remember that this type of application uses virtualized solutions to render elements to the screen. Nowadays, the majority of popular applications are data-driven (mostly from external sources similar to our carousel component). In order to improve the user experience, when implementing any application type mentioned in this study, virtualized solutions should also be considered.

5.5 Static Measurements

Static Measurements is the last section of result analysis. This section presents an analysis of measurements gathered before executing the applications. Although this type of measurements was not the primary focus of this benchmark, the four different static metrics allow some more interesting comparisons between the development tools (e.g., about the source code and the development process). Also, collecting this type of metrics was important to prove that our framework is extensible not only to gather measurements during the execution of the applications but also before. Application size is related to the performance of the applications, but the other metrics are related to the developer experience (build times) and to the maintainability of the development tools under assessment. The build duration was collected several times for each application, and therefore the value presented correspond to an average. Table 5.3 presents our findings.

Regarding the application size, it is expected and observable that web applications a far smaller value than the other type of applications. In between the web applications, Svelte presented the smallest value with around 54.2 KB. This expected because Svelte compiles everything at build time, and therefore does not need to bundle the whole library code to make computations at runtime as React and many other development tools do [34]. Consequently, Svelte application size for small applications is lower than a react-equivalent application but as the application size grows this may not be verified (more details are explored in repository [96])

preact presents the smallest size of all the React-based applications. This was also expected, since preact is a minimalistic version of React.js. The Ionic application size is 5 MB, which is great for a Native Android application. It is only 1.3 times bigger than Kotlin application size. On the other hand, Expo is around 10x larger. Since we were using Expo, to build

Table 5.3: Static measurements gathered per application

App ID	Size (KB)	LoC	Total dependencies (dev)	Avg. build time (s)
h_ionic	5301.00	1737	26 (12)	61.53
n_expo	54160.49	1368	25 (3)	198.00
n_kotlin	4002.88	2588	12 (0)	9.85
n_react_native	32704.91	1340	24 (4)	67.53
w_gatsby	211.00	1530	5 (6)	39.38
w_nextjs	243.00	1658	4 (5)	28.12
w_preact	60.50	1657	5 (8)	85.33
w_reactjs	190.00	1610	6 (7)	19.70
w_svelte	54.20	1458	6 (7)	23.30

the application, the Expo Cloud system was mandatory to build the application. As we can observe with React Native, which we were to build the APK locally, had an application size 1.66 times lower than Expo. This is also related to the abstractions introduced by Expo, which results in a bigger application size.

Regarding the dependencies number, Ionic required a high number to transform a valid React.js app (13 dependencies) into a valid hybrid app (38 dependencies). Also worth noting that Gatsby, Next.js and preact required less than 5 dependencies to be built.

Implementing the Expo or React Native application took fewer lines of code than all the other applications. Considering that Expo and React Native are cross-platform tools that can run on several platforms, this measure suggests that these tools seem easier to maintain than the native application Kotlin. We also highlight Svelte that took the fewest lines of code among the web applications.

Concerning the build times, Craco, the tool used to build the React.js app, presented the best results with an average of 19.70 seconds to build the production version. Next.js also showed good results: under half of a minute. Both these tools use webpack under-the-hood. Svelte, which presented faster build times than Next.js but inferior to Craco, uses a tool called Rollup. The comparison between the different JavaScript bundler tools are out of the scope, but a great comparison is provided in the study [97].

5.6 Threats to Validity

In this section, we discuss the points that threat the validity of the experimental results.

`T1` **Representativeness of the Reference App Spec**: A limitation in several studies is the lack of representativeness of the applications under assessment. To mitigate this issue, we analyzed the most popular applications and their features. Then, we built a Reference App Spec that covers the majority of the use cases observed.

T2 **Applications developed in-house**: Although this might introduce some bias in the assessment, we tried to mitigate it by following a reduced yet solid set of the best development practice. This will be further mitigated as this will be mitigated once the project is open to the community. Engineers will then be able to improve or fix the existing implementations (e.g., some applications may implement the features in a non-optimized way) and to add new development tools.

T3 **Automation Testing technologies overhead**: It is possible that the technologies selected to automate the functional tests (i.e., WebdriverIO and Appium) may introduce some overhead. However, this effect should be similar for all the alternatives considered, and therefore have a reduced impact in the relative comparisons. Furthermore, we analyzed several automation libraries and WebdriverIO, and Appium were the most popular ones (cf. Section 4.2.3) extensively used by the community. Also, the majority of cloud-base systems for automation are built on top of them [98]. The scrutiny that these tools have already been subjected to mitigate this threat.

T4 **Only one device used**: during the benchmark campaign, we only executed the applications in one medium-end Android device. Although this helped draw comparisons for this range of devices, the same may not be verified in smartphones with other specifications (e.g., web applications in devices with more memory available may present faster results). To address this we propose to add more devices to the benchmark in the future.

Chapter 6

Conclusion and Future Work

The huge offer of development tools to produce mobile applications makes the process of selecting one very difficult. Developers normally feel overwhelmed and end choosing the wrong tool for the job. Hence, there is a clear need for tools and principles to assess different development tools that already exist and the ones that might appear.

This study proposes a new framework for the assessment of different development tools capable of producing mobile applications considering several properties, such as performance, reliability, and dependability. As it is not feasible to directly compare these development tools, the framework defines a methodology where we compare representative applications instead.

To demonstrate our framework, we designed a concrete benchmark focused on performance that compared nine different development tools popular in the JavaScript community and Kotlin as a reference for native applications. Based on an extensive analysis of the popular mobile applications, we defined a set of representative features that the applications implemented. We also developed an auxiliary tool to support the benchmark campaign, which automates functional tests and collects metrics carefully selected to assess the applications' performance. Finally, the benchmark concludes with an analysis of the campaign results.

The results show that our framework can indeed be used to compare different development tools. Our findings showed that even though Ionic was less efficient in terms of resources (i.e., Memory consumption and CPU usage), it presented the fastest response times. Also, the native applications were the most efficient but presented slower response times. The differences between Expo and React Native are not significant, which demonstrates that Expo can facilitate the development of cross-platform applications (even supporting more platforms) without compromising their performance. We observed that web applications are already a viable alternative to native applications in most scenarios and present faster response times at the cost of higher memory and CPU usages. In between web applications, tools that are built with performance as the first requirement (i.e., Next.js and preact) presented as the fastest and more efficient applications.

As future work, we have the main objective of opening the project to the community. This would help not only improving the existing implementations, but also to add more development tools. Once there are significant changes, we plan to execute more benchmark rounds.

We also plan on supporting other type of devices. First, supporting other Android devices should easy to accomplish. In addition, a more ambitious goal for the future is to support iOS development tools and iOS devices. Although this may impose some challenges, our framework was designed to support any type of device and a huge set of development tools. In our benchmark, we also used technologies that can be extended to support the iOS environment.

During the results analysis and the benchmark campaign, we observed that some applications may benefit with certain optimizations (e.g., virtualizing elements in web applications). Creating different versions of the existing implementations can be added to the project for future rounds, for example, using virtualized solutions to render content to the screen for the web and hybrid applications.

Although our experiment was focused on performance, the framework is designed to support other properties and the benchmark can be extended to support them. For example, we can add properties such as dependability (by injecting faults) or security (by injecting attacks).

References

[1] Y. Lin, "10 MOBILE USAGE STATISTICS EVERY MARKETER SHOULD KNOW IN 2021 [INFOGRAPHIC]." [Online]. Available: https://www.oberlo.com/blog/mobile-usage-statistics

[2] S. O'Dea, "Number of smartphone users by country as of September 2019 (in millions)*." [Online]. Available: https://www.statista.com/statistics/748053/worldwide-top-countries-smartphone-users/

[3] M. Willocx, J. Vossaert, and V. Naessens, "Comparing performance parameters of mobile app development strategies," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 38–47.

[4] S. Richard and P. LePage, "What are Progressive Web Apps?" [Online]. Available: https://web.dev/what-are-pwas/

[5] I. Malavolta, "Beyond native apps: web technologies to the rescue! (keynote)," in *Proceedings of the 1st International Workshop on Mobile Development - Mobile! 2016*. Amsterdam, Netherlands: ACM Press, 2016, pp. 1–2. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3001854.3001863

[6] Z. Bowden, "Microsoft announces Windows 11 will be able to run Android apps," Jun. 2021, section: article. [Online]. Available: https://www.windowscentral.com/microsoft-announces-windows-11-will-be-able-run-android-apps

[7] Thurrottfeed, "Microsoft Worked with Google to Bring PWAs to the Play Store," Jul. 2020, section: Dev. [Online]. Available: https://www.thurrott.com/dev/237715/microsoft-worked-with-google-to-bring-pwas-to-the-play-store

[8] K. Maida, "How to Manage JavaScript Fatigue," Dec. 2020. [Online]. Available: https://auth0.com/blog/how-to-manage-javascript-fatigue/

[9] TechEmpower, "TechEmpower/FrameworkBenchmarks: Source for the TechEmpower Framework Benchmarks project," Oct. 2020. [Online]. Available: https://github.com/TechEmpower/FrameworkBenchmarks

[10] ScienceDirect, "Traditional Web Application," Dec. 2020. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/traditional-web-application

[11] A. Gajdos, "Single-Page Application vs Multiple-Page Application: Which One To Choose For Your Project," Apr. 2020. [Online]. Available: https://andrejgajdos.com/single-page-application-vs-multiple-page-application/

[12] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0," p. 6.

[13] M. Mikowski and J. Powell, *Single Page Web Applications: JavaScript end-to-end*, 1st ed. USA: Manning Publications Co., 2013.

[14] Mozilla, "Document Object Model (DOM) - Web APIs | MDN," Oct. 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[15] B. Krajka, "The difference between Virtual DOM and DOM - React Kung Fu," Oct. 2020. [Online]. Available: https://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/

[16] CodeAcademy, "React: The Virtual DOM," Jan. 2021. [Online]. Available: https://www.codecademy.com/articles/react-virtual-dom

[17] R. Harris, "Virtual DOM is pure overhead," Jan. 2021. [Online]. Available: https://svelte.dev/blog/virtual-dom-is-pure-overhead

[18] V. Savkin, "Understanding Angular Ivy: Incremental DOM and Virtual DOM," Jan. 2021. [Online]. Available: https://blog.nrwl.io/understanding-angular-ivy-incremental-dom-and-virtual-dom-243be844bf36

[19] B. Staryga, "SPA SEO: Mission Impossible?" Jan. 2021. [Online]. Available: https://www.magnolia-cms.com/blog/spa-seo-mission-impossible.html

[20] L. Maldonado, "How Next.js can help improve SEO," Jan. 2021. [Online]. Available: https://blog.logrocket.com/how-next-js-can-help-improve-seo/

[21] G. Singhal, "Why Do We Need Single-page Applications?" Jan. 2021. [Online]. Available: https://www.pluralsight.com/guides/why-do-we-need-a-single-page-application

[22] S. Jobs, "Steve Jobs introducing PWA in 2007," Oct. 2020. [Online]. Available: https://www.youtube.com/watch?v=QvQ9JNm_qWc

[23] A. Russell, "Progressive Web Apps: Escaping Tabs Without Losing Our Soul," Dec. 2020. [Online]. Available: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/

[24] T. Jankov, "The Essentials of Building Progressive Web App (PWA)," Jan. 2021. [Online]. Available: https://www.cloudways.com/blog/progressive-web-apps/

[25] I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirovic, "Assessing the Impact of Service Workers on the Energy Efficiency of Progressive Web Apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Buenos Aires, Argentina: IEEE, May 2017, pp. 35–45. [Online]. Available: http://ieeexplore.ieee.org/document/7972716/

[26] A. Osmani, "The App Shell Model," Jan. 2021. [Online]. Available: https://developers.google.com/web/fundamentals/architecture/app-shell

[27] C. Four, "PWA Stats," Oct. 2020. [Online]. Available: https://www.pwastats.com/

[28] P. Que, X. Guo, and M. Zhu, "A Comprehensive Comparison between Hybrid and Native App Paradigms," in *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*, Dec. 2016, pp. 611–614, iSSN: 2472-7555.

[29] I. Malavolta, S. Ruberto, T. Soru, and V. Terragni, "End Users' Perception of Hybrid Mobile Apps in the Google Play Store," in *2015 IEEE International Conference on Mobile Services*. New York City, NY, USA: IEEE, Jun. 2015, pp. 25–32. [Online]. Available: http://ieeexplore.ieee.org/document/7226668/

[30] W. Wu, "React Native vs Flutter, cross-platform mobile application frameworks," p. 34.

[31] "Angular." [Online]. Available: https://angular.io/

[32] "React – A JavaScript library for building user interfaces." [Online]. Available: https://reactjs.org/

[33] eggheadio, "Evan You, creator of Vue.js." [Online]. Available: https://egghead.io/podcasts/evan-you-creator-of-vue-js

[34] R. Harris, "Virtual DOM is pure overhead." [Online]. Available: https://svelte.dev/blog/virtual-dom-is-pure-overhead

[35] "Preact." [Online]. Available: https://preactjs.com/

[36] Malco, "Which To Choose in 2020: NextJS or Gatsby?" Feb. 2021. [Online]. Available: https://frontend-digest.com/which-to-choose-in-2020-nextjs-vs-gatsby-1aa7ca279d8a

[37] "Next.js." [Online]. Available: https://nextjs.org/

[38] "Blitz.js vs. RedwoodJS - LogRocket Blog." [Online]. Available: https://blog.logrocket.com/blitz-vs-redwood/

[39] "SvelteKit • The fastest way to build Svelte apps." [Online]. Available: https://kit.svelte.dev/

[40] "Nuxt.js - The Intuitive Vue Framework." [Online]. Available: https://nuxtjs.org/,/

[41] "SolidJS." [Online]. Available: https://www.solidjs.com

[42] "Marko." [Online]. Available: https://markojs.com/

[43] "Astro." [Online]. Available: https://astro.build/

[44] "Kotlin and Android." [Online]. Available: https://developer.android.com/kotlin

[45] M. Gonsalves, "Evaluating the mobile development frameworks Apache Cordova and Flutter and their impact on the development process and application characteristics," Thesis, Jun. 2019, accepted: 2019-06-25T15:54:41Z. [Online]. Available: http://dspace.calstate.edu/handle/10211.3/211157

[46] "Expo." [Online]. Available: https://expo.io/

[47] L. Dagne, "Flutter for cross-platform App and SDK development," p. 37.

[48] J. Gray, "The benchmark handbook for database and transaction systems," *Mergan Kaufmann, San Mateo*, 1993.

[49] M. Vieira, H. Madeira, K. Sachs, and S. Kounev, "Resilience Benchmarking," in *Resilience Assessment and Evaluation of Computing Systems*, K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, Eds. Berlin, Heidelberg: Springer, 2012, pp. 283–301. [Online]. Available: https://doi.org/10.1007/978-3-642-29032-9_14

[50] K. Kanoun and L. Spainhower, *Dependability benchmarking for computer systems*. Wiley Online Library, 2008, vol. 72.

[51] R. Almeida, M. Poess, R. Nambiar, I. Patil, and M. Vieira, "How to Advance TPC Benchmarks with Dependability Aspects," in *Performance Evaluation, Measurement and Characterization of Complex Systems*, ser. Lecture Notes in Computer Science, R. Nambiar and M. Poess, Eds. Berlin, Heidelberg: Springer, 2011, pp. 57–72.

[52] krausest, "krausest/js-framework-benchmark: A comparison of the perfomance of a few popular javascript frameworks," Oct. 2020. [Online]. Available: https://github.com/krausest/js-framework-benchmark

[53] A. Biørn-Hansen, T. A. Majchrzak, and T.-M. Grønli, "Progressive Web Apps: The Possible Web-native Unifier for Mobile Development:," in *Proceedings of the 13th International Conference on Web Information Systems and Technologies*. Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2017, pp. 344–351. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006353703440351

[54] R. Fransson, A. Driaguine, and J. Hagelbäck, "Comparing Progressive Web Applications with Native Android Applications," p. 59.

[55] S. Manley, M. Seltzer, and M. Courage, "A self-scaling and self-configuring benchmark for Web servers (extended abstract)," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '98/PERFORMANCE '98. New York, NY, USA: Association for Computing Machinery, Jun. 1998, pp. 270–271. [Online]. Available: https://doi.org/10.1145/277851.277945

[56] K. L. Johnson and M. M. Misic, "Benchmarking: a tool for Web site evaluation and improvement," *Internet Research*, vol. 9, no. 5, pp. 383–392, Dec. 1999. [Online]. Available: https://www.emerald.com/insight/content/doi/10.1108/10662249910297787/full/html

[57] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy, "Benchlab: an open testbed for realistic benchmarking of web applications," in *Proceedings of the 2nd USENIX conference on Web application development*. USENIX Association, 2011, pp. 37–48.

[58] F. S. Tahirshah, "Comparison between Progressive Web App and Regular Web App," p. 69.

[59] the benchmarker, "the-benchmarker/web-frameworks: Which is the fastest web framework?" Oct. 2020. [Online]. Available: https://github.com/the-benchmarker/web-frameworks

[60] mathieuancelin, "mathieuancelin/js-repaint-perfs: Playground to test repaint rates of JS libs," Oct. 2020. [Online]. Available: https://github.com/mathieuancelin/js-repaint-perfs

[61] "gothinkster/realworld," Jul. 2021, original-date: 2016-02-26T20:49:53Z. [Online]. Available: https://github.com/gothinkster/realworld

[62] "tastejs/PropertyCross," Jun. 2021, original-date: 2012-09-27T12:41:22Z. [Online]. Available: https://github.com/tastejs/PropertyCross

[63] A. Gambhir and G. Raj, "Analysis of Cache in Service Worker and Performance Scoring of Progressive Web Application," in *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*. Paris: IEEE, Jun. 2018, pp. 294–299. [Online]. Available: https://ieeexplore.ieee.org/document/8441715/

[64] R. Fredrikson, "Emulating a Native Mobile Experience with Cross-platform Applications," p. 19.

[65] G. de Andrade Cardieri and L. M. Zaina, "Analyzing User Experience in Mobile Web, Native and Progressive Web Applications: A User and HCI Specialist Perspectives," in *Proceedings of the 17th Brazilian Symposium on Human Factors in Computing Systems - IHC 2018*. Belém, Brazil: ACM Press, 2018, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3274192.3274201

[66] D. Fortunato and J. Bernardino, "Progressive web apps: An alternative to the native mobile Apps," in *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*. Caceres: IEEE, Jun. 2018, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8399228/

[67] "Most Used And Downloaded Apps In The World 2021." [Online]. Available: https://blog.sagipl.com/most-used-apps/

[68] L. Valdellon, "60 Most Popular Apps on the App Store and Google Play." [Online]. Available: https://clevertap.com/blog/most-popular-apps/

[69] J. Koetsier, "Here Are The 10 Most Downloaded Apps Of 2020." [Online]. Available: https://www.forbes.com/sites/johnkoetsier/2021/01/07/here-are-the-10-most-downloaded-apps-of-2020/?sh=16eebf445d1a

[70] J. Chan, "Top Apps Worldwide for January 2021 by Downloads." [Online]. Available: https://sensortower.com/blog/top-apps-worldwide-january-2021-by-downloads

[71] H. K. Flora, X. Wang, and S. V. Chande, "An Investigation on the Characteristics of Mobile Applications: A Survey Study," *International Journal of Information Technology and Computer Science*, vol. 6, no. 11, pp. 21–27, Oct. 2014. [Online]. Available: http://www.mecs-press.org/ijitcs/ijitcs-v6-n11/v6n11-3.html

[72] "WebdriverIO." [Online]. Available: https://webdriver.io/

[73] D. Kirkpatrick, "Google: 53% of mobile users abandon sites that take over 3 seconds to load." [Online]. Available: https://www.marketingdive.com/news/google-53-of-mobile-users-abandon-sites-that-take-over-3-seconds-to-load/426070/

[74] Ionic, "Ionic React - Ionic Documentation." [Online]. Available: https://ionicframework.com/docs/react

[75] "Guide to Web Authentication." [Online]. Available: https://webauthn.guide

[76] "[FlatList] FlatList and VirtualizedList Scroll performance is laggy after 30+ rows . · Issue #13413 · facebook/react-native." [Online]. Available: https://github.com/facebook/react-native/issues/13413

[77] "Optimizing Flatlist Configuration · React Native." [Online]. Available: https://reactnative.dev/docs/optimizing-flatlist-configuration

[78] "Appium." [Online]. Available: http://appium.io/

[79] L. Tung, "Programming languages: Kotlin rises fastest but JavaScript lures millions more developers." [Online]. Available: https://www.zdnet.com/article/programming-languages-javascript-now-used-by-12-million-developers-but-kotlin-rises-fastest/

[80] "Top 5 UI Frameworks For Android Automated Testing." [Online]. Available: https://saucelabs.com/blog/the-top-5-android-ui-frameworks-for-automated-testing

[81] Brian, "Best Automation Testing Tools for 2021 (Top 15 reviews)," Feb. 2021. [Online]. Available: https://briananderson2209.medium.com/best-automation-testing-tools-for-2018-top-10-reviews-8a4a19f664d2

[82] ——, "Top 15 Mobile Testing Tools for 2020 (Latest Update)," Feb. 2021. [Online]. Available: https://briananderson2209.medium.com/best-mobile-testing-tools-ios-android-3efb84fa39

[83] "Comparing Mobile Automation Testing Tools: Appium, TestComplete, UI Automator, SeeTest, Robotium, XCUITest and more." [Online]. Available: https://www.altexsoft.com/blog/mobile-automation-testing-tools/

[84] "10 Best Mobile Testing Tools To Automate Testing In 2021," Jan. 2021. [Online]. Available: https://theqalead.com/tools/mobile-testing-tools/

[85] "15 Best Mobile Testing Tools for Android and iOS in 2021." [Online]. Available: https://www.softwaretestinghelp.com/best-mobile-testing-tools/

[86] "_USTop 6 Mobile Testing Tools for Test Creation and Automation," Dec. 2019. [Online]. Available: https://bitbar.com/blog/mobile-testing-tools-for-test-creation-and-automation/

[87] "14 Best Mobile Testing Tools for Android & iOS App [Free/Paid]." [Online]. Available: https://www.guru99.com/mobile-testing-tools.html

[88] "Alternatives to Selenium (Appium) for Mobile Testing | SoftwareTestPro." [Online]. Available: https://www.softwaretestpro.com/alternatives-to-selenium-appium-for-mobile-testing/

[89] "Typed JavaScript at Any Scale." [Online]. Available: https://www.typescriptlang.org/

[90] "Empowering App Development for Developers | Docker." [Online]. Available: https://www.docker.com/

[91] "Nginx Proxy Manager." [Online]. Available: https://nginxproxymanager.com/

[92] "AlDanial/cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages." [Online]. Available: https://github.com/AlDanial/cloc

[93] "Performance measurement APIs | Node.js v16.1.0 Documentation." [Online]. Available: https://nodejs.org/api/perf_hooks.html

[94] L. Robinson, "A Complete Guide To Incremental Static Regeneration (ISR) With Next.js." [Online]. Available: https://www.smashingmagazine.com/2021/04/incremental-static-regeneration-nextjs/

[95] B. Vaughn, "bvaughn/react-virtualized," May 2021, original-date: 2015-11-03T00:48:07Z. [Online]. Available: https://github.com/bvaughn/react-virtualized

[96] "halfnelson/svelte-it-will-scale: Generate a chart showing svelte's overhead." [Online]. Available: https://github.com/halfnelson/svelte-it-will-scale

[97] S. Laurila, "Comparison of JavaScript Bundlers," p. 55.

[98] "The Top 4 Open Source Tools That Make Appium Easier to Use | Digital.ai." [Online]. Available: https://digital.ai/catalyst-blog/the-top-4-open-source-tools-that-make-appium-easier-to-use

Appendices

Appendix A

Application specification

Table A.1: Application elements identifiers and their functional requirements

Component	**Identifier**	**Functional Requirements**
Navbar	Navbar_View_Drawer	Main container with the navbar content
	Navbar_Button_Homepage	Link to the homepage
	Navbar_Button_Logout	Button to logout from the application and redirect to the Landing
	Navbar_Row_ <Component.label>	Link to each authenticated component page
Landing	Landing_View_Main	Main container with Landing content
	Landing_Text_Title	Paragraph with application title: `Savery-[Development Tool Name]`
	Landing_Button_ AskPermissions	Button that once clicked asks for the required permissions (geolocation, camera and notifications)
	Landing_Text_ PermissionsGranted	Paragraph that renders after the permissions are granted. Color green and must say "All permissions granted"
	Landing_Button_GoToLogin	Link that redirects to login page
Login	Login_View_Main	Main container with Login content
	Login_Input_Username	Text input field to type username
	Login_Input_Password	Password input field to type password
	Login_Button_Submit	Button that once clicked should send an HTTP request to the backend with the username and password typed. If a JWT is returned should redirect to Homepage
Homepage	Homepage_View_Main	Main container with Homepage content
	Homepage_Text_ NeedPermissions	Paragraph that renders "All permissions" if required permissions granted, "Permissions not granted" otherwise
	Homepage_View_ Components	Container with the list of available components

	Homepage_Row_ <Component.label>	Paragraph that renders each component label (e.g., Camera, Carousel)
Camera	Camera_View_Main	Main container with Camera content
	Camera_Switch_ ToggleCamera	Switch button that hides/displays camera feed
	Camera_View_ CameraAndImage	Container with Camera feed, button and an image
	Camera_Video_Camera	Container that renders camera feed
	Camera_Button_TakePhoto	Button that once clicked takes a picture from camera feed and renders it on container below
	Camera_Image_ FromTakePhoto	Container that renders image from camera feed
Geolocation	Geolocation_View_Main	Main container with Geolocation content
	Geolocation_Text_Waiting	Paragraph with text "Waiting for geolocation" is displayed while querying geolocation
	Geolocation_Text_Success	Paragraph that replaces the previous one with user location once this value is available. Should say "[latitude], [longitude] with accuracy: [accuracy]"
File Access	Fileaccess_View_Main	Main container with File Access content
	Fileaccess_Button_Upload	Button that once clicked open native image picker to select an image from the file system. Selected image should render in container below.
	Fileaccess_Image_Uploaded	Container that renders selected image from the file system.
Notifications	Notifications_View_Main	Main container with File Access content
	Notifications_Button_Local	Button that once clicked sends a native notification to the device
Background Sync	Backgroundsync_View_Main	Main container with Background Sync content
	Backgroundsync_Text_ ConnectionStatus	Paragraph that tracks the user connection. If the user has an internet connection should say "You are currently online!" in green. If the user is offline should say "You are currently offline." in red.
	Backgroundsync_Button_ FetchFromWeb	Button that once clicked should render a random image from the web below
	Backgroundsync_Button_ FetchFromCache	Button that once clicked should render a cached image below
	Backgroundsync_Image_ FromWeb	Container that renders random image from web
	Backgroundsync_Image_ FromCache	Container that renders cached image
Operation	Operation_View_Main	Main container with Operation content
	Operation_Button_ CreateSmall	Button that once clicked renders 100 rows below with random labels created on demand

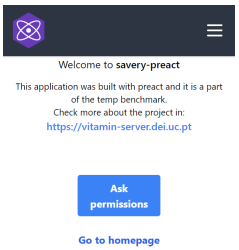
	Operation_Button_CreateBig	Button that once clicked renders 1000 rows below with random labels created on demand
	Operation_Button_Swap	Button that once clicked swaps the second row and the penultimate row
	Operation_Button_Clear	Button that once clicked clears the list of rows
	Operation_Scroll_Rows	Container that contains the list of rows
	Operation_View_Item<id>	View that contain each row and displays its ID and label

This page is intentionally left blank.

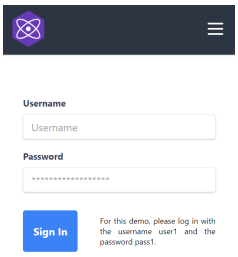
Appendix B

Example of one of the implementations (preact) that follow the Application specification

Next.js Application screenshots:

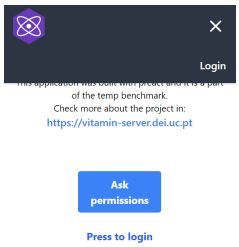


(a) Landing

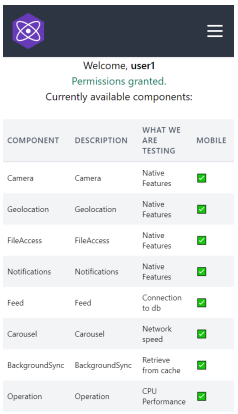


(b) Login

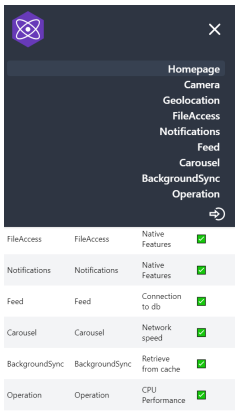
Figure B.1: Unauthenticated components of preact implementation (web application)



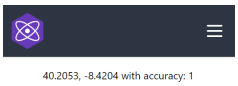
(a) Unauthenticated navbar



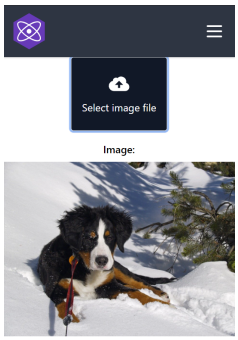
(a) Homepage



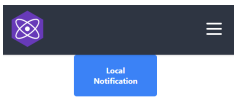
(b) Authenticated navbar



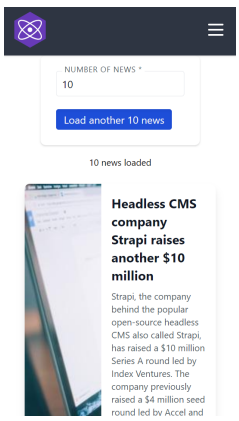
(d) Geolocation



(e) File Access

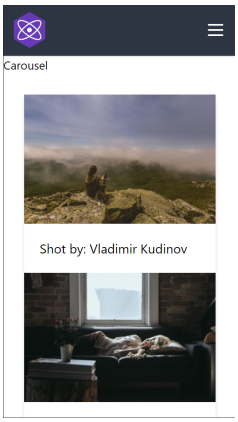


(f) Notifications

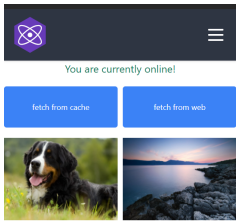


(g) Feed

Figure B.3: Part 1 of authenticated components of preact implementation (web application)



(a) Carousel



(b) Background Sync

(c) Operation

Figure B.4: Part 2 of authenticated components of preact implementation (web application)

Appendix C

Configuration file for Testing and Measurement Tool

Listing C.1: Configuration file for each execution of the Testing and Measurement Tool

```
{
  "app": {//data related to the application under test
    "name": "nextjs", //application name
    "type": "web", //application type (web, hybrid or native)
    "url": "https://nextjs.vitamin-server.dei.uc.pt", //url or application is running in case of web applications, apk location otherwise
    "packageName": "com.android.chrome", //package name to gather the CPU Usage
    "loginInfo": {//data required for apps (e.g., credentials to login)
      "username": "user1",
      "password": "pass1"
    }
  },
  "run": {//data related to the run/execution
    "times": 15,//maximum number of times application should run (tmt will check before starting the execution if this number was already reached)
    "reboot": true, //tmt will reboot device after execution
    "warmup": true, //if true warmup period tests will execute first
    "timeout": 10000,//time in milliseconds until tests timeout
    "ticker": {//data related to measurements gathering
      "active": true, //if the execution should gather measurements (will be false for functional tests and true for benchmark campaign)
      "interval": 200//interval between each measure in milliseconds
    },
    "iterations": 1,//number of runs in a row. does not matter if reboot is true
    "record": false, //record the screen during execution
```

```
23      "needPermissions": true, //the run needs permissions (if true, warm-up
            period tests will accept them)
24      "allComponents": true, //if run should test all components
25      "components": {//data related to components: i) to configure which
            components should run if allComponents is false; ii) define the
            components order in the execution
26        "camera": {
27          "order": 6,
28          "active": false
29        },
30        "geolocation": {
31          "order": 4,
32          "active": false
33        },
34        "fileaccess": {
35          "order": 2,
36          "active": false
37        },
38        "notifications": {
39          "order": 5,
40          "active": false
41        },
42        "feed": {
43          "order": 3,
44          "active": false
45        },
46        "carousel": {
47          "order": 7,
48          "active": false
49        },
50        "backgroundsync": {
51          "order": 1,
52          "active": false
53        },
54        "operation": {
55          "order": 8,
56          "active": false
57        }
58      }
59    },
60    "device": {//data related to the target device
61      "deviceName": "FRD-L09",
62      "platformVersion": "7",
63      "os": "android",
64    }
65  }
```