

Bachelor's thesis

Information and Communications Technology

2019

Minna Kankaala

# ENHANCING E-COMMERCE WITH MODERN WEB TECHNOLOGIES

**TURKU AMK**   
TURKU UNIVERSITY OF  
APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2019 | 41 pages, 6 in appendices

Minna Kankaala

# ENHANCING E-COMMERCE WITH MODERN WEB TECHNOLOGIES

The web has evolved from its early days of static HTML documents to having to serve application-like complicated user interfaces that can even be as powerful as their native counterparts. On the other hand, web browsers have not changed.

Mobile phones are becoming the new default tool for accessing the internet and web browsers are increasingly failing to offer sufficient user experiences. Users are turning to native applications for their everyday online needs at a fast pace, especially in the e-commerce field. Web developers are finding ways to deliver these kinds of experiences via web applications.

The goal of this thesis was to investigate new advancements in the web development field and to utilize them in creating an e-commerce web application. The application implemented in this thesis uses headless architecture, separating backend and frontend logic. This enables the optimization of the frontend part of the application and the creation of a progressive web application. The frontend is built with React and the data used by the application is transferred using the WooCommerce plugin.

This thesis concludes that these modern web technologies can increasingly enhance the user experience of a web page. The list of what the web can do is growing at a fast pace and the differences between native and the web are becoming narrower. The future of web development is hard to predict since in the web development field the only constant seems to be change.

## KEYWORDS:

Web Development, E-commerce, Mobile-First, Headless Architecture, React, JavaScript, PWA

Minna Kankaala

## VERKKOKAUPAN TEHOSTAMINEN MODERNEILLA WWW-TEKNIIKOILLA

Internet on kehittynyt alkuajoistaan, jolloin sen päätarkoituksena oli tarjota staattisia HTML-dokumentteja. Nykyisin se tarjoaa monimutkaisia, melkein natiivisovelluksen kaltaisia käyttöliittymiä. Kuitenkaan työkalut, joita käytämme internetin käyttämiseen, eli selaimet, eivät ole juurikaan muuttuneet.

Mobiililaitteet ovat yleistyneet ja internetiä käytetään pääosin mobiililaitteilla. Verkkosivut eivät usein tuo käyttäjille sellaisia käyttökokemuksia joita he toivoisivat. Käyttäjät käyttävät yhä enemmän natiivisovelluksia ja www-kehittäjät etsivät tapoja tuottaa samankaltaisia käyttökokemuksia myös www-sovelluksilla.

Tämän opinnäytetyön tarkoituksena oli tutkia uusia saavutuksia ja työkaluja www-kehityksen saralla sekä hyödyntää niitä verkkokauppasovelluksen toteutuksessa. Opinnäytetyössä toteutettu sovellus hyödyntää headless-arkkitehtuuria, joka erottelee sovelluksen backend- ja frontend-logiikan. Tämä mahdollistaa frontend-sovelluksen optimoimisen ja progressiivisen www-sovelluksen luomisen. Frontend-sovellus on rakennettu käyttäen React-kirjastoa JavaScriptille ja dataa välitetään WooCommerce-moduulin välityksellä.

Työn johtopäätös on, että modernit web-tekniikat voivat huomattavasti parantaa verkkosivun käyttökokemusta. Lista siitä, mitä verkkosovellukset voivat tehdä, on kasvamassa huimaa vauhtia ja erot natiivi- ja www-sovellusten välillä pienenevät. Www-kehityksen tulevaisuutta on kuitenkin liki mahdotonta ennustaa, sillä ainoa vakio tällä alalla näyttäisi olevan muutos.

### ASIASANAT:

Www-kehitys, Verkkokauppa, Mobiili ensin, JavaScript, React, PWA

# CONTENTS

<b>LIST OF ABBREVIATIONS</b>	<b>6</b>
<b>1 INTRODUCTION</b>	<b>8</b>
<b>2 FROM MOBILE-FIRST TO NATIVE AND HYBRID</b>	<b>10</b>
2.1 History of Mobile-first	10
2.2 Reasons for Mobile-first	11
2.3 Native Applications	12
2.4 Hybrid applications	13
<b>3 PROGRESSIVE WEB APPLICATIONS</b>	<b>15</b>
3.1 Definition of PWAs	15
3.2 Building blocks of PWA	15
3.2.1 The App Shell	16
3.2.2 Service workers	16
3.2.3 Web App Manifest	18
3.3 Improving page load performance	19
3.4 PWAs Benefits and Disadvantages	20
3.5 Native vs. Cross-platform Solutions	21
<b>4 HEADLESS ARCHITECTURE</b>	<b>22</b>
4.1 Reasons for headless architecture	23
4.2 Project application architecture	24
<b>5 BUILDING USER INTERFACES IN COMPONENTS</b>	<b>26</b>
5.1 Component based architecture (CBA)	26
5.1.1 Component independency improves performance	26
5.2 Breaking the UI into a Component Hierarchy	27
5.3 Development tools and helpers for JavaScript applications	28
5.4 React Components	30
<b>6 HANDLING APPLICATION STATE WITH REDUX</b>	<b>31</b>
6.1 Defining the application's state	31
6.2 Actions and Reducers	31
6.3 The Store	32

6.4 Redux Data Flow	33
6.5 When to use Redux	33
<b>7 AUDITS AND FUTURE POSSIBILITIES</b>	<b>35</b>
7.1 Lighthouse audits	35
7.2 Possibilities for the future	37
<b>8 CONCLUSION</b>	<b>38</b>
8.1 Results	38
8.2 The future of web development	38
<b>REFERENCES</b>	<b>40</b>

## APPENDICES

Appendix 1. Finished product in desktop and mobile.

Appendix 2. Lighthouse report of product.

## FIGURES

Figure 1. Flurry State of Mobile 2017 (Flurry, 2018).	12
Figure 2: Lifecycle of service worker on first installation (Google, 2019).	17
Figure 3: Swift VS React Native CPU Usage (Calderaio, 2017).	21
Figure 4: Headless web structure (Koenig, 2018).	22
Figure 5: Project application structure.	24
Figure 6: Application Wireframe.	27
Figure 7: Products in state.	32
Figure 8: Cart in State.	32
Figure 9: Redux Data Flow (Geary, 2016).	33
Figure 10: Lighthouse audits on performance and PWA.	36

## LIST OF ABBREVIATIONS

AJAX	Asynchronous JavaScript and XML is a set of web development techniques on the client side to create asynchronous web applications that can send and receive data without affecting other aspects of the UI.
API	Application Programming Interface is a set of public methods and properties that it uses to interact with other objects in your application.
CBA	Component Based Architecture is a concept in which the user interface is split into separate sections that are independent and have their own interfaces.
CMS	Content Management System is a software application used to create and manage digital content, allowing multiple contributors to collaborate.
DOM	Document Object Model is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style and content. It represents the document as nodes and objects.
JSON	JavaScript Object Notation is a lightweight format for storing and transporting data. Often used when data is sent from a server to a web page.
JSX	JSX is a syntax used by React that allows HTML inside JavaScript code.
npm	Node package manager is a package manager that handles the applications external code dependencies.
React	React is a JavaScript library and a tool for building UI components.

Redux	Redux is a predictable state container for JavaScript apps.
REST	Representational State Transfer is a software architectural style providing standards between computer systems on the web.
PWA	Progressive Web Application is a combination of technologies, that try to provide more high-quality web applications and a better interactive experience, focusing strongly on mobile.
SSR	Server-side Rendering is the ability of a frontend framework/library to render markup while running on a backend system.
UI	User Interface is the space where interactions between humans and machines occur.

# 1 INTRODUCTION

Web has evolved from its early days of static HTML documents to having to serve application-like complicated user interfaces that can even be as powerful as their native counterparts. On the other hand, web browsers, the tools that are used to access the internet have not changed.

Over 52,2% of all internet traffic was generated by mobile devices in 2018 (Statista, 2019). Yet, the average time to fully load the average mobile landing page is 22 seconds. To combine that with the fact that 53% of people will leave a mobile page if it takes longer than 3 seconds to load (MachMetrics, 2018), it is no wonder that there is a desperate need for better mobile experiences.

A large portion of web users have moved onto using native applications for their everyday online purposes. This area has grown especially in the e-commerce field, having increased by 54% only in the year 2017 (Flurry, 2018). Time spent in browsers has significantly decreased in return and web developers are now finding ways to keep up with experiences that native applications offer.

This thesis aims to investigate these modern techniques and methods of web development that aim to tackle the need for better mobile experiences. An e-commerce application will be implemented alongside this thesis, reinforcing the points made in the theoretical sections. The application in this thesis will be implemented with headless architecture, which separates the backend and frontend logic of the application. This allows for building the frontend of the application separately with a JavaScript application rendering the content. To tailor the user experience of the application further, it will be bundled as a progressive web application (PWA).

There are large amounts of publications around modern web technologies. The technologies in this field are constantly changing and evolving, so a publication on this topic does not stay relevant for long. The subject can also be studied from an endless number of viewpoints and technologies.

There are a few recent theses on progressive web applications specifically that touch similar kind of subjects. At the end of 2018, these two theses were published: "Progressive Web App – a new trend in e-commerce" (Quynh, 2018) and "Creating multiplatform experiences with Progressive Web Apps" (Hiltunen, 2018). The former



dives deeper into service workers and how to utilize them in mobile e-commerce. The latter focuses more in different application types and goes into slightly more detail about progressive web applications. This thesis instead focuses more on the implementation of an e-commerce application that utilizes these formerly mentioned technologies.

This thesis is divided into 8 sections. The order of the sections follows the implementation phases of the project developed in this thesis. Section 1 focuses on comparing possibilities in application development and aims to explore them from the aspects of cost, possibilities and limitations. Section 2 dives deeper in PWAs and their building blocks, also introducing tools for improving application performance. In addition, this section aims to investigate the benefits and limitations of web applications. To further address these points, native and cross-browser solutions are compared. Section 4, 5 and 6 follow the implementation phases of the project, from planning of the architecture to concrete examples, explaining further the technologies used to achieve the end result. Section 7 and 8 conclude this thesis with audits and the end result of the application. The final sections also include reflections and discussion on possibilities in modern web development.

## 2 FROM MOBILE-FIRST TO NATIVE AND HYBRID

Mobile-first is a design and content strategy that refers to designing for mobile devices first, then moving onto bigger devices in a progressive fashion. It is a response to the growing needs of improved mobile experiences.

One catalyst to the popularity of mobile-first design was when in the Mobile World Congress in 2010, Google announced that they are going to be moving towards a mobile-first approach and encouraged designers to do so as well: "I think it's now the joint project of all of us to make mobile the answer to pretty much everything" (Schmidt, 2010).

### 2.1 History of Mobile-first

Mobile-first is a tenet of progressive enhancement. The perspective of progressive enhancement is a strategy that evolved from graceful degradation.

**Graceful degradation** is the strategy of maintaining functionality when portions of a system break down. In this perspective, the website is built for the most advanced and capable browsers but allows the page to degrade or remain presentable on older and more incapable technologies. The main idea is to expect poor, but passable experience from these browsers and serve optimal experience on the most capable browsers (Gustafson, 2008).

**Progressive enhancement** evolved as an alternative to graceful degradation. In a way, it is opposite to the former technique, as its focus is on the content. It aims to offer access to the basic content and functionality on a web page, using any browser or internet connection, also providing an enhanced version of the page to those with more advanced browser software or greater bandwidth.

Progressive enhancement leaves it up to the developer to define the baseline of support. It does not necessarily mean that the site or application needs to work without JavaScript, for example (Gustafson, 2008).

To describe the layers of progressive enhancement, an application can be divided into smaller subsets. It needs to start with a foundation and structure, the core. This could be a HTML-only application that would work in any browser or mobile device.

The next layer of the application would be the styling, in this case adding CSS. CSS is progressively enhanced by default, since when a browser does not understand a CSS property, it ignores it.

The final layer would be adding JavaScript. During the implementation of this layer, it is important to consider browser support of language and APIs and to add proper error handling, so the site does not become inaccessible (Gustafson, 2008). As stated previously, it is still up to the developer to set the standards for the baseline of support.

Even if progressive enhancement is an older concept, it is still very relevant as it has the same ideas as mobile-first: the content is the most important part of the site and it needs to be accessible with every device. PWA concepts, such as service workers and push notifications are also ways of progressive enhancement (Tse, 2017).

## 2.2 Reasons for Mobile-first

Mobile-first design improves accessibility and user experience. When designing mobile-first, the content is the focus instead of the visual features that could be implemented on desktop, for example. More focus is put into what the user wants and needs. In addition, as the limitations of mobile networks is considered before implementation, it is likely the loading times will be improved as well, giving access to users in areas that do not have sufficient coverage.

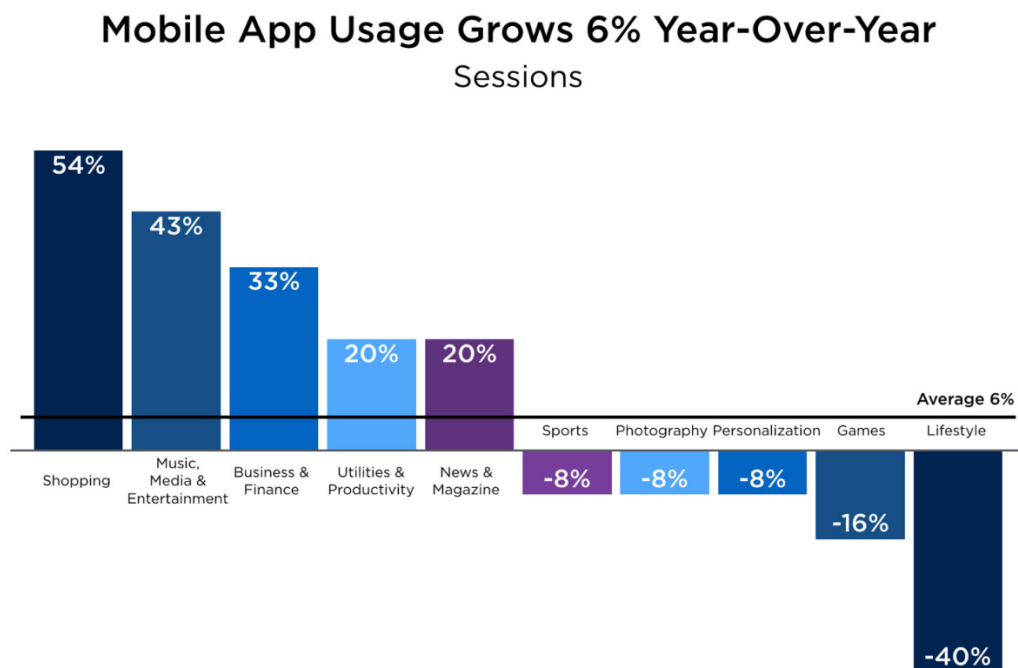
To further reason why it is important to follow the mobile-first concept, in March 2018, Google announced that they would officially start indexing and ranking websites based on the mobile version of the site (Zhang, 2018). Previously, sites were ranked on the desktop version and given extra ranking if there was a mobile design as well. This does not mean that sites that are not mobile-friendly are not ranked at all, but they might see a decline in rankings, especially in mobile search engine pages. Mobile-first index is not a concept created by Google, instead it is the reality of our browsing habits that are now focused massively on mobile.

### 2.3 Native Applications

Mobile usage has increased and so has the usage of native applications. Native applications are programs developed for use on a certain platform or device. Since they are built on a specific device and operating system, they have some clear benefits.

Native applications offer the fastest, most reliable and most responsive experience to users. They can use device-specific hardware and software, for example, push notifications, camera and microphone. Designing for a certain platform is easier and, therefore, the UI/UX can be made to match platform conventions.

Due to their extremely fast loading speeds and offline availability, to name a few, it is no surprise that native applications are taking over time spent in browsers. Figure 1 displays growth of mobile applications in 2017, and as one can see, mobile app usage grows 6% year-over-year (Flurry, 2018).



Source: Flurry Analytics, 2016-2017 Year-Over-Year Session Growth

Figure 1. Flurry State of Mobile 2017 (Flurry, 2018).

More time is spent on apps and less time on browsers, especially in the shopping field. As seen in Figure 1, time spent in e-commerce applications has grown massively (54%) as consumers continue to shift their spending into e-commerce via mobile shopping apps (Flurry, 2018).

Mobile web reach, however, is almost three times that of native apps. Most users are not actively engaged and spend 80% of their time on only their top three apps. Users do not like installing native applications in general: the average user installs 0 applications in a month (Kapoor, 2018). One reason could be that they need to be updated by the user regularly, also taking up space on their devices.

There are also other disadvantages to native applications. They are harder to scale and must be developed independently to each operating system which all have their own codebases. They usually take longer to build than web or hybrid applications and require a team with these kinds of skillsets.

## 2.4 Hybrid applications

Hybrid applications work across platforms and behave like native applications. They are built using a combination of web and native technologies. Technically, they are web applications packed in a native application container. Applications can run on a mobile device and have access to the device, such as the camera or GPS features.

Similar to native apps, they are distributed via a native application store. These applications go through, for example, Apple's and Google's application store review process. Some of the world's most popular applications currently are hybrid, including Amazon and Netflix to name a few. These applications can be developed in a few ways, for example, with frameworks such as Apache Cordova or Ionic or with React Native. (Lastovetska, 2018)

Some clear benefits in hybrid applications developed with **frameworks** are the cost effectiveness and quick delivery. The user experience of an application built with this sort of framework is still not on par with native applications and contain some drawbacks: UX issues including slowness and phantom clicking while scrolling, no offline-availability and depending on the complexity, the application can run relatively slow (Lastovetska, 2018). Capabilities can also be somewhat limited since, for example, the Apache Cordova framework does not support fingerprint scanning by default (Apache, 2015).

**React Native** is one of the newest technologies in the world of hybrid apps. A migration from an existing web app built on React can be implemented easily and the result is a mobile application that uses native components of a smartphone's operating system. Performance is also good, since source code converts to a native mobile app rather than running in a built-in browser window (Lastovetska, 2018).

Like native applications, hybrid applications must be installed on the end devices. To target this audience that do not want to install applications on their devices, developers have created new possibilities in web development, such as the concept of PWAs, which will be further addressed in the next section.

## 3 PROGRESSIVE WEB APPLICATIONS

PWAs are the answer for providing websites with improved user experiences. The user experience is enhanced gradually based on the browser's abilities, hence the name Progressive Web Application. A PWA is a combination of existing technologies and best practices and not a technology itself. The following sections will further explain the concept of PWAs.

### 3.1 Definition of PWAs

A PWA is a web application which makes use of latest web technologies to make a web application feel in a way like a native application and to access some native-like features such as push notifications and offline mode. A PWA will allow one to add a website to their device's home screen. It can be launched as a standalone without the traditional URL bar and other browser features.

In general, for a website to be a PWA, it needs to establish certain requirements. According to Google, there are a few testable requirements that work as a baseline:

- Site is served over HTTPS
- Pages are responsive on tablets & mobile devices
- All app URLs load while offline
- Metadata provided for Add to Home Screen (Web App Manifest)
- First load fast even on 3G
- Site works cross-browser
- Page transitions don't feel like they block on the network
- Each page has a URL

(Google, 2019).

### 3.2 Building blocks of PWA

A PWA should be written ES6, which is a JavaScript language standard. Since the main idea of PWAs is to enhance the mobile experience, it should provide a good layout for all devices without leaving out any content, in other words, follow responsive design

(Google, 2019). Three strongest pillars of PWAs are the application shell, service workers and the web application manifest, which will be investigated in the next sections.

### 3.2.1 The App Shell

The first step is to define the most important aspects of the application as a base, in other words, an app shell. After this, other features can be stacked on it, kind of like in progressive enhancement discussed in Section 2.1.

According to Google's developer documentation: "An app shell is the minimal HTML, CSS, and JavaScript that is required to power the user interface of a progressive web app and is one of the components that ensures reliably good performance" (Google, 2019).

App shell architecture separates the core application and UI from the data. To determine our app shell architecture, the following points have to be considered:

- What needs to be on screen on first load?
- What other UI components are key to our app?
- What supporting resources are needed for the app shell?

After the app shell is defined, service workers can be utilized to cache it. When cached by a service worker, the meaningful components are loaded on the screen on repeated visits without any network.

### 3.2.2 Service workers

A service worker is a JavaScript script that a browser runs in the background, even if the application is closed. It is separate from the web-page and opens a door to features that don't need a web page or user interaction, for example push notifications, offline capabilities and background synchronization.

#### **Lifecycle of a service worker**

For effective use of service workers, it is important to understand their lifecycle. First, a service worker must be **registered** so that it can run in the background. During registration, the scope of the pages that service worker can control can be defined.



A service worker is installed when a visitor opens the web application the first time, if the registration phase of the service worker is successful. On a successful installation, an install event is fired which can be listened to for performing application specific tasks, such as caching data.

If the installation phase is successful, the service worker enters an **installed** “waiting” state. It is not activated automatically and is waiting to take control of the page. It will be activated if there is no service worker currently active, if the `self.skipWaiting()` method is called in the install event handler the script or if the user navigates away from the page (releasing the previous active service worker). On activation, the activate event is fired.

An **activated** service worker has full control of the pages. It can handle events such as fetch, push and background sync. If no events are received, it will enter an idle state and after a while, go into a terminated state. Terminated does not mean uninstalled or unregistered, and it will become idle again as soon as it begins to receive events. A service worker can also become **redundant** if the installing or activating event failed or if a new service worker replaces it as the active one. (Google, 2019)

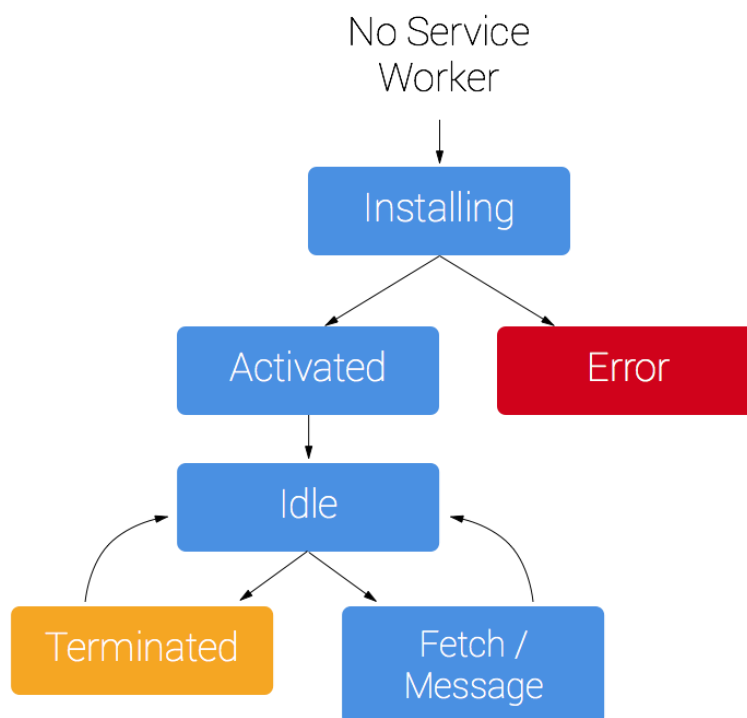


Figure 2: Lifecycle of service worker on first installation (Google, 2019).

## Caching the app shell

As previously mentioned, the app shell can be cached manually using service workers. Below is an excerpt of service worker code that caches static resources from the app shell into the Cache API using service worker's install event.

Code 1. An example of a service worker (Google, 2019)

```
var cacheName = 'shell-content';
var filesToCache = [
  '/css/styles.css',
  '/js/scripts.js',
  '/images/logo.svg',
  '/offline.html',
  '/',
];

self.addEventListener('install', function(e) {
  console.log('[ServiceWorker] Install');
  e.waitUntil(
    caches.open(cacheName).then(function(cache) {
      console.log('[ServiceWorker] Caching app shell');
      return cache.addAll(filesToCache);
    })
  );
});
```

As mentioned, service workers enable native-like features and act a big role in optimizing page load performance, which is another important component of PWAs. This will be further addressed in Section 3.3.

### 3.2.3 Web App Manifest

Finally, a PWA needs to have a web app manifest. The purpose of a manifest file is to provide information about the web application. The JSON structure of that file can contain information such as name, icons and description. This information then can be used to

install the web application to the home screen of a device so that the user can access the application more easily and has an overall app-like existence.

Code 2. An example of a manifest.json file (Google, 2019)

```
{
  "name": "ReactApp",
  "short_name": "ReactApp",
  "start_url": ".",
  "display": "standalone",
  "background_color": "#474747",
  "theme_color": "#474747",
  "description": "Hello World!",
  "icons": [{
    "src": "images/icons/icon-72x72.png",
    "sizes": "72x72",
    "type": "image/png"
  }],
}
```

### 3.3 Improving page load performance

Page load performance is vital in making a successful PWA. As previously stated, loading times impact bounce ratings and the overall user experience tremendously. Making a JavaScript application a PWA does not automatically mean that it is well-optimized and fast. Service workers are one way to decrease loading speeds, but there are other important aspects as well.

#### **Minifying files and lazy loading**

Minifying JavaScript code shrinks file sizes to decrease page load times. It removes things like line breaks, additional spaces, comments to compress the code into a minified file. Images can also use a lot of resources on a site as they usually are large and are easily the single biggest contributor to a page size. Lazy loading is a technique that allows for loading images only when you need them, shortening the initial loading time of a page.

## Workbox

Workbox is a collection of JavaScript libraries that help with service worker related functionalities. It replaces the need for sw-precache and sw-toolbox helpers since it bakes in a set of best practices when working with service workers.

It contains the following functionalities:

- Precaching and runtime caching
- Strategies
- Request routing
- Background sync
- Helpful debugging
- Greater flexibility and feature set than sw-precache and sw-toolbox

(Google, 2019).

### 3.4 PWAs Benefits and Disadvantages

PWAs embrace browsers instead of trying to abstract away from them. They combine the best features there are and bundle them in a tailored application, increasing the user experience of web pages. They don't need to be installed and continuously updated, therefore they don't take any space unlike native applications.

PWAs allow for push notifications and enable apps to work in an offline mode. This is especially useful for sites that have a catalog of some sort, since it does not have to be loaded again and will be available offline as well. This is the case also in the project implemented in this thesis.

Usually applications that follow PWA principles are better optimized in general. They can be developed in a fast pace with traditional web development skillsets. There are many boilerplates for this purpose also, so prototyping is also easy to do.

In the end, they are still technically only websites. In its simplicity, the technique is just a browser wrapper and not a fully-functional application, meaning that the users won't get the same kind of sleek native experience. They also don't have access to all native features, such as GPS or fingerprint scanners – however, the list of what PWA's can access is growing and these kinds of limitations might be removed at some point.

### 3.5 Native vs. Cross-platform Solutions

The future seems to lean on cross-platform non-native solutions increasingly, where the development is done for one codebase only, whether it be hybrid or PWAs.

There are a few reasons for this claim. The main reason is cost. Companies don't want to spend twice as much money making and maintaining the same kind of application in two different code bases in two different languages. That means, in addition to maintaining two code bases, the company would have to have two separate teams with the specific skillsets hired for these tasks. Combining that with the fact that hybrid applications can have almost the same performance speeds as native ones (Calderaio, 2017), it usually does not make sense to pay that high of a difference for a slight improvement in performance.

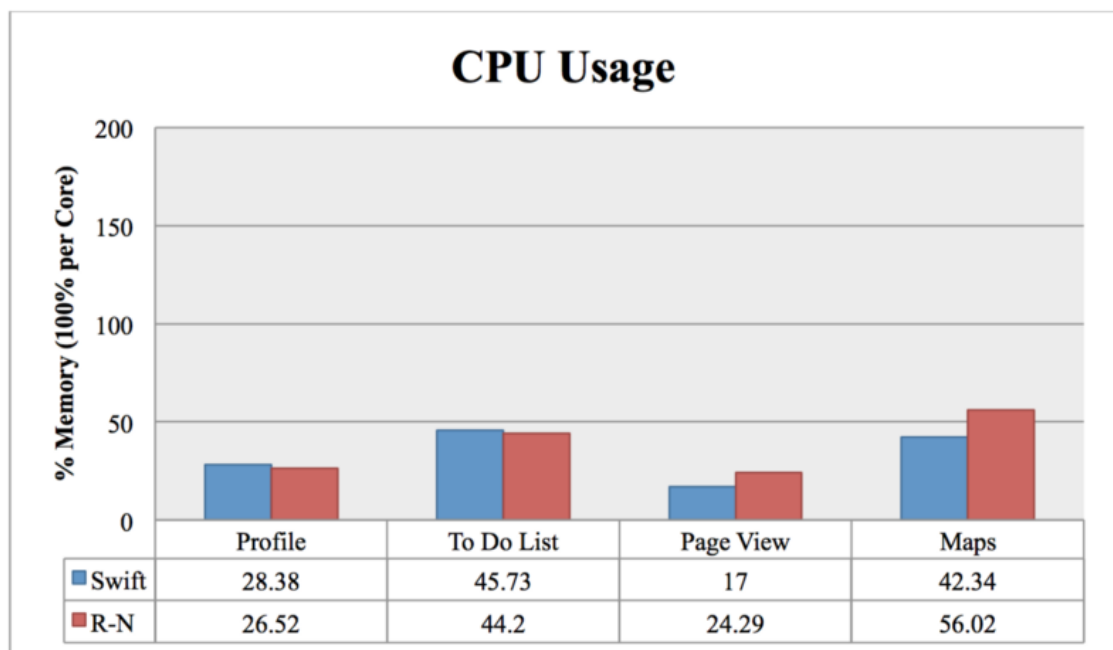


Figure 3: Swift VS React Native CPU Usage (Calderaio, 2017).

The performance aspect made more sense a few years ago, when phones were not yet that powerful. Some graphical drawbacks still exist, so in those cases it could make sense to go for Native. There are also still cases where the application needs to access some hardware that the cross-browser solutions can't access. That is also the use case where one should go for native applications.

## 4 HEADLESS ARCHITECTURE

Headless architecture refers to a situation where a database-driven CMSs content is accessible via a web-service API and then the end-user experience is delivered by a JavaScript application rendering the output of the API.

This adds an additional layer between the content and display, giving developers greater flexibility for innovation over traditional monolithic CMSs. Monolithic CMSs combines everything required for managing and publishing content to the web, meaning that it is an all-in-one content-management solution (Heslop, 2018). Some CMSs, like WordPress and Drupal contain JSON REST API's as a built-in feature, which enables the API-driven headless architecture easily.

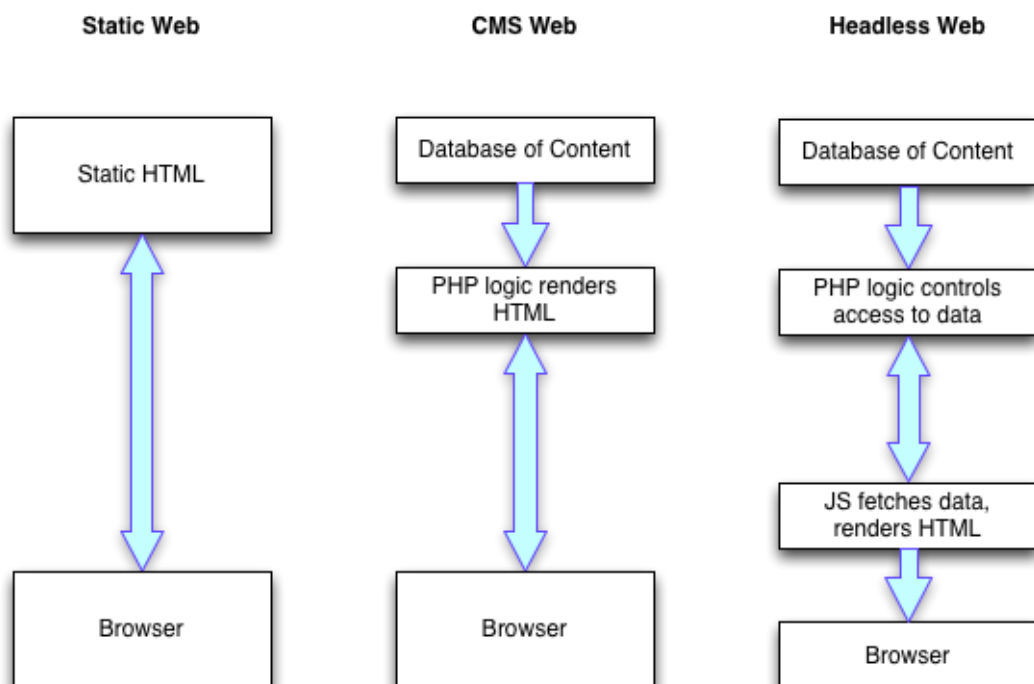


Figure 4: Headless web structure (Koenig, 2018).

Headless and decoupled are often mentioned in the same context. A decoupled system consists of two or more systems that can interact without being connected, in a similar way that the HTML content file is separated from the CSS formatting file and the JavaScript programming file, for example. Typically, a decoupled CMS includes some

type of front-end, whereas a headless CMS does not. Headless is API-first, meaning that it integrates content management tools with an API (Heslop, 2018).

#### 4.1 Reasons for headless architecture

Headless architecture provides wide flexibility in presentation. Presentation can be handled in a variety of ways, for example with interactive JavaScript frameworks like React. Even more tailored experiences can be achieved by packaging the application as a PWA. The frontend application can be constructed from scratch, without worrying about the backend structure, which can even help with aspects like accessibility. Multiple frontends can also coexist peacefully for the same backend.

Optimization is also a key factor, since display logic can be shifted to the client-side. This helps with streamlining the backend, since the backend does not have to focus on rendering HTML templates, only on serving the data.

Delivery is quicker as updates and development can be done separately and asynchronously to both systems. Headless also enables for easily combining various services or content points into one display.

Scalability is a big factor. A headless CMS can scale and to avoid database bottlenecks that are common in the uses of monolithic CMS. Most headless CMS offerings are cloud-hosted, so it is also possible to automatically adjust the cloud infrastructure to match demands (Heslop, 2018).

The performance of content delivery can be even further improved by enabling CDN in the headless CMS. CDN (Content Delivery Network) is a network of servers spread across the globe which caches static assets and dynamic content of CDN-enabled websites. Websites then retrieve the cached content from their closest CDN server on requests and deliver it to the client (Heslop, 2018).

Like any technology, the model does have trade-offs. For example, CMSs have gained a lot of features over the years that have to be re-implemented such as user permissions and routing-related issues (Kraft, 2018). Headless is not the answer to all issues but the popularity of it amongst developers does seem to be justified because of the benefits it brings.

## 4.2 Project application architecture

Due to previously mentioned benefits and the intention of following PWA concepts, the client project is implemented using headless architecture as the diagram in Figure 5 describes.

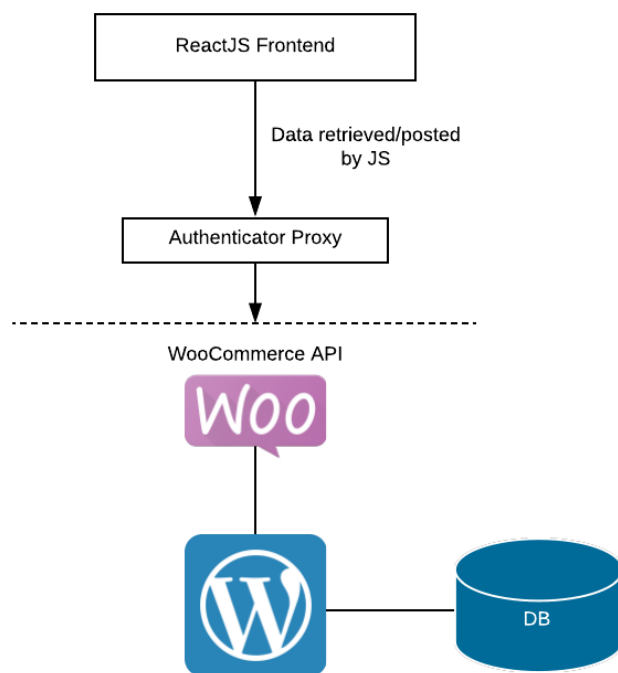


Figure 5: Project application structure.

### Backend

The backend is implemented with WordPress together with WooCommerce. WooCommerce is a popular e-commerce plugin for WordPress.

This setup was chosen because of a few reasons:

- WordPress has a built-in REST API
- WooCommerce has an API that is fully-integrated with the former
- WordPress and WooCommerce together provide a very user-friendly content editing experience, which was very important for this case in point



## API Layer and Proxy Server

WooCommerce API allows WooCommerce data, such as products, to be created, read, updated and deleted using requests in JSON format. Between the backend and frontend layer is a simple proxy server for authenticating requests to WooCommerce API, since the API did not serve a safe out of the box solution itself. It also strips out the unnecessary and harmful data that shouldn't be exposed to the outside. The server was built with Python 3.6. together with Flask and WooCommerce API wrapper for Python.

Flask is a Python web framework (Makai, 2019). An example of an up and running application in Flask:

Code 3. An example application in Flask (Makai, 2019)

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

After deploying the project, the Flask server can be launched with the mod\_wsgi package. The mod\_wsgi package implements a simple to use Apache module which can host any Python web application (Dumpleton, 2018).

The application's domain will then be pointed to the server. Apache catches the call and depending on the subdomain, the correct page will be called. By default, the domain will be pointed to the React frontend, which is a separate implementation.

## Frontend

The frontend is built as a JavaScript application, with the React library. The React application fetches the data it needs from the proxy's endpoints, for example '/products'. The endpoint returns the data in JSON format, containing properties defined in the WooCommerce API documentation. These calls are implemented in Redux actions, that is discussed in Section 6.

## 5 BUILDING USER INTERFACES IN COMPONENTS

This thesis uses React to build the frontend of the application, which is a JavaScript library for building user interfaces. This thesis however is not opinionated on a certain JavaScript library. React is clearly the most popular JavaScript library amongst developers currently, Angular coming in second and Vue.js third. (Benitte R., Greif S., Rambeau M., 2018) Vue.js is growing at a rate which some say will surpass Angular and maybe even catch up to React.

The following sections will dive deeper into building the UI with React. The development setup is described, and some necessary tools are introduced. Some main principles of React are discussed. The components discussed in this section are React components specifically.

### 5.1 Component based architecture (CBA)

Component based architecture as a concept was widely introduced when Facebook released React in 2013. It introduced a method for encapsulating individual pieces of a larger user interface (UI) into self-sustaining and independent micro-systems, otherwise known as components.

Components coexist in the same interface but have their own structure, own methods and own APIs. Due to their independent nature, they are highly reusable, which allows developers to create UI's with moving parts and to reuse features with ease.

The reason why this isolation of sections is beneficial can be demonstrated with the following example: think of the Facebook front page when a user is logged in. There are sections for the post feed, friends list and navigation. Long before, each time a section updated the whole page would have to be re-rendered, meaning that if the feed updated, the friends list and navigation would have to reload as well.

#### 5.1.1 Component independency improves performance

AJAX (Asynchronous JavaScript and XML) requests are a concept that forms the basis for component-based thinking. In AJAX requests, calls to the server are made directly

from the client-side, allowing the DOM (Document Object Model), in this case page, to be dynamically updated without the need of refreshing the page (MDN, 2018). Each component has their own interfaces, that can make calls to the server and update their interfaces, without affecting other components or the UI.

React handles components in a performant way, since it uses something called a virtual DOM instead of the traditional DOM. The virtual DOM only exists in-memory and is a representation of the browser's DOM. After the virtual DOM updates, React intelligently detects changes to a component and only renders those to the actual DOM, as opposed to re-rendering the entire component or DOM (Shapiro, 2016).

## 5.2 Breaking the UI into a Component Hierarchy

Creating a wireframe, a skeletal model helps on visualizing the elements, and therefore components, that will be displayed on the user interface.

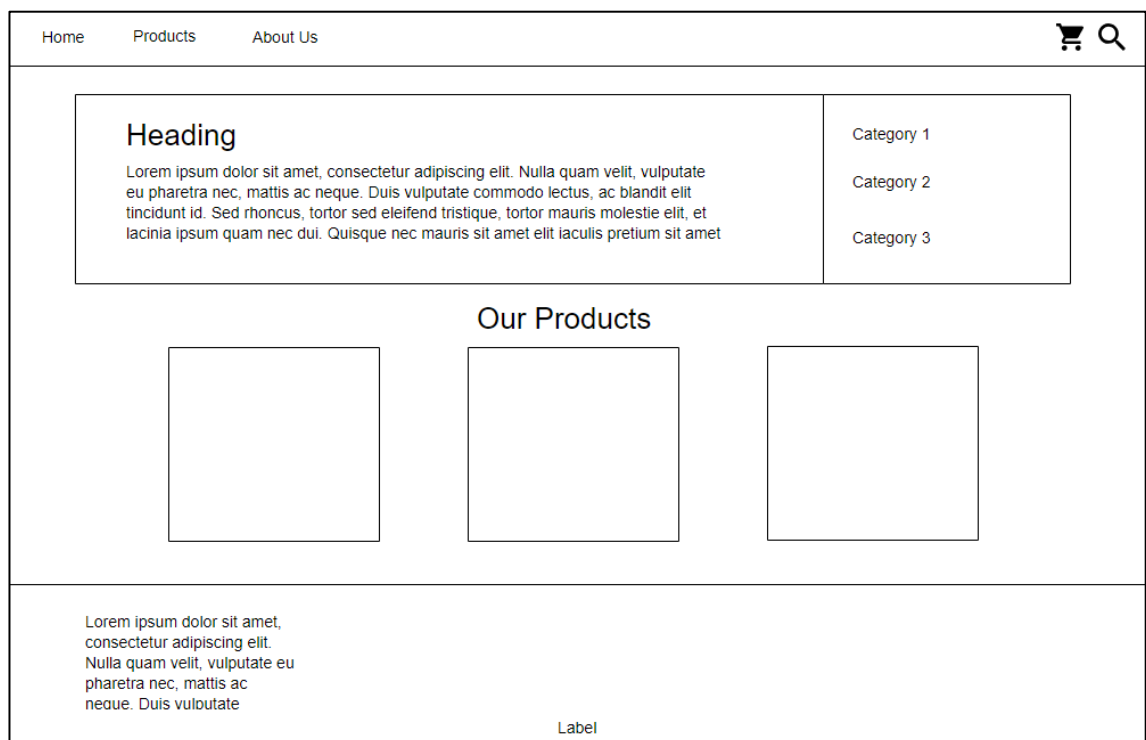


Figure 6: Application Wireframe.

From the wireframe in Figure 6, different sections can be separated to individual components. A technique for deciding what should be its own component is using a single responsibility principle, meaning that a component should ideally do one thing and if it does not or ends up growing, it should be decomposed into smaller subcomponents (Facebook, 2019).

From the former wireframe, the following components can be identified: Navigation, Cart, Search, Content building block, Product listing, Product card, Footer.

Now that the components are identified, they can be arranged in a hierarchy:

- Navigation
  - o Cart
  - o Search
- ContentBlock
- ProductListing
  - o ProductCard
- Footer

### 5.3 Development tools and helpers for JavaScript applications

#### **Style guide and linter**

This thesis used a JavaScript style guide created by AirBnB. It is a popular style guide amongst developers due to it being well-maintained, comprehensive and enforced via ESLint. ESLint is a tool for linting your code, it analyzes and gives warnings of potential errors and linting mistakes.

These helpers make reading and maintainability a lot easier, since they add a lot of clarity to the code. It is always a smart idea to improve code clarity, since it can become difficult to maintain large codebases since it is easy to forget the purposes of written code.

#### **Package Managers**

A package manager lets one manage the external code dependencies of the application. Two popular package managers for JavaScript applications are npm and yarn, the latter increasing in popularity recently.

This thesis uses npm for package management. It downloads libraries and puts them in a folder called `node_modules` so they can be easily be found by the node runtime.

Two important files to keep track of the project's dependencies are `package.json` and the lock file. The former contains all the dependencies for a project and the latter contains all the information needed to reproduce the full dependency source tree.

"To make it more clear, your `package.json` states "what I want" for the project whereas your lockfile says "what I had" in terms of dependencies." (Abramov, 2016)

### **Babel**

Babel is a JavaScript compiler. It is mainly used to convert code ES6 into a backwards compatible version of JavaScript in current and older browser environments. It can for example transform syntax, provide polyfills for features that are missing in the target environment and perform source code transformations. It also has built-in support for React's JSX syntax (Babel, 2018).

### **Webpack**

Webpack is a JavaScript module bundler. It can take care of bundling alongside a separate task runner. It internally builds a dependency graph which maps every module your project needs and generates one or more bundles. (Webpack, 2019)

### **Create React App**

Create React App is a tool for creating React apps with no build configuration. It requires the following commands to create and start a fully configured React application:

```
npx create-react-app my-app  
cd my-app  
npm start
```

This means that both Babel and Webpack are preconfigured and hidden so developers can focus on coding instead of managing configurations (Facebook, 2019).

## 5.4 React Components

A component is a JavaScript class or function. It can accept inputs like properties. The component returns a React element that describes how a section of the UI should appear.

React uses JSX syntax, that allows HTML inside JavaScript. It is the same way one would write a `React.createElement()` declaration. It outputs a tree of React elements, that are plain objects, which is a visual representation of the HTML elements this component outputs. (Kagga, 2018)

**Functional components** are purely presentational and are represented by a function that optionally receives properties and returns a React element. They aren't aware of state, so they are referred as stateless components. In their most simple form, they can look something like the following code:

```
const HelloWorld = () => <h1>Hello World!</h1>;
```

They are predictable and concise and are suggested to be used over class components whenever possible.

**Class components** are created using ES6's class syntax. They are referred to as smart components, since they can contain logic. They can hold and or manage state. In their simplest form, they can look something like the following code:

```
class HelloWorld extends Component {  
  render() {  
    return <h1>Hello World!</h1>;  
  }  
}
```

In the past, functional components were meant only to output UI elements. React introduced hooks in version 16.8.0 so that state and other React features could be used without a class. Some of the motivation behind hooks was that they allow reusing stateful logic without changing component's hierarchy and simplifying components in general: class components require a lot more markup and using them unnecessarily can affect for example performance, code readability and maintainability. (Facebook, 2019)

## 6 HANDLING APPLICATION STATE WITH REDUX

Redux is a state management tool for JavaScript applications. It is founded on the following principles: The state and logic of your application is kept in a store and any state each component needs from that store, they can access. To update the store, an action describing the state changes must be dispatched. That action will **replace** the state, meaning that the application state is immutable. Reducers create the next state, given the current state and an action. (Geary, 2016)

### 6.1 Defining the application's state

In order to implement state management in the project, it needs to be defined what data needs to be moved or changed in the user interface.

In the application mentioned in this thesis, the following data and logic was required:

- Products need to be loaded from a backend service via WooCommerce API
- Individual products need to be filtered from backend service by ID
- Products need to be filtered by category and sorted by price
- Cart contents need to be fetched and persisting, user needs to be able to add and remove items to and from cart

### 6.2 Actions and Reducers

The only way to change the state tree is to emit an **action**. An action is an object describing what happened, a payload of information that sends data from your application to your store.

Actions only describe what happened and not how it changes the application's state. This is where **reducers** come into place. They specify, how the actions transform the state tree. When building reducers, it is important to think about the structure of the state tree. (Geary, 2016)

### 6.3 The Store

The Store is the object that brings actions and reducers together. The store has the responsibilities of holding the state, allowing access to the state, to register and unregister listeners (Geary, 2016). There should only be a single store in a Redux application.

After the actions and reducers are defined and the store is created, the data can be seen in the state tree. The following figures are excerpts from the application created along this thesis. In this example, the state contains products and the cart items and their data.

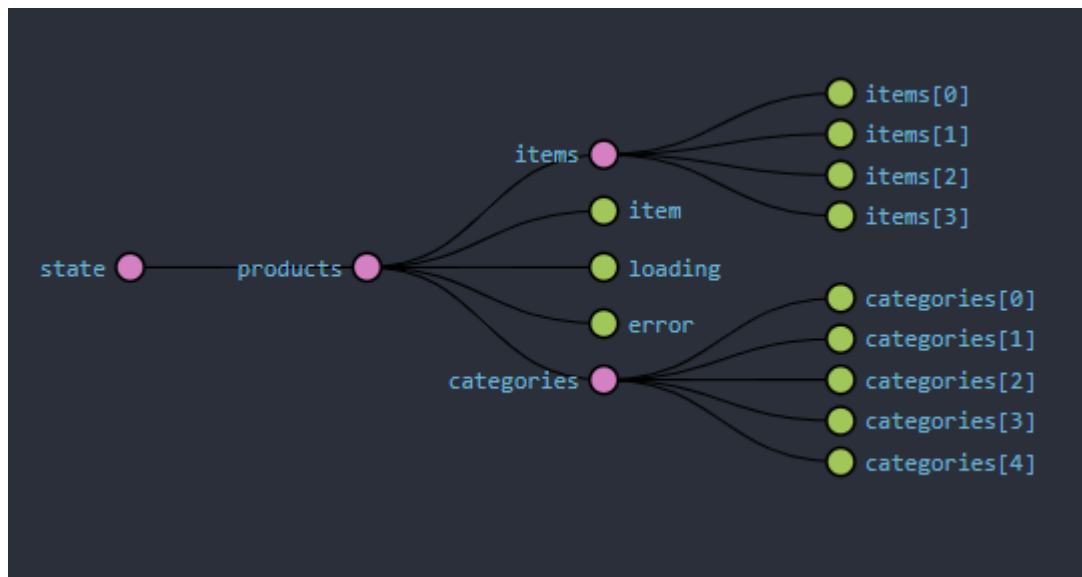


Figure 7: Products in state.

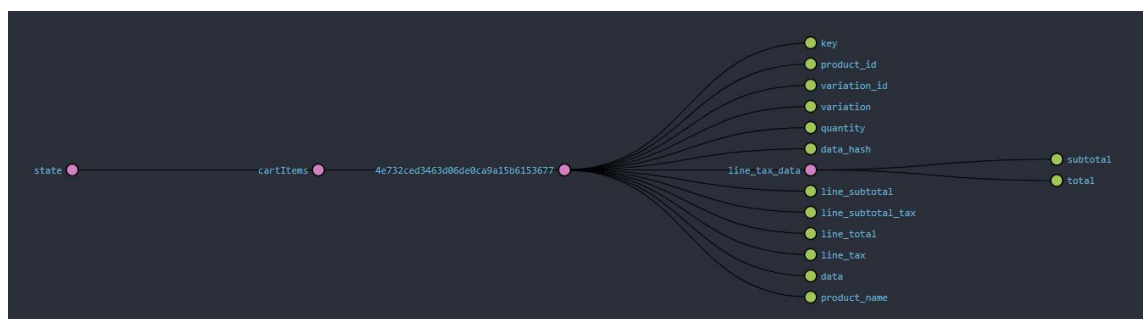


Figure 8: Cart in State.



## 6.4 Redux Data Flow

Redux architecture revolves around a strict unidirectional data flow, meaning that all data in an application follows the same lifecycle pattern.

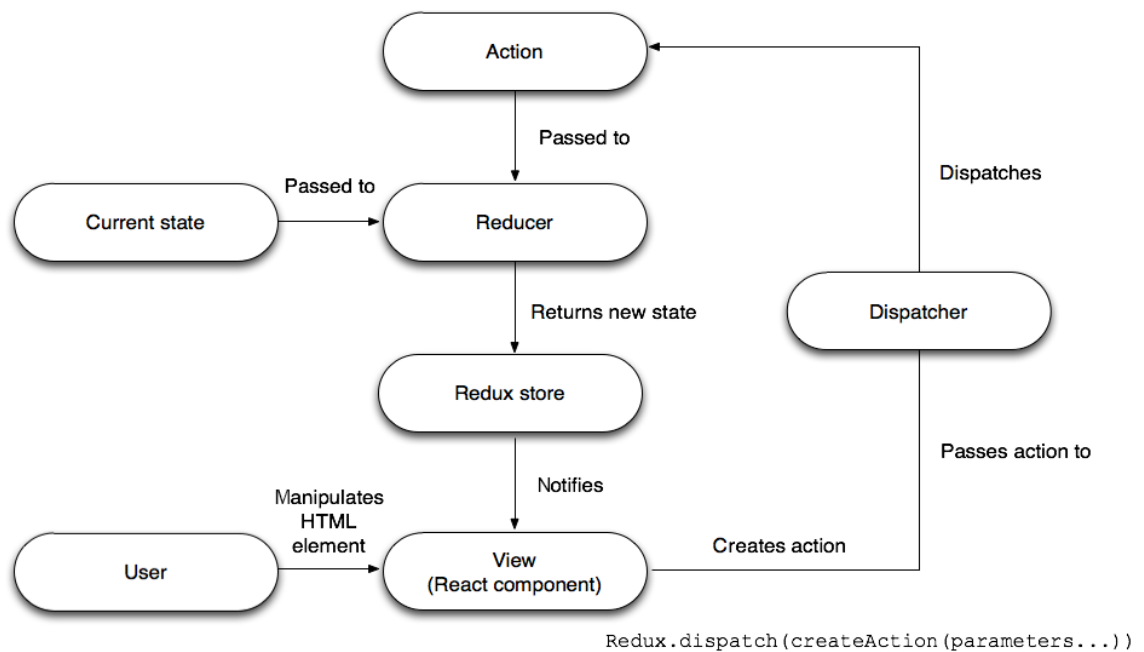


Figure 9: Redux Data Flow (Geary, 2016).

As demonstrated on Figure 9, the data lifecycle follows these four steps:

- A dispatch action is called
- Redux store calls the reducer function it has been given
- The root reducer may combine the output of multiple reducers into a single state tree
- The Redux store saves the complete state tree returned by the root reducer

## 6.5 When to use Redux

JavaScript libraries such as React and Angular, are built with a way for components to internally manage their state without any need for an external library or tool. (Ighodaro, 2018)

Managing state between lots of components can become difficult. The ideal way would be that the data in a component should live in just one component. Thinking of sharing data between sibling components: the state would have to live in the parent component, which also provides a method for updating the state. The parent component passes the state down as props to the sibling components.

This former is still relatively simple, but when state must be shared between components that are far apart in the component tree, the state must be passed from one component to another until it gets to where it is needed. This is called **prop drilling**. Prop drilling is not necessarily bad and can be considered a good practice, however, it can be an issue in especially the process of refactoring. (Ighodaro, 2018)

Luckily, the new React's Context API (released in version 16.3.0) provides a solution for this prop drilling issue. State management libraries, like Redux, use the exact same context feature that has before been labeled experimental in React and now released.

State handling can perfectly be done without adding additional libraries like Redux. Even one of the creators of Redux, Dan Abramov, suggests approaching Redux with caution, just like any highly opinionated tool (Abramov, 2016).

However, there are several benefits to using Redux as it can allow some more complicated actions, such as: maintain an undo history, persist state to local storage and restore it after a refresh and server-side rendering. Server-side rendering in Redux can be done by handling the initial render of the app by sending the state of an app to the server along with its response to the server request. (Ighodaro, 2018)

While it was not actually necessary due to the previously mentioned points, this thesis implemented Redux for the following reasons: It offers predictable state. The state is also immutable and is never changed. Redux is easy to maintain and debug – the structure and order of code is important and logging actions and state helps to see errors and possible bugs. In real-life projects it is important to try to avoid reaching for these kinds of tools right away, since often they are tradeoffs. So, remember: "If you trade something off, make sure you get something in return" (Abramov, 2016).

## 7 AUDITS AND FUTURE POSSIBILITIES

Website audits are essential when wanting to maximise the benefits of websites. There are multiple parts that a website audit can or should include, such as technical audits and SEO audits. The following sections include an audit for the finished application and reflections based on the audit results.

### 7.1 Lighthouse audits

A helpful tool for performance and PWA testing is Lighthouse. It is an open-source, automated tool for improving the quality of web pages (Google, 2018). It includes checks for performance audits and PWA features amongst others.

Lighthouse suggests ways to improve the application and gives tips on possible errors. The application in this project was audited with Lighthouse on performance and PWA.

Lighthouse looks at the following points regarding performance:

- First meaningful paint (when the main content of the page is visible)
- Speed index (average time at which visible parts of the page are displayed)
- Estimated input latency and time to interactive

(Irish, 2016).

Regarding PWA features, Lighthouse checks for Google's PWA requirements that were listed previously in Section 3.1.

Lighthouse audits were run against the local build, in the local environment. The performance may vary from what it would be on a server, since some optimizations can also be done there. These would be, for example, configuring the web server and enabling PHP accelerators.

The results can be seen in Figure 10. The audit in this case is carried out without throttling. As can be seen, the performance and PWA results look promising: a 100 for performance and 92 for PWA. Only criteria that is not fulfilled as a PWA is the criteria of being served over HTTPS. In this case it can be ignored, since it will be served via HTTPS after deployment.

There are a few warning messages for performance about web font loading and efficient caching of resources. The number of requests seems to be too high, which can have something to do with firing them unnecessarily in React components. This would have to be further investigated in component-level.

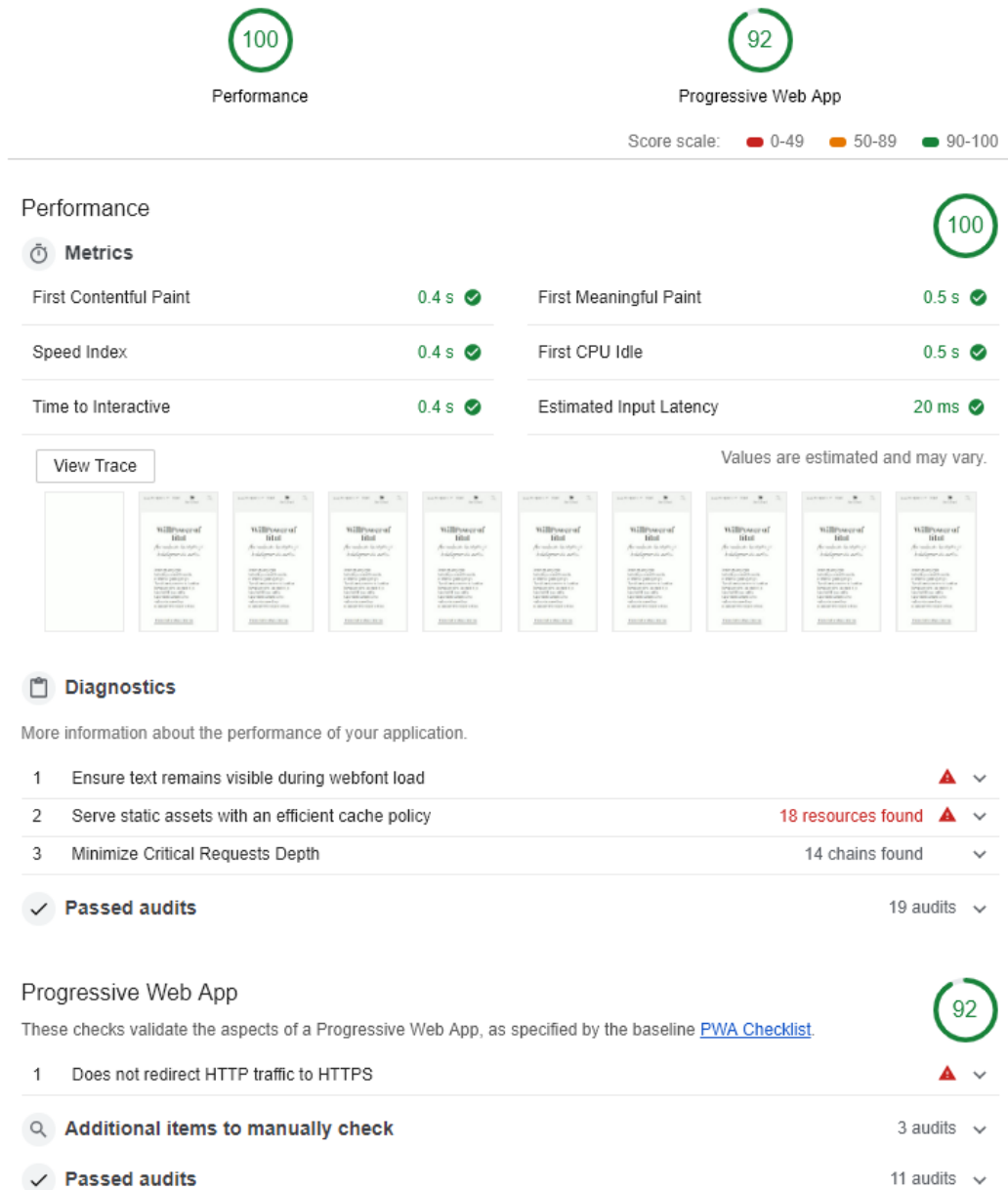


Figure 10: Lighthouse audits on performance and PWA.

When the same audits were carried out in throttling mode, the previous warnings significantly decreased results in performance. Images were a huge factor, since they were missing optimizations, such as lazy loading and packaging them into smaller files. The issues in performance will be fixed before deployment of the application. Throttling did not affect the PWA results of the audit. The audit performed in throttling mode can be found attached in Appendix 2, along with metrics for other aspects as well.

## 7.2 Possibilities for the future

There is a huge number of tools for web developers to use that could also be utilized in this project. One great example of this could be server-side rendering (SSR). SSR is a complicated topic that can be approached from many angles. As previously mentioned, it could be implemented with Redux by handling the initial render of the app by sending the state of an app to the server along with its response to the server request (Ighodaro, 2018). It can also be implemented by configuring a server to serve static files from your React application (Moldovan, 2018).

The reason why SSR would be useful are factors such as SEO and speed since the idea is to render the app on the server initially, then to leverage the capabilities of a JavaScript application on the client (Moldovan, 2018). It is a considerable investment as it can become complicated, especially in larger applications. The investment may not always be worth it compared to the benefits.

Static site generators, such as Gatsby and Hugo, could also enhance the e-commerce experience even further. In simplicity, static site generators take source files and generate an entirely static website. This means that the hosted page runs without a backend, improving aspects like speed and security of the website.

## 8 CONCLUSION

This thesis only touched the surface of possibilities in modern web development. The following conclusions can be drawn from the results of this thesis: User experiences in the web can be improved tremendously with new technologies in web development. In addition to the user experiences, developers are offered better development experiences as well. The results of this thesis are summarized in the next section.

### 8.1 Results

A headless WordPress was used together with React to implement a PWA. Mobile-first principles were followed throughout the project. The development of the application continues, since it still uses test data and some styling improvements need to be made. Some optimization efforts will also have to be re-visited. After these issues and thorough testing, it will be deployed online.

Alongside with the implementation of the application, modern web technologies were investigated from different angles, mostly around the technologies used in the project. To better understand the need for improved web technologies, a comparison was made between cross-platform and native applications. It was found that there is clearly a market for PWAs and they are becoming increasingly popular, along with other cross-platform solutions such as hybrid applications.

Implementing a headless CMS with a JavaScript frontend seems to have many benefits, for example, the presentation of the application can be tailored much further than in traditional CMS architectures. The backend is streamlined, enabling better performance. The approach can also improve factors such as accessibility, scalability, and time of delivery. Additionally, bundling the application as a PWA can increase the user experience tremendously.

### 8.2 The future of web development

Browsers have existed for almost 30 years, yet the concept of them has not changed. They were designed for simple HTML files, not complex applications. They have kept

evolving and trying to keep up with the new requirements brought by, for example, the rising popularity of mobile devices.

React has already abstracted the DOM away by utilizing a virtual DOM. This means that to React, the browser only is a render target. Maybe this was because React developers did not want to be dependent on the browsers since the future of them is unknown. Only future will tell whether more technologies will take the same approach, or whether there will be an innovation that will overtake browsers completely. In any case, JavaScript is and has been a giant in web development and will probably continue to be for the next years to come.

For now, next in the horizon seems to be at least artificial intelligence (AI), IoT and previously mentioned static site generators. AI especially seems to have generated a great deal of publicity. Even the CEO of Google, Sundar Pichai, called AI “one of the most important things that humanity is working on”, saying that it is “more profound than electricity or fire.” (CNBC, 2018)

The saying “the only constant is change” seems to define the industry well. The scope of what web developers can do is huge, and it is ever-growing. New technologies and trends are coming out every year and it can be hard to stay on track, especially for newcomers in the field.

To conclude this thesis, a fitting quote by Ryan Kavanaugh: “Innovation – it’s such an easy principle: adapt or die, but our industry has always been scared of it (...) The key is to embrace disruption and change early. Don’t react to it decades later.” (Kavanaugh, 2014)

## REFERENCES

- Abramov, D. (2016). You Might Not Need Redux. (Referenced on 13.2.2019) [https://medium.com/@dan\\_abramov/you-might-not-need-redux-be46360cf367](https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367)
- Airbnb. (2012). JavaScript Style Guide. (Referenced on 13.2.2019) <http://airbnb.io/javascript/>
- Apache Cordova. (2015). Apache Cordova Documentation. (Referenced on 17.2.2019) <https://cordova.apache.org/docs/en/latest/>
- Babel. (2018). What is Babel? (Referenced on 17.2.2019) <https://babeljs.io/docs/en/>
- Benitte R, Greif S, Rambeau M. (2018). State of JS 2018. (Referenced on 16.2.2019) <https://2018.stateofjs.com/front-end-frameworks/overview/>
- Calderaio, J. (2017). Comparing the performance between Native iOS (Swift) and React-Native. (Referenced on 17.2.2019) <https://medium.com/the-react-native-log/comparing-the-performance-between-native-ios-swift-and-react-native-7b5490d363e2>
- CNBC. (2018). Google CEO: A.I. is more important than fire or electricity. (Referenced on 21.2.2019) <https://www.cnbc.com/2018/02/01/google-ceo-sundar-pichai-ai-is-more-important-than-fire-electricity.html>
- Dumpleton, G. (2019). mod\_wsgi. (Referenced on 5.1.2019) <https://modwsgi.readthedocs.io/en/develop/>
- Facebook. (2019). React Documentation. (Referenced on 20.1.2019, 15.2.2019) <https://reactjs.org/docs/>
- Flurry. (2017). With Captive Mobile Audiences, New App Growth Stagnates. (Referenced on 27.1.2019) <https://flurrymobile.tumblr.com/post/169545749110/state-of-mobile-2017-mobile-stagnates>
- Geary, D. (2016). Introducing Redux. (Referenced on 20.1.2019) <https://developer.ibm.com/tutorials/wa-manage-state-with-redux-p1-david-geary/>
- Google. (2010). Eric Schmidt at Mobile World Congress. (Referenced on 15.12.2018) [https://www.youtube.com/watch?v=ClkQA2Lb\\_iE](https://www.youtube.com/watch?v=ClkQA2Lb_iE)
- Google. (2019). Progressive Web Apps. (Referenced on 12.2.2019) <https://developers.google.com/web/progressive-web-apps/checklist#baseline>
- Google. (2019). Service Workers: an Introduction (Referenced on 15.2.2019) <https://developers.google.com/web/fundamentals/primers/service-workers/>
- Gustafson, A. (2008). Understanding Progressive Enhancement. (Referenced on 20.11.2018) <https://alistapart.com/article/understandingprogressiveenhancement>
- Heslop, B. (2018). A history of Content Management Systems and the Rise of the Headless CMS. (Referenced on 20.2.2019) <https://www.contentstack.com/blog/content-management-systems-history-and-headless-cms>
- Hiltunen, M. (2018). Creating multiplatform experiences with Progressive Web Apps. Metropolia University of Applied Sciences. (Referenced on 15.01.2019) <http://urn.fi/URN:NBN:fi:amk-2018120520318>
- Ighodaro, N. (2018). Why use Redux? Reasons with Clear Examples. (Referenced on 13.2.2019) <https://blog.logrocket.com/why-use-redux-reasons-with-clear-examples-d21bffd5835>



- Irish, P. (2016). Progressive Web Metrics session in BlinkOn Munich 2016. (Referenced on 17.2.2019) [https://www.youtube.com/watch?v=lxXGMesq\\_8s](https://www.youtube.com/watch?v=lxXGMesq_8s)
- Kagga, J. (2018). Understanding React Components. (Referenced on 17.2.2019) <https://medium.com/the-andela-way/understanding-react-components-37f841c1f3bb>
- Koenig, J. (2018). Headless Websites: What's the Big Deal with Decoupled Architecture? (Referenced on 15.12.2018) <https://pantheon.io/blog/headless-websites-whats-big-deal-decoupled-architecture>
- Kraft, B. (2016). A Headless CMS won't solve all your woes. (Referenced on 17.2.2019) <https://www.cmswire.com/web-cms/a-headless-cms-wont-solve-all-your-woes/>
- Lastovetska, A. (2018). Native App Development vs. Hybrid and Web App Building. (Referenced on 01.12.2018) <https://mlsdev.com/blog/167-native-app-development>
- MachMetrics. (2018). Average Page Load Times for 2018 – How does yours compare? (Referenced on 15.12.2018) <https://www.machmetrics.com/speed-blog/average-page-load-times-websites-2018/>
- Makai, M. (2012-2019). Flask. (Referenced on 15.12.2018) <https://www.fullstackpython.com/flask.html>
- MDN. (2018) AJAX. [https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting\\_Started](https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started)
- Moldovan, A. (2018). Demystifying Server-Side Rendering in React (Referenced on 21.2.2019) <https://medium.freecodecamp.org/demystifying-reacts-server-side-render-de335d408fe4>
- Quynh, HP. (2018). Progressive Web App – a new trend in e-commerce. Haaga-Helia University of Applied Sciences. (Referenced on 15.01.2019) <http://urn.fi/URN:NBN:fi:amk-2018120520217>
- Shapiro, D. (2016). Understanding Component-Based Architecture. (Referenced on 15.11.2018) <https://medium.com/@dan.shapiro1210/understanding-component-based-architecture-3ff48ec0c238>
- Statista. (2019). Percentage of all global web pages served to mobile phones from 2009 to 2018. (Referenced on 27.1.2019) <https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/>
- Tse, T. (2017). What is progressive enhancement and why should you care? (Referenced on 25.11.2018) <https://www.shopify.com/partners/blog/what-is-progressive-enhancement-and-why-should-you-care>
- Variety. (2014). Ryan Kavanaugh CES 2014 Keynote. (Referenced on 21.2.2019) <https://www.youtube.com/watch?v=3e76OKWOKJw>
- Webpack. (2019). Concepts. (Referenced on 11.2.2019) <http://webpack.js.org/concepts/>
- Zhang, F. (2018). Rolling out mobile-first indexing. Google Webmaster Central Blog. (Referenced on 01.12.2018) <https://webmasters.googleblog.com/2018/03/rolling-out-mobile-first-indexing.html>

IITUT Tuotteemme ▾ Meistä
Ostoskori Haku

## WillPower of Iitut

*Suomalaista käsityötä jo kaksikymmentä vuotta*

Iitut Oy on jo yli kaksikymmentä vuotta toiminut perheyrittys. Tavoitteenamme on tuottaa hyvää mieltä kauniilla ja kestävillä tuotteilla. Suunnittelemme että valmistamme itse tuotteemme Naantalissa.

[Tutustu tuotteisiimme](#)


### Käsitöitä täydellä sydämmellä

Kaikki tuotteemme ovat käsityönä valmistettuja kotimaisia käyttötekstiilejä.


[Tutustu meihin](#)

**19.90 €**


*Kettu lämpömyssy tee- ja rucokailuohkeihin*




### Klassikot



**Koiruli**  
RECOMMENDED UNCATEGORIZED



**Kala-aiheinen**  
DECORATIONS RECOMMENDED  
**12.99 €**



**Ketut**  
PILLOWCASES RECOMMENDED  
**19.90 €**

**Ota yhteyttä**

Puhelin: +358 2 236 0292  
Mannerheiminkatu 19  
21100 Naantali

**Tuotteemme**

Tyynyllinat  
Vilit  
Lahjat tuotteet  
Satunnaiset

© Willpower of Iitut Oy, 2018

ITUT Tuotteemme Meistä Ostoskori Haku

## WillPower of Iitut

*Suomalaisista käsityötä ja  
käsiköynnä tuotta*

Iitut Oy on jo yli kaksikymmentä vuotta toiminut perheyrittys. Tavoitteenamme on tuottaa hyvää mieltä kauniilla ja kestäväillä tuotteilla. Suunnittelemme että valmistamme itse tuotteemme Naantalissa.


[Tutustu tuotteisiimme](#)

## Käsityötä täydellä sydämmellä


Kaikki tuotteemme ovat käsityönä valmistettuja kotimaisia käyttökäsitteille.

[Tutustu meihin](#)


**19.90 €**  
*Kettu lämpömyy tee- ja  
ruchailukelhin*




### Klassikot



**Koiruli**  
RECOMMENDED UNCATEGORIZED



**Kala-aiheinen**  
DECORATIONS RECOMMENDED  
12.99 €



**Ketut**  
PILLOWCASES RECOMMENDED  
19.90 €

**Ota yhteyttä**  
Puhelli: +358 2 236 0292  
Mannerheiminkatu 19  
21100 Naantali

**Tuotteemme**  
Tyynyliinat  
Viitit  
Lahjat tuotteet  
Satunnaiset

© Willpower of Iitut Oy. 2018

17.2.2019

Lighthouse Report

<http://localhost:5000/>  
 Feb 17, 2019, 5:22 PM GMT+2  
 Emulated Nexus 5X, Simulated Slow 4G network

Performance

Accessibility

Best Practices

SEO

Progressive Web App

Score scale: ● 90-100 ● 50-89 ● 0-49

**There were issues affecting this run of Lighthouse:**

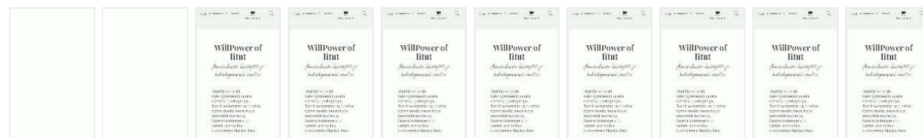
- Chrome extensions negatively affected this page's load performance. Try auditing the page in incognito mode or from a Chrome profile without extensions.

## Performance

### ⌚ Metrics

First Contentful Paint	4.1 s <span style="color: red;">▲</span>	First Meaningful Paint	7.3 s <span style="color: red;">▲</span>
Speed Index	4.3 s <span style="color: orange;">●</span>	First CPU Idle	7.3 s <span style="color: red;">▲</span>
Time to Interactive	8.1 s <span style="color: red;">▲</span>	Estimated Input Latency	190 ms <span style="color: red;">▲</span>

Values are estimated and may vary.



### 🔧 Opportunities

These optimizations can speed up your page load.

Opportunity	Estimated Savings
1 Serve images in next-gen formats	<span style="color: red;">▬</span> 4.05 s <span style="float: right;">▼</span>
2 Defer offscreen images	<span style="color: red;">▬</span> 3.45 s <span style="float: right;">▼</span>
3 Efficiently encode images	<span style="color: red;">▬</span> 2.1 s <span style="float: right;">▼</span>

blob:chrome-extension://blipmdconlkipinefhnjammfjpmbjk/c4435122-654a-424f-830f-33221bc668aa#pwa

1/4

17.2.2019

Lighthouse Report

4	Eliminate render-blocking resources	0.46 s	▼
5	Preconnect to required origins	0.15 s	▼
6	Enable text compression	0.15 s	▼

#### Diagnostics

More information about the performance of your application.

1	Ensure text remains visible during webfont load		▲ ▼
2	Serve static assets with an efficient cache policy	18 resources found	▲ ▼
3	Minimize main-thread work	4.1 s	▲ ▼
4	Reduce JavaScript execution time	2.5 s	ⓘ ▼
5	Avoid enormous network payloads	Total size was 2,830 KB	ⓘ ▼
6	Minimize Critical Requests Depth	14 chains found	▼

 **Passed audits** 10 audits ▼

## Accessibility

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.

81

### Color Contrast Is Satisfactory

These are opportunities to improve the legibility of your content.

1	Background and foreground colors do not have a sufficient contrast ratio.	▲ ▼
---	---------------------------------------------------------------------------	-----

### Elements Use Attributes Correctly

These are opportunities to improve the configuration of your HTML elements.

2	Image elements do not have [alt] attributes	▲ ▼
---	---------------------------------------------	-----

 **Additional items to manually check** 12 audits ▼

 **Passed audits** 13 audits ▼

 **Not applicable** 19 audits ▼

## Best Practices

blob:chrome-extension://blipmdconlkipnefhnjammfjpmbjk/c4435122-654a-424f-830f-33221bc668aa#pwa

2/4

17.2.2019

Lighthouse Report

1	Does not use HTTP/2 for all of its resources	16 requests not served via HTTP/2 ▲	86
2	Browser errors were logged to the console		▲
✓ <b>Passed audits</b>			13 audits

## SEO

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](#).

89

### Content Best Practices

Format your HTML in a way that enables crawlers to better understand your app's content.

1	Document does not have a meta description		▲
🔍 <b>Additional items to manually check</b>			2 audits
✓ <b>Passed audits</b>			8 audits
⊖ <b>Not applicable</b>			2 audits

## Progressive Web App

These checks validate the aspects of a Progressive Web App. [Learn more](#).



⚡ <b>Fast and reliable</b>			
1	Page load is fast enough on mobile networks		✓
2	Current page responds with a 200 when offline		✓
3	start_url responds with a 200 when offline		✓
+ <b>Installable</b>			
4	Uses HTTPS		✓
5	Registers a service worker that controls page and start_url		✓
6	Web app manifest meets the installability requirements		✓








### ★ PWA Optimized

blob:chrome-extension://blipmdconlkipinefehnmjammfjpmbjk/c4435122-654a-424f-830f-33221bc668aa#pwa

3/4

17.2.2019

Lighthouse Report

7	Does not redirect HTTP traffic to HTTPS		▼
8	Configured for a custom splash screen		▼
9	Sets an address-bar theme color		▼
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		▼
 <b>Additional items to manually check</b>		3 audits ▼	

#### Runtime settings

- **URL:** http://localhost:5000/
- **Fetch time:** Feb 17, 2019, 5:22 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 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36
- **User agent (network):** Mozilla/5.0 (Linux; Android 6.0.1; Nexus 5 Build/MRA58N) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3559.0 Mobile Safari/537.36
- **CPU/Memory Power:** 490

Generated by **Lighthouse 4.0.0** | [File an issue](#)

blob:chrome-extension://blipmdconlkipnefehnjammfjpmbjk/c4435122-654a-424f-830f-33221bc668aa#pwa

4/4