Haaga-Helia
University of Applied Sciences

Progressive Web App – a new trend in e-commerce
Case study: applying service worker for a webshop

Huong Pham Quynh

Bachelor's Thesis
Degree Programme in Business Information Technology
2018

Haaga-Helia
University of Applied Sciences

Abstract

November 21, 2018

Author(s)
Huong Pham Quynh

Degree programme
Business Information Technology

Report/thesis title
Progressive Web App – a new trend in e-commerce
Case study: applying service worker for a webshop

Number of pages and appendix pages
35 + 6

With the evolution of the Internet, the number of online buyers continuously increase. As a result, the e-commerce market, especially mobile e-commerce, is rising dramatically. Commonly, building native mobile app is considered as a compelling solution for businesses to develop their mobile e-commerce products, which fully exploits device capabilities to provide customers with a pleasant experience. However, Progressive Web App (PWA) was born and has become a game changer.

The purpose of this thesis was not only to clarify the concept of PWA and its characteristics but also to provide practical results of a service worker, explaining how it can help businesses develop a better e-commerce solution. The idea behind this thesis is giving audiences an adequate understanding of PWA to compare and improve current products.

The thesis was divided into two parts including theoretical part and empirical part. The theoretical part composes of literature collected from books, academic articles and reliable online sources. The practical section presents a case study. Additionally, the author's own opinions were included.

The thesis indicated that by providing app-like features to a web app, service worker supports PWA to become a top priority solution in the mobile e-commerce market. As a practical result, the thesis delivered a webshop with an integrated service worker to prove that the service worker can make a web app faster and able to work offline by pre-fetching resource and smart caching.

The thesis concludes that implementing a service worker into a webshop helps business to reach out the customers in mobile e-commerce quickly and economically. However, applying the service worker is more suitable for companies that require small cached content. For businesses who need an app that takes all advantages of the mobile device and a massive deal of cached content, the native app should be a good solution.

Keywords
E-commerce, Progressive Web App, Service worker, Workbox plugin

Table of contents

List of Abbreviations

API	Application Programming Interface
CDN	Content Delivery Network
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTPS	Hypertext Transfer Protocol Secure
PWA	Progressive Web App
URL	Uniform Resource Locator

1 Introduction

In the age of digital, electric commerce (e-commerce) plays a significant role in the economy. With the enormous growth of smartphones as well as tablets, the mobile-friendly factor is considered as one of the most important principles when businesses implement their own e-commerce solution. In that situation, developing a native mobile app is usually a top priority solution to meet the user experience expectation. However, building a native requires a huge of resource and expense.

In 2017, being supported by the Chrome browser, a web technology called Progressive Web App (PWA) becomes a spectacular alternative way for companies to take advantage to develop their e-commerce solutions (Borodescu 2018). PWA is known as a web app which can act like a native app on mobile devices, providing a better experience for users. The heart of a PWA is the service worker which provides significant features. Knowing how a service worker works associated with the existing knowledge of web technologies such as HTML, CSS and JavaScript enable developers to build web apps which indeed encounter with native mobile apps.

All thing considered, it seems beneficial to conduct research on PWA and the role of the service worker in helping businesses deliver a better e-commerce solution.

1.1 Objectives of thesis

This thesis aims to provide adequate knowledge and understanding of PWA's characteristics and terms related to it. Also, the application of service worker, an important component of PWA, in an e-commerce site is presented to help audiences recognise the practical results.

Research question: How service worker can help businesses develop a better e-commerce solution?

This thesis is an effort to bring benefits to both entrepreneurs and developers. Understanding deeply about Progressive Web App concept, they are able to compare it to other current solutions. Besides, developers have the ability to apply service worker to improve existing products.

1.2 Working method

To solve the research question, this thesis performs two tasks including researching and implementing a case study. The researching part provides information about PWA phenomenon and terms related to it. This part aims to shed some light on the PWA concept as well as its impact on the e-commerce market and especially investigate on service worker to explain how it is considered as the heart of a PWA.

The second part is implementing a service worker to a webshop. The purposes of the case study are to facilitate the communication between theory and fact and to present how a service worker improves the webshop.

1.3 Product and studied scope

Apart from presenting the concept of PWA and its role for e-commerce, this thesis focuses on the service worker terminologies and explanations of the service worker's working process in a web app. Moreover, the thesis delivers a webshop which is integrated the service worker as a use case. At the end of the project, the webshop can work offline and is enhanced in page load performance.

The project does not show how to develop a webshop. The webshop used for use case is built beforehand. Besides, the use case is not implemented to meet all requirements of a PWA. That means the webshop does not have some features such as push notification and background synchronisation.

1.4 Structure of thesis

This thesis consists of four chapters. Chapter 1 contains an introduction and presents motivations for this thesis. Objectives and research questions are provided to clarify the purpose and direction of this study.

Chapter 2 is the theoretical part including researches related to the concept and terms used in the thesis. The concept of PWA in the e-commerce market and the working process of service worker with relevant technologies are illustrated in this chapter.

Chapter 3 is the practical part. It presents the implementation process of service worker for a webshop and the comparison of the webshop before and after the service worker is integrated, leading to statements about the role of the service worker in a webshop.

Chapter 4 summaries the outcomes of this thesis and puts forward recommendations.

2 Theoretical background

This chapter summarizes theory of the study by discussing terms and concepts related to the role of Progressive Web App and its characteristics in developing e-commerce solutions, especially in mobile e-commerce. Furthermore, the service worker is highlighted.

2.1 E-commerce

According to Laudon and Traver (2018), e-commerce is defined as using the Internet, the World Wide Web, mobile applications (apps) and mobile browsers to execute commercial transactions digitally. In this context, the Internet refers to a global computer network, the Web is an Internet service which enables users to access to billions of web pages, and the mobile browser is a version of web browsers running on mobile devices.

With the growth of the Internet, the number of online buyers continuously increase to 1.7 billion worldwide in 2017. Shopping and buying online is now a regular and mainstream experience. Most of the global Internet users have become online buyers or shoppers. Therefore, the online retail sector is snowballing than ever. Retailers have started to expand the usage of omni-channel. Omni-channel is described as a cross-channel strategy that organisations use a diversity of channels to sell products and combine their physical stores with their e-commerce platform. They are designed to communicate, support and cooperate. (Laudon & Traver 2018.)

2.1.1 Mobile e-commerce

Newzoo (2018) shows that the number of global smartphone users will reach 3 billion in 2018 and will pass 3.8 billion which is equivalent to 48% of the global population by 2021. This fact has resulted in increasing the demand for mobile e-commerce (m-commerce) which is defined using mobile devices to complete any monetary transaction (Laudon & Traver 2018, 64). Consequently, AudienceProject (2017) states that the number of users using mobile in Finland to buy products online has almost tripled in two years (figure 1).

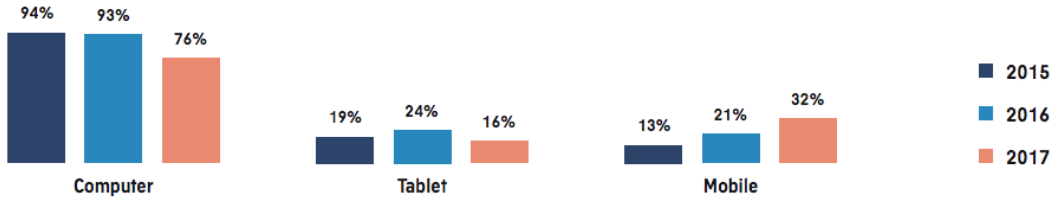


Figure 1. Different devices for online shopping in Finland (AudienceProject 2017)

With the significant growth of m-commerce, optimising e-commerce solutions for better mobile use is crucial. Nowadays, businesses have various selections to develop an m-commerce presence such as mobile website, mobile web app, native app and hybrid app. Each of them has unique advantages to consider.

A mobile website is a version of a regular website which is designed for smaller handled displays with lesser content and navigation in order to help users find what they want fast and effectively. The difference between a mobile website and a regular website is shown in figure 2. The mobile site is cleaner, lighter and more efficiently for finger navigation. The mobile site performance depends on bandwidth. Generally, a mobile website commonly serves more slowly than a regular website which is viewed on a desktop computer. (Laudon & Traver 2018, 218.)

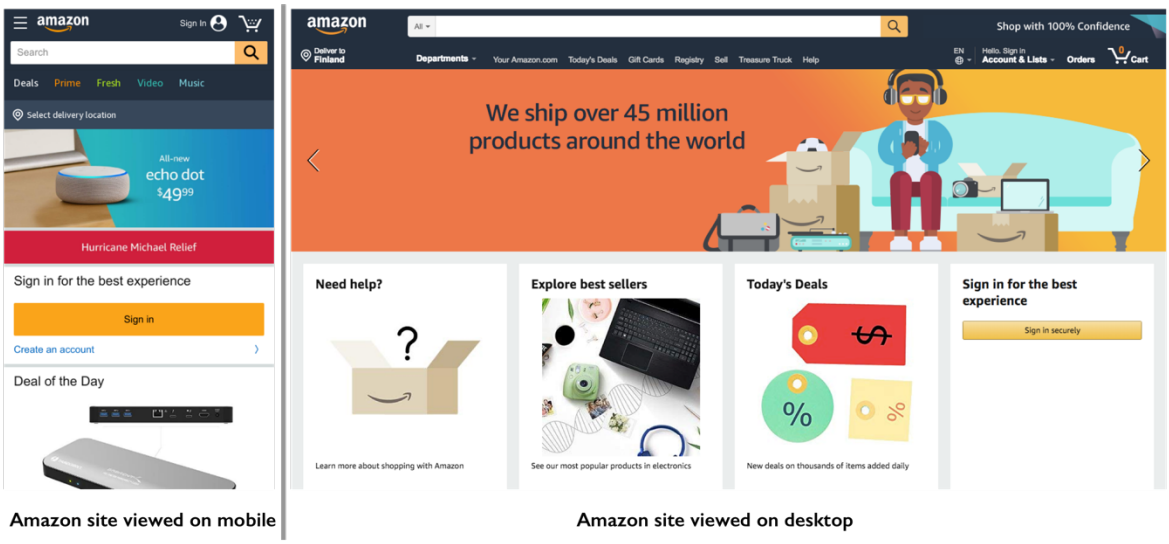


Figure 2. Amazon site viewed on mobile versus desktop computer

A mobile web app refers to an application for mobile devices that requires Web browsers installed such as Chrome, Safari and Firefox to run on the devices. The mobile web app looks like a native app except for the implementation. The basis web technologies are HTML5, CSS3 and different flavours of JavaScript. Mobile web apps are hosted on a remote server using protocols and accessed by users through a unique URL. (Malavolta, Procaccianti, Noorland & Vukmirovic 2017, 36.) When the usage of HTML5 in building websites becomes more popular today, there is almost no distinction between web apps and regular web pages.

Laudon and Traver (2018) define a native mobile app as an application is designed to run on a specific mobile device's hardware and operating system. The native apps are installed from the devoted app store and stored on the user's mobile device. Being built

specifically for one platform, native apps enable to take full advantage of all the device features as well as perform more promptly than mobile web apps. That also means developers need to build different apps for different mobile platforms such as using Java to develop Android apps and using Swift to develop iOS apps because an iOS app which runs on an iPhone is unable to run on Androids phone, and vice versa (Gambhir & Raj 2018, 294).

To combine the advantages of mobile web app and native app, hybrid app was born. Like native apps, hybrid apps are downloaded from the app store and stored locally. As stated by Malavolta, Ruberto, Soru and Terragni (2015, 26), hybrid apps are built by using standard web technologies such as HTML5, CSS3 and JavaScript with the support of native wrapper that enables developers to develop a single mobile app distributed across multiple mobile platforms minimum or even no changes. Nevertheless, developers need to consider how the hybrid app integrates with each specific platform, for example, the adjusting volume button of iPhone can be used to take a picture but not so for Android phones. Figure 3 is the comparison among mobile webs, native apps and hybrid apps from both technical and nontechnical side.

Criteria to consider when choosing a native, hybrid, or Web app approach.

Considerations	Native	Hybrid	Web
Effort of supporting platforms and versions	High	Medium	Low
Device capabilities access	Full	Full	Partial
User experience	Full	Full	Medium
Performance	Very high	Very high	High
Upgrade in the client	Needed	Needed	Not needed
Ease of publication/distribution	Medium	Medium	High
Approval cycle	Mandatory	In some cases	Not required
Monetization in app store	Available	Available	Not available

Figure 3. Comparison among native apps, hybrid apps and web (Serrano, Hernantes & Gallardo 2013, 25)

2.1.2 The new trend of e-commerce

Thanks to the continuous evolution of technology, e-commerce is indeed overgrowing and bringing massive profit to the businesses. Statista (2018) has pointed out that the global retail e-commerce sales reached 2.3 trillion US dollars in 2017 and it is expected to grow to 4.88 trillion US dollars in 2021. Consequently, continually updating e-commerce emerging trends to stay ahead in the competition is necessary for businesses.

Undoubtedly, it is time for mobile. M-commerce sales are expected to comprise 54% of total e-commerce sales by 2021 (Mali 2018). Currently, mobile customers desire to fast speed, app-like experience, and latest features available on their devices. While a regular mobile web app cannot meet the requirements, businesses tend to develop their native apps. However, the cost of building and maintaining a native app is expensive, and it is challenging to get customers to use the app frequently. Moreover, the memory of a mobile device is limited causing the user must decide if the new app is worth installing. According to the study (Pratskevich 2016), just 26.4% of users who visit a page in an app store install the app. The other 73.6% do not want to try the app. Among the ones who install the app, an average of 99% is dropped in the following 90 days.

In 2015, a kind of web app was developed to solve these problems. It is called Progressive Web App (PWA) which is a website taking the features of a native app. Increasingly, PWA with its advantages has become a trend to consider in developing e-commerce.

2.2 Progressive Web Application

Osmani (2018) defines PWAs as “use modern web capabilities to deliver an app-like user experience. They evolve from pages in browser tabs to immersive, top-level apps, maintaining the web's low friction at every moment”. Discovered by Google, Progressive Web App is considered as a bridge between native apps and web apps. Developed using web stack (HTML5, CSS3 and JavaScript) and new Web APIs, PWA runs on browser without installation from app stores, but it can be accessed via an app icon like mobile apps (Gambhir & Raj 2018, 295).

Recently, browsers like Chrome and Opera have fully supported PWA. Firefox is on its way to accomplish several remaining features of PWA. Samsung has a serious commitment to PWA deriving from their leading participation in some of the critical standards work. With iOS 11.3, Apple has finally added support for the basic set of new technologies that make PWA work. (Borodescu 2018.) Besides, Microsoft (2018) states that PWA has sufficient access to Windows 10 feature APIs.

2.2.1 Progressive Web App features

Being first presented by Russell and Berriman in 2015, PWAs were assumed to a revolution class of applications. PWAs have significant characteristics as they can operate on any supported mobile browsers without installation via an app store and work even in the slowest connections, they can be added to the home screen like a native app, and they

can continuously update when running in the background. To form the PWAs for a baseline, Google Developers (2018a) has delivered a Progressive Web Apps checklist including site is served over HTTPS, mobile-friendly design, all app URLs load while offline, be able to add to home screen, the first load is fast even on 3G, sites work in multiple browsers, transitions should be snappy even on a slow network and each page has a URL. With these requirements, PWAs are described with keywords including progressive, responsive, app-like, fresh, safe, discoverable, re-engageable, installable and linkable (Osmani 2018).

Progressive: Being developed with progressive enhancement as a core principle, PWAs enable every user to access them despite of browser selection. Progressive enhancement firstly focuses on the core content like static HTML contents, making them be always available regardless of browser or connection speed. Then, more advanced features such as animation and asynchronous network access are progressively performed if the browser supports.

Responsive: PWAs need to accommodate seamlessly to screen size, resolution, and aspect ratio of any devices.

Connectivity independent: Thanks to the service worker (which is discussed respectively in section 2.3), PWAs are able to work offline or on low-quality networks.

App-like: PWAs make users feel like they are using a native app instead of a web with smooth transitions and app-style interactions and navigations.

Fresh: PWA is self-update. That depends on if the user is connecting to the internet.

Safe: Being served only through HTTPS, PWAs prevent attacks and accessing content from unauthorised users.

Discoverable: Search engines classify PWAs as "applications", but instead of being distributed via app stores, they can be shared through a URL that enables search engines to find them. It is a vital advantage of PWAs over native apps.

Re-engageable: PWAs are equipped with features like push notification which can keep customers to use the apps more.

Installable: It is easy to install PWAs by visiting and adding the site to device home screen.

Linkable: PWAs are directly shared via URL without complicated installation.

A web app can be considered as a PWA if it is served through HTTPS for a secure connection and uses a web app manifest and at least a service worker (Malavolta & al. 2017). According to Osmani (2018), web app manifest is defined a simple JSON file support developers to control how the app appearance to the user's desire areas like the device home screen and instructs what the user can launch and how he/she can launch it. He also defines a service worker as “a script that runs in the background, separate from your web page. It responds to events, including network requests made from pages it serves.”.

2.2.2 Comparison between PWA and other e-commerce solutions

Choosing the most proper mobile presence keeping users’ engagement is always challenging for business. To take all advantages of the device operating system features as well as to bring about satisfying experience for users, developing native apps and PWAs are the most preferred choices. Table 1 presents the comparison between native apps and PWAs in some common factors.

Table 1. Comparison between native apps and PWAs

Criteria	Native apps	PWAs
Installation	Need to be downloaded from app stores	Can be added from browser
Audience reach	Reach small audience	Reach wide audience
Cost of production and maintenance	Expensive cost to build and maintain	Lower cost compare to native apps
Distribution	Submit different app versions for different marketplaces	Be access in multiple platforms via browsers
User experience	Good	Good but having limitation

In general, PWAs are more straightforward in acquiring new customers to use company apps. While native apps require to be downloaded from an app store, Behl & Raj (2018, 370) states that about 80% of users use only a particular number of apps regular. That causes wasting memory for the apps which user does not use often. Hence, PWAs help users to reduce cost by saving memory. Moreover, with the likability, PWAs facilitate the

sharing of information and favourite products among users. Besides, PWAs work like websites so they can be indexed and are supported by search engines thanks to Google mobile-first indexing while to increase the rank marketplace store of native takes extra time and cost.

The cost to build and maintain native apps is more expensive than PWAs. Regarding native apps, business demands to build separate apps for different platforms and update them regularly. Also, the developer needs to submit these apps to the corresponding app stores such as Apple's App Store, Android's Play Store and Window Store. Each app store has various requirements to get the app to be published and sometimes requires a fee to access. On the other side, PWAs are web-based. Multiple platforms can use one codebase. Therefore, update and maintenance are more manageable. (Naylor 2017.)

Native apps bring about the compelling experience because the user can open them fast and use device-specific features integrated into the app such as proximity sensor, ambient light detection, or smart lock. On the other hand, PWAs performance is improved. With the support of service worker, PWAs can work offline and in a poor network connection. Like native apps, push notification helps PWAs to increase customer engagement. (Belh & Raj 2018, 370.) However, the functions of PWAs are limited when comparing to native apps such as geofencing, mobile payment and interact with other apps.

It is oblivious that PWAs bring revolution to the e-commerce market. Various large brands have implemented PWAs, and some have achieved significant results such as AliExpress and Lancôme.

As a primary platform to advertise, the mobile web is always focused on design and functionality. However, AliExpress found difficulty in building an engaging web experience as fast as their mobile apps. To solve their problems, they developed a cross-browser PWA to consolidate the best of their app with the extensive reach of the web. This led to surprising results for AliExpress including 104% increase in conversion rates for new users, twice more pages visited per session per user and 74% increase in spending time per session across all browsers. (Google Developers 2017a.)

Lancôme had trouble with mobile conversion rates when they did not match those for desktop. While 38% of shopping carts were checked out on desktop, there was 15% on mobile., which implies that customers had obstacles to pay on mobile. Lancôme decided to take PWAs as their solution. Having launched a PWAs with Mobify, Lancôme got 84%

decrease in loading time, 17% increase in conversion rate and 51% increase in mobile sessions. (Google Developers 2017b.)

2.3 Technology framework

In this chapter, JavaScript and its libraries are introduced as main technologies used to develop websites as well as service workers. JavaScript is a programming language making websites more interactive by accessing and modifying their content and markup while they are viewed browsers. It is the only programming language which can run in browsers. The standard for JavaScript is ECMAScript. The latest standardised version is officially called ECMAScript 2015, which is also known as ECMAScript 6 and ES6. There are three majors feature categories that ES6 improves including the better syntax for existing features such as Classes and Modules, new functionalities in the standard library such as Promises, Maps and Sets and entirely new features such as Generators, Proxies and WeakMaps (Rauschmayer 2018).

2.3.1 JavaScript Promises

Elliot (2017) defines a Promise as “an object that may produce a single value sometime in the future: either a resolved value or a reason that it’s not resolved”. Rauschmayer (2018) repose the definition, stating that Promises are an alternative to callbacks for executing, composing, and managing asynchronous operations. Comparing to callbacks, Promises deliver more advantages. Chaining is more manageable by using *.then()* to return a Promise. Besides, function signatures are cleaner due to all parameters are input instead of being mixed with callbacks. Error handling is also more straightforward. (Rauschmayer 2018.)

According to Elliott (2017), a Promise can be in one of three states:

- Fulfilled: the asynchronous operation has succeeded and returns a result.
- Rejected: the asynchronous operation failed. In this state, a Promise has a reason indicating why the action failed.
- Pending: the Promise has not yet been *Fulfilled* or *Rejected*

A Promise is settled when it is either *Fulfilled* or *Rejected*. It can be only settled once then subsequent attempts to settle like calling *.resolve()* or *.reject()* are meaningless. It is a significant feature of Promise. (Elliott 2017.)

A Promise is created using Promise constructor (Figure 4) which takes only one argument, a callback with two parameters, *resolve* and *reject*. The *.then()* is used to access either the future value or the reason why the Promise cannot be *Fulfilled*. This method takes two arguments, a callback for a success case and another for the failure case, which are

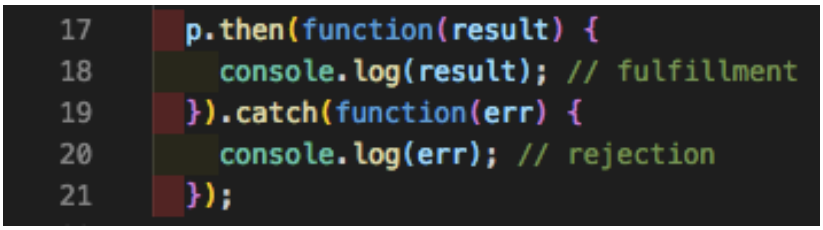
optional. That means a callback either for the success or error case will be passed. (Elliott 2017.)

```
const p = new Promise(function(resolve, reject) {
    if () {
        resolve(value); // success
    }
    else {
        reject(reason); // failure
    }
});

p.then(function(result) {
    console.log(result); // fulfillment
}, function(err) {
    console.log(err); // rejection
});
```

Figure 4. Creating and using a Promise

Instead of using the second argument to handle error, *.catch()* is an alternative way (Figure 5). In case, there is one or more *.then()* callbacks that does not come along with error handlers, the error is passed on to the first error handler after the callback. (Elliott 2017.)



```
p.then(function(result) {
    console.log(result); // fulfillment
}).catch(function(err) {
    console.log(err); // rejection
});
```

Figure 5. *.catch()* as an error handler

2.3.2 JavaScript Worker

According to Green (2012), JavaScript is a single-threaded language meaning multiple scripts cannot run at the same time. Thanks to web workers - a dominant feature of HTML5 - providing API, developers enable to run JavaScript in a separate thread that does not conflict the user interface. He states that “the Web Workers specification defines an API for running computationally intensive code in a thread other than the web application user interface”. That means developers can execute long-running scripts without affecting the user interface since the worker lives in its own thread. However, web workers are heavyweight so that developers need to consider when using them. Green (2012) suggests that web worker should be used if a task takes more than 150 milliseconds and for mobile applications, the time is even shorter.

Web workers do not have access to several things such as DOM of the parent page, *window* object, *document* object and parent objects. Due to its multi-thread nature, web worker can only use limited features of JavaScript including navigator object, location object (read-only*), XMLHttpRequest* function, *atob()* and *btoa()* functions for converting Base 64 ASCII to and from binary data, *setTimeout()* / *clearTimeout()* and *setInterval()* / *clearInter-val()*, *dump()*, application cache, external scripts using the *importScripts()* method and spawning other Web Workers. (Green 2012.)

There are two types of web workers. The first one registers an *onmessage* event to run in the background for a long-running task and always be there to listen for new messages. The other one does not register *onmessage* event because they handle tasks which are offset from the main web app thread such as fetching and parsing a massive JSON object. They will exit when the process is completed or wait until all of the callbacks are done. (Green 2012.)

2.3.3 React essentials

React is a JavaScript library for building user interface (UI). React was created to serve large-scale user interfaces, dealing with data changing over time. As a pure JavaScript library, React can run on browsers, servers and mobile devices. The goal of React is reducing the complexity of creating and maintaining user interfaces by breaking them into reusable components such as a textbox, a button or a form. (Gackenheimer 2015.)

A React component is created using an ES6 class with a *render* method to return a React element and provide a container location where the component will be added or mounted to the DOM. As a transform layer, JSX (a syntax extension to JavaScript) converts the XML React components syntax into the one that React uses to render elements in JavaScript. Besides, two kinds of data which React components listen to are *props* and state. *Props* are properties given to component as it is initialised and they are not changing during the lifecycle of the component. *State* is set on each component when it is created and changed during the lifecycle of the component. The component is re-rendered if *state* or *props* are changed. (Gackenheimer 2015.)

React uses virtual DOM - an in-memory, lightweight representation of the DOM - to examine the current state of the application. All changes are done first to virtual DOM then compared to current DOM and only changed parts of the DOM are updated. This process makes React be more efficient and faster. (Gackenheimer 2015.)

2.4 Service worker

The PWAs have offline mode capability and push notification and self-update features like native apps, which is due to the service worker. Service worker is “a worker or a servant of the browser that works in a separate browser thread and works along with several APIs to provide native application like features”. It is responsible for pre-loading content of the app and serving it to users. Service worker also supports offline experience, handles network requests and minimises the data traffic. (Behl & Raj 2018, 368.)

Gambhir and Raj (2018, 295) state that as a JavaScript Worker, service worker is not able to access DOM directly. Alternatively, the service worker interacts with the pages it controls by responding to messages sent via *postMessage()* and these pages can manipulate the DOM if required. Service worker acts as a network proxy which intercepts network requests and handles them. In other words, a service worker sits between the app and the network, handling when the browser requests any resource such as an HTML page or an image. For each request, service worker can choose to go to the network and fetch new content from a server or to retrieve resource from cache storage. (Gardner 2018, 185.)

Service worker has its own life which is separate from the web page. That means it does not disappear when the user closes a tab or quit the browser. Service worker works in a different context and is event-driven. (Gardner 2018, 188.)

2.4.1 Scope

As a kind of web worker (Malavolta & at. 2017, 36), service worker inherits the characteristic of running in the background. A service worker executes within a *ServiceWorkerGlobalScope* extending *WorkerGlobalScope*. Otherwise, a site in the browser performs in a browser context where Document objects are displayed to the user. Due to various contexts, a service worker and a site cannot directly modify each other. (Gardner 2018, 189-190.)

According to Gardner (2018, 190.), a service worker applies to a range of URLs it controls and is not able to handle anything out of this scope. Besides, in order to implement the Service Worker API, a browser needs to follow registered service workers and the scope which each of them manages. A service worker is responsible for only a URL at one time.

2.4.2 Life cycle and events

According to Archibald (2018), the objectives of a service worker lifecycle (figure 6) include making offline-first feasible, supporting a new service worker to be ready without disturbing the current one, assuring an in-scope page is controlled by the same service worker (or no service worker) and ensuring only one version of the site operates at a time.

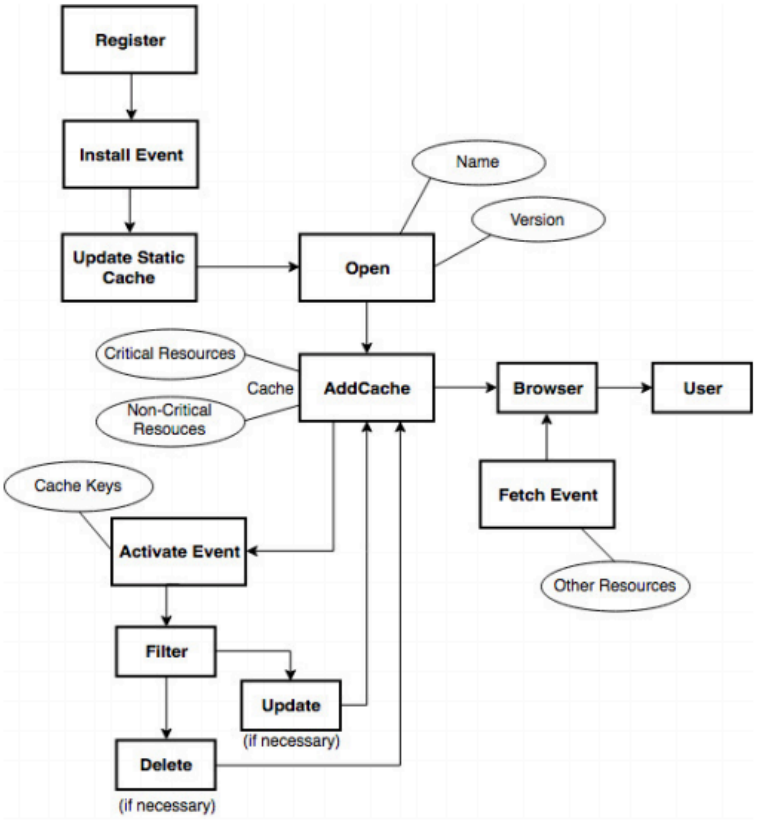


Figure 6. The service worker life cycle (Gambhir & Raj 2018, 296)

According to Gambhir and Raj (2018, 295), the service worker is downloaded in the background when the user visits the site. When the user comes back, the browser compares the old and the new file, deleting the old file if finding any slight difference between them. Besides, Gardner (2018, 188) points out that service worker is event-driven so that if a service worker does not have any events to handle at the moment, the browser has right to terminate it. Several notable events that make service worker remain involves two life cycle events (install and activate) and functional event (fetch).

The *install* event is the first one and trigged only once in the lifecycle of a service worker. To get service worker installed, the developer needs to register it first in the site's JavaScript entry file. This leads to browser starts installing the service worker in the background then the service worker initially caches the necessary assets of the site by opening a

cache, caching the files and finally verifying if all of the required assets are cached. (Gambhir & Raj 2018, 295.)

The *activate* event is dispatched when the service worker is ready to control all page falling under its scope and handle functional events like push and sync (Archibald 2018). The *activate* is fired only once to update or delete files from cache (Gambhir & Raj 2018, 295). Then the service worker is in control, it either handles fetch and message events, which happen when the page makes a network request or message, or it is terminated.

2.4.3 Caching

In the background, the service worker is terminated when it is idle, and it is restarted when required. Therefore, caching technique becomes a vital technique to keep data persistent. (Gambhir & Raj 2018, 295.)

A service worker can manipulate items like creating, adding or removing them in caches. Each cache object is a collection of Response objects keyed by Request objects. All available caches are accessed and handle by *CacheStorage* interface – a kind of directory. Moreover, they are controlled by the logic in a service worker. There are three methods to cache files into the Cache objects including *Cache.put(request, response)*, *Cache.add(RequestoOrURL)* and *Cache.addAll([Urls])*. *Cache.put* allows the developer to put a Request-Response pair into the Cache object. *Cache.add* and *Cache.addAll* go to network to retrieve the corresponding responses and store them. When all the files are cached, the service worker is installed completely. (Gardner 2018, 199; Gaunt 2018.)

After being installed, the service worker starts receiving HTTP request events when the user navigates to another page or refreshes the current page. At that moment, the service worker decides to return cached assets or perform a new request and then cache the result. There are three methods to find cached results: *Cache.match(request)*, *Cache.matchAll([requests])* and *caches.match(request)*. *Cache.match(request)* finds a match for the given request in a specific Cache object. *Cache.matchAll([requests])* looks for all results that match an array of requests inside of a particular Cache object. *Caches.match(request)* spots a match to a given request in all accessed caches. All of them return Promises. If there is a matching result, the response is retrieved. Otherwise, a fetch is performed to make a network request and return the data. (Gardner 2018, 200.)

2.4.4 Fetching

There are several conventional techniques to respond a fetch including using the Fetch API to retrieve a resource from the network or using the Cache API to retrieve a previously cached resource or generating a Response object.

Fetch API enables the developer to take the resource from the network asynchronously. It is quite similar to the *XMLHttpRequest* but provides more powerful and flexible features. The fetch method receives a Request object and returns a Promise then if there is no problem, the Promise resolves to a Response object (Figure 7). (Gardner 2018, 194.)

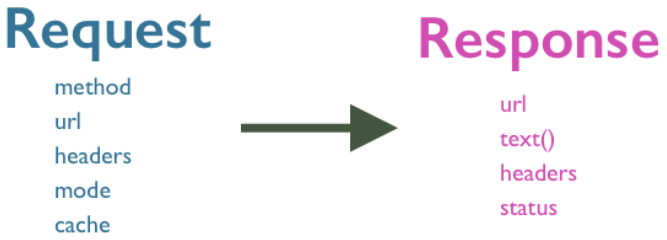


Figure 7. Fetch takes request objects and results in Response objects

In service worker, a handler is registered to process fetch events. The only argument passed to the function is a FetchEvent object. FetchEvent is an event type for fetch events dispatched on the service worker global scope. It holds information about the fetch action, including a Request object and how the receiver handles the response. The service worker intercepts fetch calls and supplies responses which are Promises by using the *respondWith()* method of FetchEvent object (Figure 8). (Gardner 2018, 195.)

```
self.addEventListener('fetch', event => {
  event.respondWith (
    fetch(event.request)
  )
});
```

Figure 8. FetchEvent with *respondWith* method allowing to provide response to this fetch

2.4.5 Updating service worker

When users come back to the web app, the browser tries to re-download the *.js* file containing the service worker code in the background. If there is any difference in the prior service worker' file compared to the current file, the browser considers it new. Then the new service worker will be started, and the *install* event will be executed. At this time, the

old service worker is still controlling the current page leading to the new service worker enters a waiting state. Once the currently opened site is closed, the old service worker is killed by the browser, and the new service worker takes full control. It is time for *activate* event is fired. (Gaunt 2018.)

Cache management is the most common task happening in the *activate* callback because if any old caches are deleted in the *install* step, the old service workers which control all the current pages will stop serving files from that cache. (Gaunt 2018.)

2.4.6 Browser support

Table 2 and Table 3 show that all major browsers support service worker for both desktop and mobile (Mozilla 2018).

Table 2. Desktop browser compatality (Mozilla 2018)

Feature	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari
Basic support	40	17	44	No	24	11.1
install/*activate* events	40	17	44	No	Yes	No
fetch event/request/ *respondWith()*	40	17	44	No	No	No
caches/*cache*	42	17	39	No	No	No
MessageEvent	57	No	55	No	No	No
NavigationPreloadManager	59	No	No	No	46	No

Table 3. Mobile browsers compatality (Mozilla 2018)

Feature	Android Webview	Chrome Android	Edge Mobile	Firefox Android	IE phone	Opera Android	Safari iOS
Basic support	No	40	17	44	No	Yes	11.1
install/*activate* events	No	40	17	44	No	Yes	No
fetch *event*/*request*/ *respondWith()*	No	40	17	44	No	No	No
caches/*cache*	No	42	17	39	No	No	No
MessageEvent	No	57	No	55	No	No	No
NavigationPreloadManager	No	59	No	No	No	46	No

3 Case study: applying service worker for a webshop

This chapter presents how to implement a service worker into an available webshop. Challenges and alternative solutions of the case are provided to figure out the most suitable solution to execute. The outcome of this project is making a webshop have better performance and be able to work offline. The future development is also discussed.

3.1 Executive summary

X is a local food store which provides organic food. They sell food at both physical store and online store. Increasingly, their customers tend to purchase products online and have them to be delivered right to their doors. Apart from selling food, X also provides customers with plenty of healthy recipes using their food products. Therefore, their web app is heavyweight due to visual content such as images and advertising banner. As the number of mobile visitors is growing quickly, X needs to improve their mobile web app faster which does not have their customers to wait for the page loading even on the old mobile devices, giving them more satisfied once they access their website. Further, X desires that their customers can visit their webshop even when there is no internet connection. For example, their customer goes to X store to shop products for recipes on the web app. If the customer does not have internet, they still possibility access the recipes they have checked before.

This case goals to solve all the challenges that X has to face at that time. The possible solutions are developing a native app or enhancing the current web app to a closer PWA. The native app always brings about the best user experience and can serve an enormous number of customers at the same time because they take all advantages of the mobile devices. However, developing a native app is a long story. It requires a great deal of time for investigating, building and maintaining. That means the cost is expensive. Otherwise, improving the current solution, providing it with advantages of a PWA by implementing a service worker, is reasonable and doable. The disadvantages are that a mobile web app is not fast and unable to serve many transactions as a native app. However, implementing a service worker for the current web app is suitable for X because their scale is small (just serving for the local customers) and the cost is cheaper.

The challenges are resolved thanks to the service worker. The service worker is successfully implemented using *injectManifest* plugin. As a result, the webshop performance is improved from score 56 to 80 out of 100 based on testing using Lighthouse, and it is able to work in offline mode. In the future, webshop X need to be optimised key user moments

like time first meaningful paint, speed index and time to interactive. Besides, push notification function should be achieved to improve customer engagement and conversion rate.

3.2 Background

The webshop X sells food products online and provides customers with a plenty of recipes using its own products. Figure 9 presents how a Customer actor interacts with the Webshop X. Top level use cases are View Recipes, View Items and Manage Cart. View Recipes is used once customer browses and takes a look at recipes. This use case is extended by the optional View Items use case: the customer can click to the ingredient of the recipe to view it. View Items is used when the customer wants to find or see any product. Then the customer will decide whether to purchase it or not. If the customer chooses to pay, the checkout process will be performed. Furthermore, the customer can manage his/her cart such as adding items, deleting items or making a purchase.

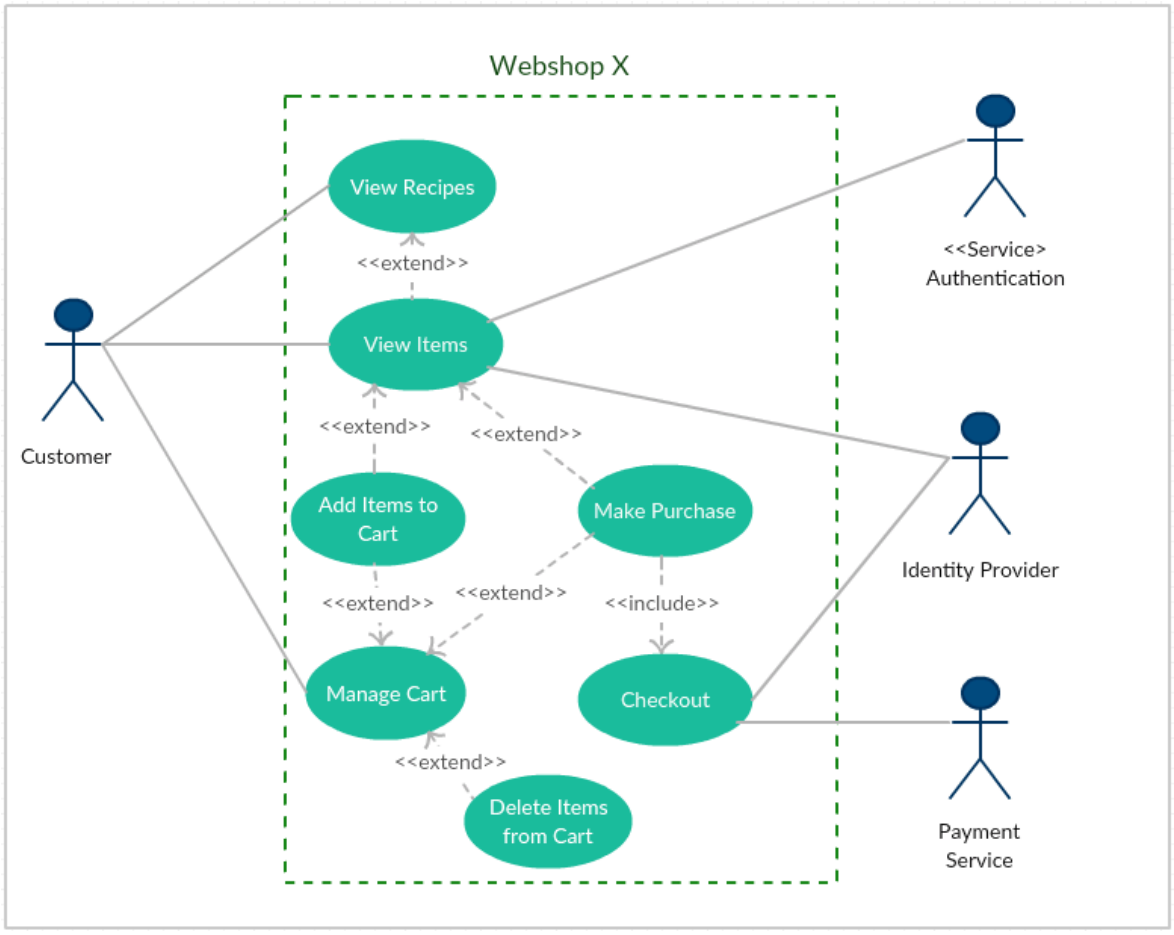
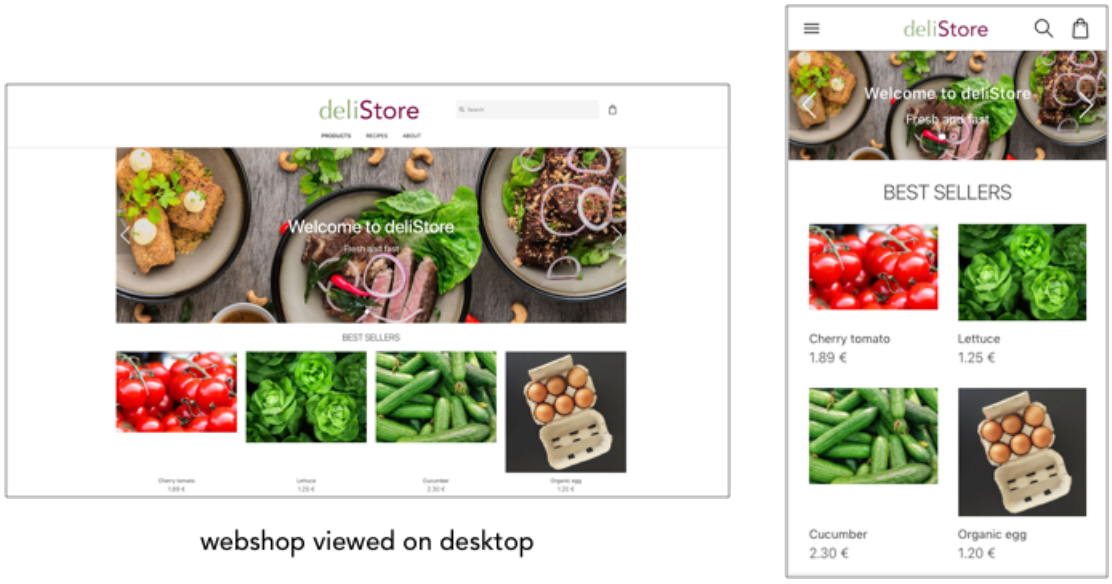


Figure 9. Use case diagram of webshop X

Figure 10 shows how the webshop X is presented in different devices.



webshop viewed on desktop

webshop viewed on mobile

Figure 10. Webshop X is viewed on desktop and mobile

3.3 Challenges

Like any webshops, the current X's webshop has a huge of static assets which introduce its products. The performance metrics of the webshop X is presented as Table 4. Through the performance testing done by Lighthouse (which is mentioned in implementation section), the current performance of webshop is scored 56 out of 100.

Table 4. Performance metrics of webshop X (full report is available at Appendix 1)

Key user moment	Definition	Metric
First Contentful Paint	Measuring when the first content is visible	3.9 s
Speed Index	How quickly the content of page is visibly populated	6.2 s
Time to Interactive	Measuring when page is fully interactive	6.2 s
First Meaningful Paint	Measuring when the primary content of a page is visible	4.0 s
Frist CPU Idle	Recording the first time when the page's main thread is quiet enough to handle input.	4.8 s
Estimated Input Latency	Measuring how long the app takes to respond to user input during the busiest 5s window of page load.	730 ms

Along with the booming of the number of mobile users, the mobile e-commerce market is enormously growing. Having recognised this fact, X desires to improve the webshop performance to satisfy their customer experience even on the old devices.

Besides, applying omni-channel as a strategy in selling and marketing, X aims to provide their customers with a seamless experience. To enhance this, X considers making their webshop work offline. The use case is described as follow: *Laura is checking out the recipes on the webshop X, and she finds several inspiring recipes. On the next day, on the way coming home, Laura pops in the X shop to buy some food. Accidentally, she thinks about the recipes she found yesterday and decides to buy needed ingredients to try them. Without Internet connection, Laura can still open the webshop on mobile browser then access her visited recipes to see their ingredients.* By which, X would like to have their recipes accessible even when there is no Internet connection.

3.4 Solutions

There are several solutions to resolve the X's difficulties. The first solution could be building a native mobile app. Being built with the code living on the device where it is being processed, many elements are preloaded, and the user data is fetched from the network rather than the whole application, the native app is undoubtedly faster than any other platforms. The native app can work offline based on caching data to local storage or data synchronisation. In many cases, the app may have both client-side storage and server-side storage. Therefore, a native app has a large capacity to cache content.

The other solution is improving the current webshop by implementing a service worker. Having the ability to intercept network requests, service workers can enhance performance and offer offline capabilities by pre-fetching resource and smart caching. However, service workers use browser’s memory to cache content, so the capacity is sometimes limited. That means the web integrated service workers is not able to serve a great deal of content when it is offline.

In general, both solutions enable X to solve the mentioned challenges. Nevertheless, building a native app involves more work, especially when making it compatible with different mobile devices and platforms. Moreover, the cost of maintenance is high. Differently, implementing a service worker does not require as much time and resource, and the cost is reasonable. Besides, the scale of X is small so that an integrated service worker is a good fit to provide a compelling mobile presence.

3.5 Implementation

As discussed, implementing a service worker is better. Hence, the process how to integrate a service worker into the current webshop X is presented in this section.

3.5.1 Technology stack

The webshop X is currently a single page application built by React. In the back-end side, the webshop's API is developed using Node.js and mongoDB. Moreover, services and tools are used in the project includes:

Workbox webpack plugins - Workbox offers two webpack plugins including *generateSW* plugin and *injectManifest*. The *generateSW* plugin generates a complete service worker and adds it into the webpack asset pipeline, allowing to pre-cache files with simple runtime configuration requirements. However, it does not allow to add additional scripts or logic, and other features of the service worker cannot be achieved. More difficultly, the *injectManifest* plugin generates a set of assets to pre-cache which is added to a service worker file. It gives more control over the service worker, so it is possible to add more scripts or features. (Google Developers 2018b.)

Netlify – a service for hosting static sites. It offers a rich set of features. Netlify enables developers to deploy a secured site with automatic HTTPS to an ultra-redundant global Application Delivery Network to serve the pages quickly and consistently. Besides, the Continuous Deployment built in Netlify automatically run build commands and deploy the result whenever there is a new push to Git repository. (Netlify 2018.)

Lighthouse – an audit tool. It is in a set of developer tools provided by Google. Lighthouse is available both in Chrome DevTools and a Node module. Lighthouse evaluates how well a page did by running a set of audits then generates a report which includes scores about performance, accessibility, PWA features, best practices and SEO. (Google Developers 2018c.)

3.5.2 Service worker implementation

The service worker implementation process involves four main stages: install workbox plugin, build service worker script, integrate *injectManifest*, and deploy and test. (Figure 11)

Figure 11. Implementation process

Firstly, the workbox plugin is installed through *npm* (package manager for JavaScript). In this case project, the author chooses *injectManifest* plugin to implement the service worker because the webshop has a set of URLs under different hostnames to cache. Using *injectManifest* also allows to customize precached files as well as develop and maintain conveniently. Hence, using *injectManifest* is considered as a long-term solution.

To build service worker, there are two files needed to develop including *sw.js* (service worker script) and *registerServiceWorker.js* (service worker registration script).

In sw.js, the tasks of the webshop's service worker are designed. The service worker uses *workbox.core.setCacheNameDetails* (figure 12) to define the default cache name. Commonly, "prefix" is the name of the app and "suffix" is the version of the service worker. "precache" indicates where assets are precached, while "runtime" defines the name of cache which stores responses while user interacts with the app.

```
workbox.core.setCacheNameDetails({
  prefix: 'deli-store',
  suffix: 'v1',
  precache: 'assets',
  runtime: 'api'
})
```

Figure 12. Define default cache

After that, the precache controller is created to cache primary assets of the webshop before service worker be installed. Workbox plugin automatically generates a list of primary assets (*self.__precacheManifest*) which are used to render the app shell. (Figure 13)

```
// Precache assets needed to render "application shell"
const precacheController = new workbox.precaching.PrecacheController()
precacheController.addToCacheList(self.__precacheManifest || [])
```

Figure 13. Create precache assets

Install event handler and activation event handler are set up by *addEventListener* method (figure 14). In install event handler, the *install* event is configured to wait for the completion of precache installation process. Similarly, the *activation* event is configured to wait until the precache is fully activated.

```
// Install event handler
self.addEventListener('install', (event) => {
  console.log('[Service worker] install')
  event.waitUntil(precacheController.install())
})

// Activation event handler
self.addEventListener('activate', (event) => {
  console.log('[Service worker] activate')
  event.waitUntil(
    precacheController
      .activate()
      .then(() =>
        self.clients
          .matchAll()
          .then((clients) =>
            Promise.all(clients.map((client) => client.postMessage('ready')))
          )
      )
  )
})
```

Figure 14. Set up install event and activation event handler

The next step is to configure navigation of application using *workbox-routing*, which makes routing responses to the matched requests becomes easier. Because the webshop is a Single Page Application built by React, *workbox.routing.registerNavigationRoute* is used to return a particular response (*index.html* file) for all navigation requests. Next, by using a handler, the API routes are registered to intercept *fetch* events to save responses to *runtime* cache storage. In case a request fails, the handler will return the last success response. Otherwise, CDN routes are handled by network-only strategy since responses always have Cache-Control header which helps browser cache automatically. (Figure 15)

```
// Handle navigation requests
workbox.routing.registerNavigationRoute('/index.html')

// Handle API requests
workbox.routing.registerRoute(
  new RegExp('https://web-shop-api\\.herokuapp\\.com/api/v1/.+'),
  ({url, event}) =>
    fetch(event.request).then((response) => {
      const clonedResponse = response.clone()

      if (!response.ok) {
        const msg = `Network request for ${url} was responded
        with status ${response.status} (${response.statusText}),
        falling back to cached resources`

        console.log(msg)
        return caches.match(event.request)
      }

      console.log(`Network request for ${url} succeeded`)
      caches.open(workbox.core.cacheNames.runtime).then((cache) => {
        cache
          .put(url, clonedResponse)
          .then(() => console.log(`Updated runtime cache for: ${url}`))
      })

      return response
    })
)

// Handle requests for images
workbox.routing.registerRoute(
  new RegExp(
    'https://res\\.cloudinary\\.com.+'
  ),
  ({url, event}) =>
    fetch(event.request).then((response) => {
      console.log(`Network request for ${url} succeeded`)
      return response
    })
)
```

Figure 15. Configure service worker routes

When a *fetch* event requests to an unregistered route (other than API and CDN addresses), the default handler will be executed. It is implemented to intercept *fetch* events to return matched cache in offline mode. If there is any error in *fetch* event, catch handler will be fired to fix the problems. For instance, in offline mode, API requests cannot get responses from server. Therefore, the catch handler is responsible for returning matched cached responses (figure 16). At this point, the service worker script is completed.

```
// Default handler
workbox.routing.setDefaultHandler(({url, event}) => {
  if (event.request.method !== 'GET') {
    return
  }

  return fetch(event.request).then((response) => {
    if (response.ok) {
      console.log(`Network request for ${url} succeeded`)
      return response
    }

    let msg = `Network request for ${url} was responded with status ${
      response.status
    }`
    msg += ` (${response.statusText}), falling back to cached resources`
    console.log(msg)
    return caches.match(event.request)
  })
})

// Catch handler
workbox.routing.setCatchHandler(({url, event}) => {
  console.log(
    `Network request for ${
      event.request.url
    } failed, falling back to cached resources`
  )
  return caches.match(event.request)
})
```

Figure 16. Configure routing default handler and catch handler

To make service worker work, the *sw.js* need to be registered to browser at run time leading to the implementation of service worker registration script (*registerServiceWorker.js*). The script firstly checks browser compatibility to decide if registration occurs or not. The registration of the service worker is done by *navigator.serviceWorker.register* only after the window is loaded (figure 17). Last but not least, this script is required to be called firstly in the application.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker
      .register('/sw.js')
      .then((registration) => {
        console.log(
          'Service worker registration successful, scope is: ',
          registration.scope
        )
      })
      .catch((registrationError) => {
        console.log('Service worker registration failed: ', registrationError)
      })
  })
}
```

Figure 17. Register service worker

The next stage is integrating *injectManifest* plugin into build process (figure18). By specifying the *sw.js* source and destination path, the plugin will generate needed service worker script which is used in production environment. Besides, the workbox library is configured to be fetched from CDN hosted by Google. Thanks to this plugin, the precache manifest script is also generated and imported to service worker in the build process.

```
const serviceWorker = new WorkboxPlugin.InjectManifest({
  swSrc: utils.root('src/sw.js'),
  swDest: `${constants.distPath}/sw.js`,
  importWorkboxFrom: 'cdn'
})
```

Figure 18. Integrate injectManifest plugin

When the scripts are developed, the source code is pushed to remote repository in order to trigger a new deployment using Netlify. The deployed app is available to test at https://deli-store-demo.netlify.com/. By accessing the address and auditing the console, the service worker is successfully implemented (figure 19).

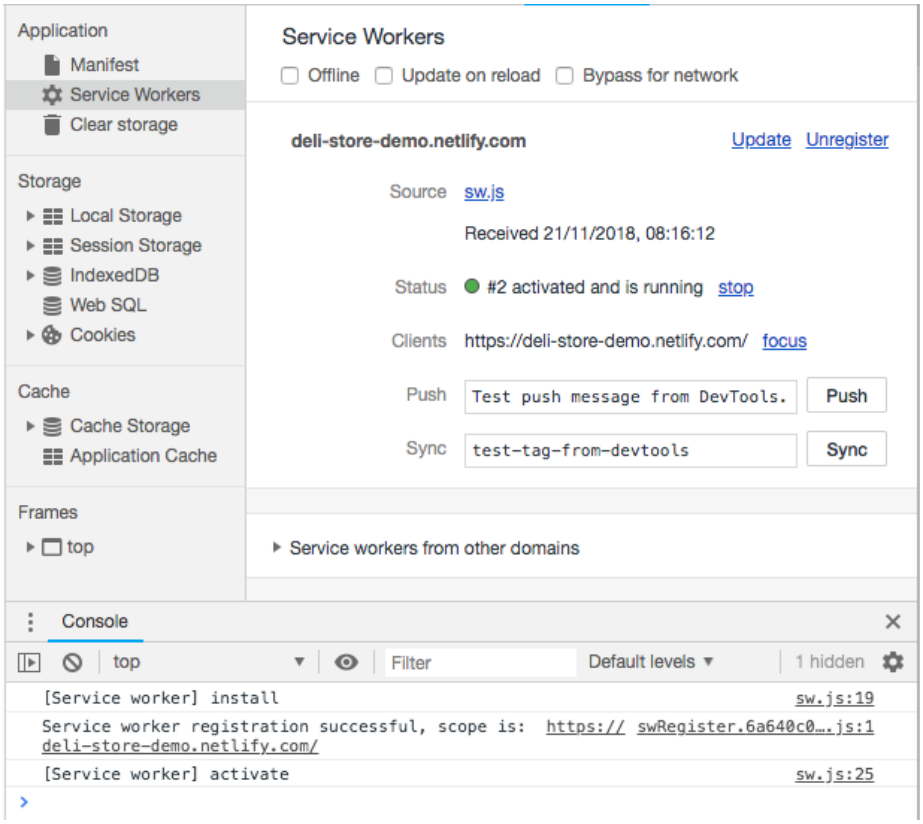


Figure 19. Check service worker status using develop tool

3.6 Result

By following the implementation process, the service worker is successfully integrated thanks to workbox *injectMainfest* plugin. When navigating through the app, the console logs present that service worker works perfectly (figure 20).

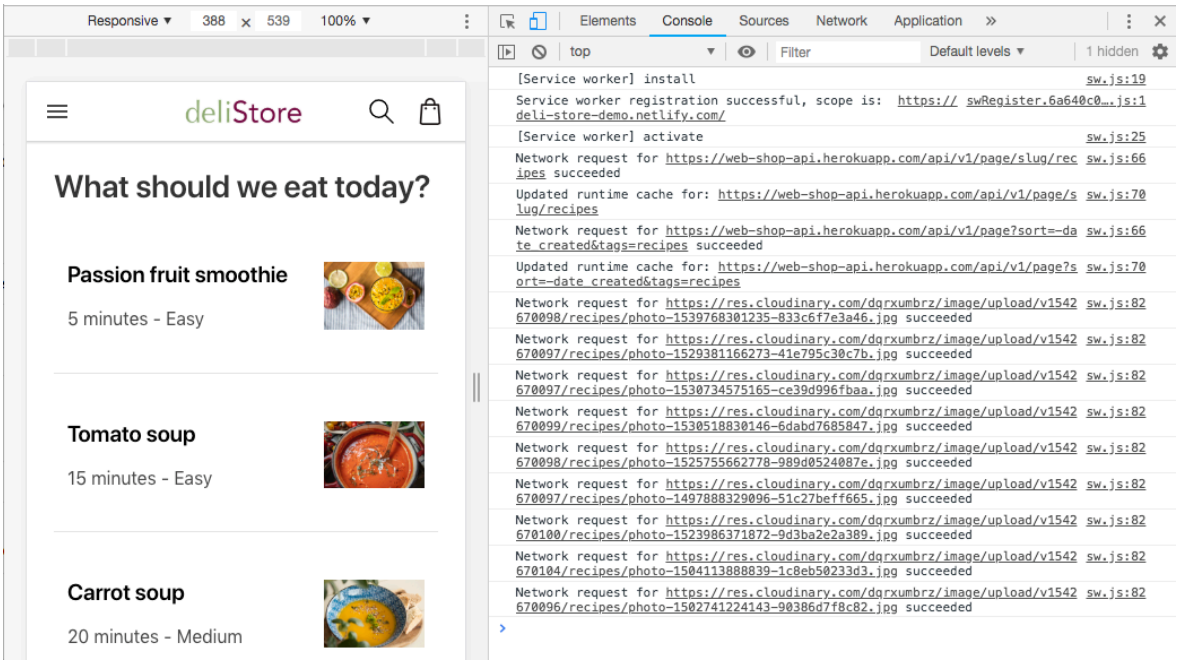


Figure 20. Service worker is working

Through service worker, the challenges of X are resolved. Firstly, the page performance is enhanced (figure 21). Being tested several times with Lighthouse, the page performance has score varies from 75 to 83. As React components in this case study fetch resource from API and CDN servers in *componentDidMount* or *componentWillRecieveProps* methods, service worker is implanted to intercept fetch events and cache responses leading to a shorter duration for page content fully populated.

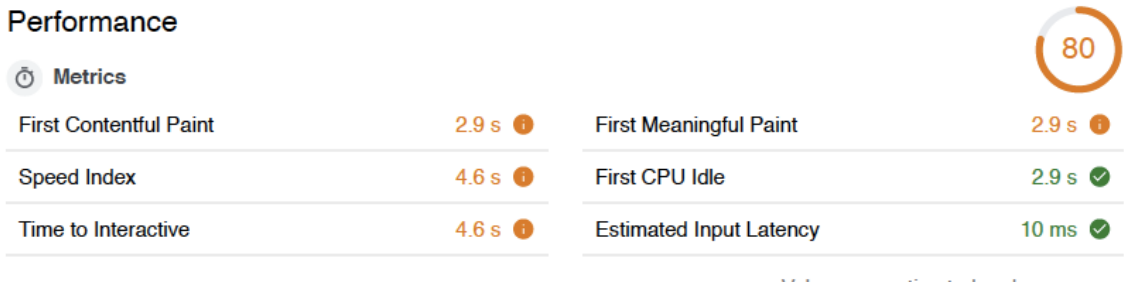


Figure 21. Performance testing using Lighthouse (full report is available at Appendix 2)

Secondly, the webshop X is able to work offline in recipes pages. Users can still access the visited recipes in the webshop, after turning off Internet connection (figure 22). Even when users refresh the page in offline mode, service worker will return precached *index.html* for all navigation routes and navigate users to the right site.

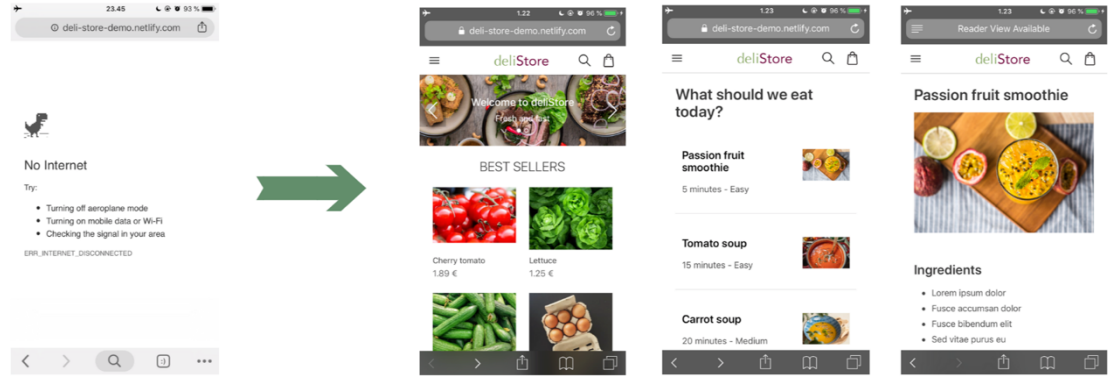


Figure 22. Webshop X in offline mode

In conclusion, thanks to integrated service worker, the webshop X is not only faster but also able to work in offline mode. Consequently, store X enable to provide their customers with a gateway to a better mobile experience, even when they are offline.

3.7 Future development

Though the service worker is implemented successfully leading to the desired results, several improvements should be considered. Optimising assets can enhance the performance. For example, images should be generated into several versions which of each is used for a specific screen size. Moreover, offscreen photos that appear behind the banner slider can be intelligently determined when to load by using *IntersectionObserver*. It allows images to be fetched only when the user scrolls down the page.

Moreover, on the way to turn webshop X meet all requirements of a PWA, web app *manifest* implementation is needed to enable user to access the webshop more quickly. Besides, webshop X should be integrated with push notification function. It helps to engage customers to the webshop and effortlessly send out promotion campaigns.

4 Conclusion

In conclusion, the Progressive Web App with its significant features like web push notification, offline mode and add to home screen ability has immense potential to become a top priority solution in the mobile e-commerce market. With the support of service worker, PWA advocates the business have faster, more reliable and engaging mobile web app.

In addition, the project delivers a webshop with a service worker integrated successfully using a workbox plugin called *injectManifest*. Thanks to the service worker, the webshop is enhanced in performance (from score 56 to 80 out of 100 based on Lighthouse audit report) and available even when there is no internet connection. The webshop has improved to bring a better user experience for mobile users. In the future, the webshop needs to optimise assets to become even faster. Besides, developing the webshop to meet all criteria of PWA should be considered to keep engagement with customers.

From the author's perspective, choosing native app or PWA is still an open question. It depends on the various business contexts. PWAs should be a perfect match for retailers that need a consumer app which provides a seamless experience to their customers. Especially for small businesses which do not have enough resource, building a webshop with integrated service worker seems like a reasonable solution since it requires basis web technology stack (based on JavaScript and ready-made plugins) resulting in short time to research and develop. Otherwise, while many features of the service worker are gradually evolving and shipping, PWA is probably a good choice for large-scale business. However, how efficiently the service worker achieves entirely depends on browser capability. Currently, essential characteristics of the service worker are supported on nearly all common browsers, but it will take a long time for different browsers to deliver full features of the service worker. Additionally, the cache storage which service worker uses to cache content is limited by browser and device. Hence, applying service worker is more suitable for cases that require small cached content. For business who need an app that takes all advantages of the mobile device and a massive deal of cached content, the native app is the answer.

Overall, the author reckons that this thesis has delivered a solid knowledge about PWA and a practical way to learn about service worker. The most challenging issue while conducting the thesis is how to control the service worker to maximize the benefits it brings to the product. Besides, the author has learnt how to choose proper tools and services to develop service worker in a particular case. It is just the early stage of PWA and this new

technology has a very promising future. Hence, getting ahead of the curve will bring more opportunities for people who want a breakout in their professional path.

References

Antonio, C. S. 2015. Pro React. Apress.

Archibald, J. 2018. The service worker life. URL: https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle. Accessed: 21/10/2018.

AudienceProject. 2017. E-commerce in the US, UK & Nordics. URL: https://www.audienceproject.com/wp-content/uploads/audienceproject_study_ecommerce.pdf. Accessed: 9/10/2018.

Behl, K. & Raj, G. 2018. Architectural Pattern of Progressive Web and Background Synchronization. 2018 International Conference on Advances in Computing and Communication Engineering (ICACCE), pp. 366-371.

Borodescu, C. 2018. 2018 State of Progressive Web Apps. URL: https://medium.com/progressive-web-apps/2018-state-of-progressive-web-apps-f7517d43ba70. Accessed: 14/10/2018.

Elliott, E. 2017. Master the JavaScript Interview: What is a Promise? URL: https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-promise-27fc71e7726. Accessed: 1/11/2018.

Gackenheimer, C. 2015. Introduction to React. Apress.

Gambhir, A. & Raj, G. 2018. Analysis of Cache in Service Worker and Performance Scoring of Progressive Web Application. 2018 International Conference on Advances in Computing and Communication Engineering (ICACCE), pp. 294-299.

Gardner, L. D. 2018. Building an Advanced Service Worker. Smashing book 6, pp. 183-233. Smashing Magazine AG. Germany.

Gaunt, M. 2018. Service Workers: An Introduction. URL: https://developers.google.com/web/fundamentals/primers/service-workers/#cache_and_return_requests. Accessed: 20/10/2018.

Google Developers. 2017a. AliExpress. URL: https://developers.google.com/web/showcase/2016/aliexpress. Accessed: 17/10/2018.

Google Developers. 2017b. Lancôme rebuilds their mobile website as a PWA, increases conversions 17%. URL: https://developers.google.com/web/showcase/2017/lancome. Accessed: 17/10/2018.

Google Developers. 2018a. Progressive Web App Checklist. URL: https://developers.google.com/web/progressive-web-apps/checklist#site-is-served-over-https. Accessed: 14/10/2018/.

Google Developers. 2018b. Workbox webpack Plugins. URL: https://developers.google.com/web/tools/workbox/modules/workbox-webpack-plugin. Accessed: 17/11/2018.

Google Developers. 2018c. Lighthouse. URL: https://developers.google.com/web/tools/lighthouse/. Accessed: 17/11/2018.

Green, I. 2012. Web workers. O'Reilly Media. United States of America.

Laudon, K. C. & Traver, C. G. 2018. E-commerce: Business, technology, society. 14th ed. Pearson Education Limited. Edinburgh Gate.

Malavolta, I., Procaccianti, G., Noorland, P. & Vukmirovic, P. 2017. Assessing the Impact of Service Workers on the Energy Efficiency of Progressive Web Apps. 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 35-45.

Malavolta, I., Ruberto, S., Soru, T. & Terragni, V. 2015. End Users' Perception of Hybrid Mobile Apps in the Google Play Store. 2015 IEEE International Conference on Mobile Services, pp. 25-32.

Mali, N. 2018. Your M-Commerce Deep Dive: Data, Trends and What's Next in the Mobile Retail Revenue World. URL: https://www.bigcommerce.com/blog/mobile-commerce/. Accessed: 12/10/2018.

Microsoft. 2018. Progressive Web Apps. URL: https://developer.microsoft.com/en-us/windows/pwa. Accessed: 14/10/2018.

Mozilla. 2018. Service worker API. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API. Accessed: 5/11/2018.

Naylor, I. 2017. Progressive Web Apps vs Native Apps – Who Wins? URL: https://appinstitute.com/pwa-vs-native-apps/. Accessed: 14/10/2018.

Netlify. 2018. Netlify features. URL: https://www.netlify.com/features/. Accessed: 17/11/2018.

Newzoo. 2018. Global mobile market report. URL: https://resources.newzoo.com/hubfs/Reports/Newzoo_2018_Global_Mobile_Market_Report_Free.pdf?submissionGuid=6330db8b-d01e-42cd-bb33-79318b37220. Accessed: 9/10/2018.

Osmani, A. 2018. Getting Started with Progressive Web Apps. URL: https://developers.google.com/web/updates/2015/12/getting-started-pwa. Accessed: 14/10/2018.

Pratskevich, A. 2016. What's a Good App Store Page Conversion Rate? We Asked 10M Users. URL: https://splitmetrics.com/blog/whats-a-good-app-store-page-conversion-rate/. Accessed: 9/10/2018.

Rauschmayer, A. 2018. Exploring ES6. URL: http://exploringjs.com/es6/. Accessed: 1/11/2018.

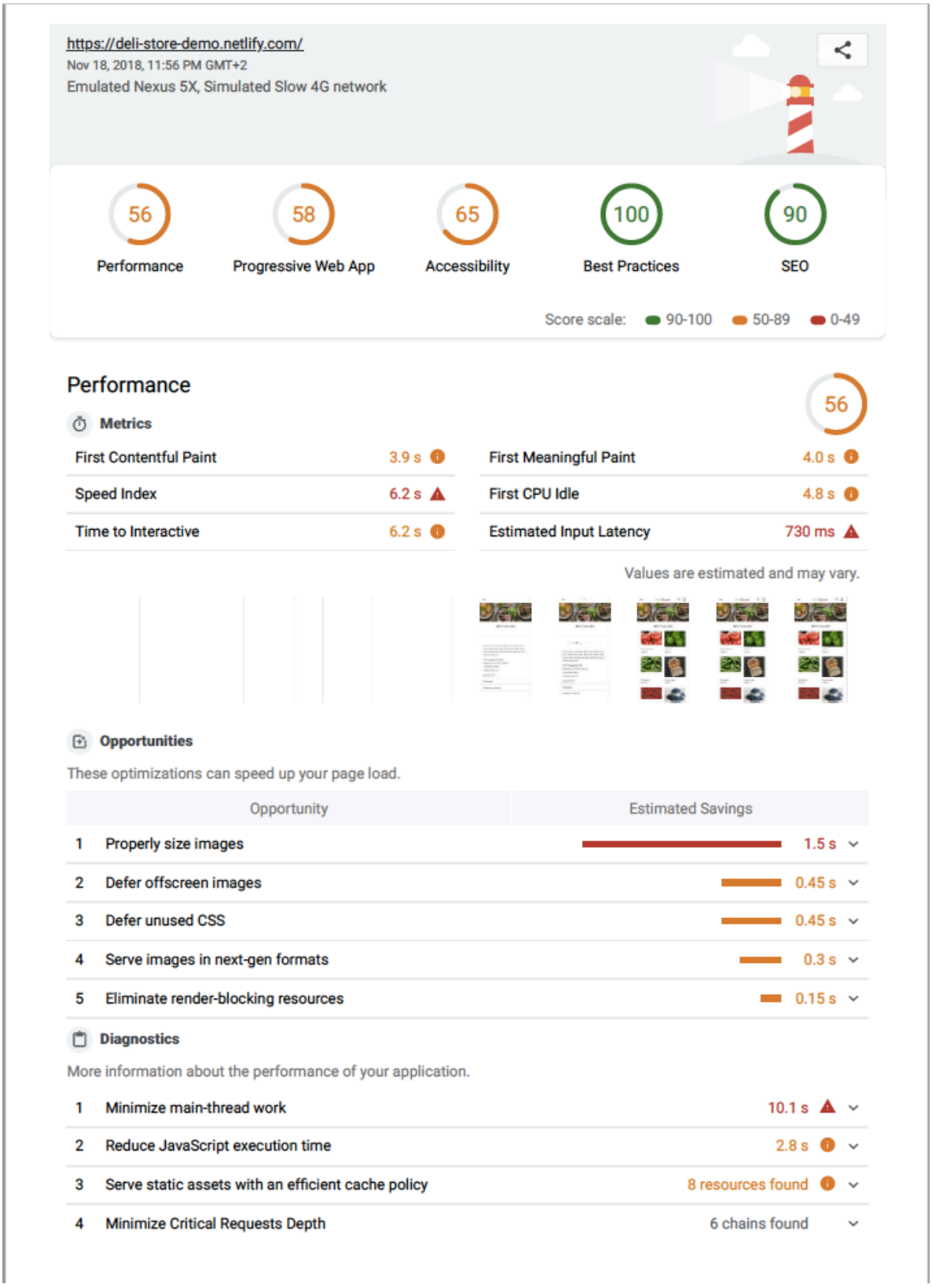
Russel, A. & Berriman, F. 2015. Progressive Web Apps: Escaping Tabs Without Losing Our Soul. URL: https://medium.com/@slightlylate/progressive-apps-escaping-tabs-without-losing-our-soul-3b93a8561955. Accessed: 14/10/2018.

Serrano, N., Hernantes, J. & Gallardo, G. 2013. Mobile Web Apps. IEEE Software, 30, 5, pp. 22-27.

Simpson, K. 2016. ES6 & Beyond. O'Reilly Media. United States of America.

Statistic. 2018. Retail e-commerce sales worldwide from 2014 to 2021 (in billion U.S. dollars). URL: https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales/. Accessed: 9/10/2018.

Appendix 1. Lighthouse report without service worker implemented



Passed audits 13 audits

Progressive Web App

58

These checks validate the aspects of a Progressive Web App. Learn more.

Fast and reliable

1 Page load is fast enough on mobile networks

2 Current page does not respond with a 200 when offline

3 start_url does not respond with a 200 when offline

No usable web app manifest found on page.

Installable

4 Uses HTTPS

5 Does not register a service worker

6 User will not be prompted to Install the Web App

Failures: No manifest was fetched, Site does not register a service worker.

PWA Optimized

7 Redirects HTTP traffic to HTTPS

8 Is not configured for a custom splash screen

Failures: No manifest was fetched.

9 Does not set an address-bar theme color

Failures: No manifest was fetched.

10 Content is sized correctly for the viewport

11 Has a `<meta name="viewport">` tag with `width` or `initial-scale`

12 Contains some content when JavaScript is not available

Additional items to manually check 3 audits

Accessibility

65

These checks highlight opportunities to improve the accessibility of your web app. Only a subset of accessibility issues can be automatically detected so manual testing is also encouraged.

Elements Have Discernible Names

These are opportunities to improve the semantics of the controls in your application. This may enhance the experience for users of assistive technology, like a screen reader.

1 Buttons do not have an accessible name

Elements Describe Contents Well

These are opportunities to make your content easier to understand for a user of assistive technology, like a screen reader.

2 The page does not contain a heading, skip link, or landmark region

3 Form elements do not have associated labels

Color Contrast Is Satisfactory

These are opportunities to improve the legibility of your content.

4 Background and foreground colors do not have a sufficient contrast ratio.

Additional items to manually check 11 audits

Passed audits 17 audits

Not applicable 14 audits

Best Practices

100

Passed audits 15 audits

SEO

90

These checks ensure that your page is optimized for search engine results ranking. There are additional factors Lighthouse does not check that may affect your search ranking. Learn more.

Crawling and Indexing

To appear in search results, crawlers need access to your app.

1 robots.txt is not valid 1 error found

Additional items to manually check 2 audits

Passed audits 9 audits

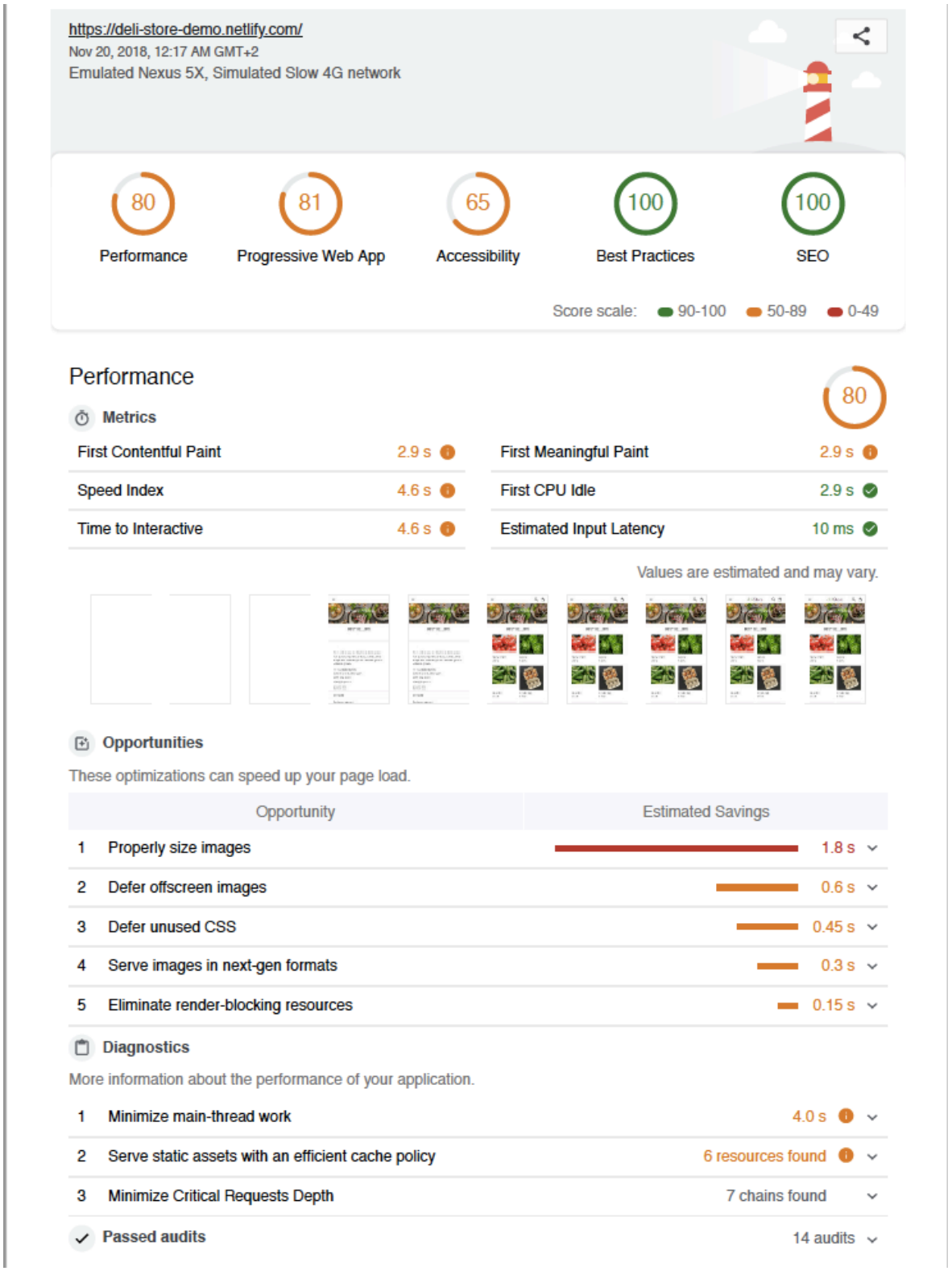
Not applicable 1 audits

Runtime settings

- **URL:** `https://deli-store-demo.netlify.com/`
- **Fetch time:** `Nov 18, 2018, 11:56 PM GMT+2`
- **Device:** `Emulated Nexus 5X`
- **Network throttling:** `150 ms TCP RTT, 1,638.4 Kbps throughput (Simulated)`
- **CPU throttling:** `4x slowdown (Simulated)`
- **User agent (host):** `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.77 Safari/537.36`
- **User agent (network):** `Mozilla/5.0 (Linux; Android 6.0; Nexus 5 Build/MRA58N) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.77 Mobile Safari/537.36`
- **CPU/Memory Power:** `148`

Generated by **Lighthouse** 4.0.0-alpha.1 | File an issue

Appendix 2. Lighthouse report with service worker implemented



Progressive Web App

81

These checks validate the aspects of a Progressive Web App. Learn more.

Fast and reliable

1	Page load is fast enough on mobile networks	✓
2	Current page responds with a 200 when offline	✓
3	start_url does not respond with a 200 when offline No usable web app manifest found on page.	⚠

Installable

4	Uses HTTPS	✓
5	Registers a service worker	✓
6	User will not be prompted to Install the Web App Failures: No manifest was fetched.	⚠

PWA Optimized

7	Redirects HTTP traffic to HTTPS	✓
8	Is not configured for a custom splash screen Failures: No manifest was fetched.	⚠
9	Does not set an address-bar theme color Failures: No manifest was fetched.	⚠
10	Content is sized correctly for the viewport	✓
11	Has a `<meta name="viewport">` tag with `width` or `initial-scale`	✓
12	Contains some content when JavaScript is not available	✓

Additional items to manually check — 3 audits

Accessibility

65

These checks highlight opportunities to improve the accessibility of your web app. Only a subset of accessibility issues can be automatically detected so manual testing is also encouraged.

Elements Have Discernible Names

These are opportunities to improve the semantics of the controls in your application. This may enhance the experience for users of assistive technology, like a screen reader.

1	Buttons do not have an accessible name	⚠

Elements Describe Contents Well

These are opportunities to make your content easier to understand for a user of assistive technology, like a screen reader.

2	The page does not contain a heading, skip link, or landmark region	⚠

3 Form elements do not have associated labels

Color Contrast Is Satisfactory

These are opportunities to improve the legibility of your content.

4 Background and foreground colors do not have a sufficient contrast ratio.

Additional items to manually check	11 audits
Passed audits	17 audits
Not applicable	14 audits

Best Practices

100

Passed audits	15 audits

SEO

100

These checks ensure that your page is optimized for search engine results ranking. There are additional factors Lighthouse does not check that may affect your search ranking. Learn more.

Additional items to manually check	2 audits
Passed audits	9 audits
Not applicable	2 audits

Runtime settings

- URL: `https://deli-store-demo.netlify.com/`
- Fetch time: `Nov 20, 2018, 12:17 AM GMT+2`
- Device: `Emulated Nexus 5X`
- Network throttling: `150 ms TCP RTT, 1,638.4 Kbps throughput (Simulated)`
- CPU throttling: `4x slowdown (Simulated)`
- User agent (host): `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.102 Safari/537.36`
- User agent (network): `Mozilla/5.0 (Linux; Android 6.0; Nexus 5 Build/MRA58N) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.102 Mobile Safari/537.36`
- CPU/Memory Power: `906`

Generated by **Lighthouse** 4.0.0-alpha.1 | File an issue