Bachelor Project

Czech Technical University in Prague

FEL

Faculty of Electrical Engineering
Department of Computer Science

Progressive Web Application Pingl

Vojtěch Rychnovský

Supervisor: Ing. Martin Ledvinka
May 2020

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Rychnovský Vojtěch** Personal ID number: **466198**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Software Engineering and Technology**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Pingl - A Progressive Web Application

Bachelor's thesis title in Czech:

Progresivní webová aplikace Pingl

Guidelines:

1. Analyze the field of progressive Web applications (PWAs) and their restrictions on various platforms. Analyze also the principles of designing adaptive user interfaces for such applications.
2. Review existing technologies for developing PWAs. Take into account their ability to overcome platform-specific PWA restrictions with respect to the functionality of the Pingl application.
3. Design the progressive Web application Pingl. In it, users would be able to list nearby restaurants with menus, search for restaurants, or view them on a map. The design should take into account the requirement of running on various devices (desktop, tablet, smartphone).
4. Implement the restaurant-selection part of Pingl based on your design.
5. Conduct user testing of your implementation, simulating typical Pingl usage scenarios.

Bibliography / sources:

[1] R. Wieruch, The Road to learn React: Your journey to master plain yet pragmatic React.js, 2018
[2] T. Ater, Building Progressive Web Apps: Bringing the Power of Native to the Browser, O'Reilly Media, 2017
[3] J. J. Garrett, The Elements of User Experience: User-Centered Design for the Web and Beyond, New Riders, 2010

Name and workplace of bachelor's thesis supervisor:

Ing. Martin Ledvinka, Knowledge-based Software Systems, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **14.02.2020** Deadline for bachelor thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Ing. Martin Ledvinka Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Acknowledgements

I am highly thankful to my supervisor Ing. Martin Ledvinka for support and guidance during my work on this bachelor thesis. Equally, I would like to thank the Czech Technical University for knowledge given to me during my studies.

Declaration

I hereby declare that I have completed this thesis independently and that I have mentioned all the used information sources in accordance with the Guideline for compliance with ethical principles in the course of writing final theses.

In Prague, 22. May 2020

Abstract

Thanks to Pingl web application, gastro business customers can order and pay without waiting, whether they want to order a takeaway for a certain time or are already sitting at a restaurant table. For takeaway orders, a clear and intuitive gateway to the application is required. It is the part where the customer chooses a restaurant according to the entered criteria or the location. Also, the user should see individual meals nearby.

In this work, the prototype of this gateway was implemented as a progressive web application (PWA). The resulted app was user-tested. This implementation was preceded by a detailed analysis of PWAs, their capabilities and limitations that exist for them in operating systems of various mobile and desktop platforms. The work also includes an analysis of technologies for PWA development.

Keywords: Progressive Web Applications, React, Pingl

Supervisor: Ing. Martin Ledvinka

Abstrakt

Díky webové aplikaci Pingl si mohou zákazníci gastro podniků objednat a zaplatit bez čekání, ať už si chtějí objednat takeaway na určitý čas, nebo již sedí v restauraci u stolu. Pro takeaway objednávky je nutná přehledná a intuitivní vstupní brána do aplikace, část, kde si zákazník vybere restaurace podle zadaných kritérií nebo podle lokace. Zároveň by měl uživatel mít možnost vidět i jednotlivá jídla v okolí.

V rámci této práce byl naimplementován prototyp této vstupní brány ve formě progresivní webové aplikace (PWA). Výsledná aplikace byla uživatelsky otestována. Této implementaci předcházela detailní analýza PWA, jejich možností a omezení, která pro ně existují v operačních systémech různých mobilních i desktopových zařízení. Součástí práce je také analýza technologií pro vývoj PWA.

Klíčová slova: progresivní webové aplikace, React, Pingl

Překlad názvu: Progresivní webová aplikace Pingl

Contents

Figures

Tables

Chapter 1

Introduction

Progressive Web Application (PWA) is a buzzword widely used in today's world of web developers. PWAs have appeared continuously as a set of trends and intentions behind building modern websites without having any particular release date. However, suddenly, if we take a look at all changes which have happened in the last years, we can distinguish a clear concept - websites are able to operate almost the same way as native apps do, but they still keep all the advantages of traditional websites.

Alex Russell, Google Chrome engineer, wrote in 2015 an article[1] where he and his coworker and wife Frances Berriman gave them the name *"Progressive Web Apps"*. Nowadays, we can see many large companies using PWA technologies to help them with building online platforms for their customers, which have a much higher conversion rate than conventional websites.

PWAs sum up a new perspective on how developers can build web apps and how they have delighted user experience with many new features that were unimaginable at the early age of web development.

This work aims to:

1. highlight and explain the main capabilities of PWAs
2. research technologies usable for PWAs development
3. design a responsive user interface of Pingl app
4. implement a prototype of Pingl app as a PWA
5. conduct user testing of the implementation

Chapter 2

Progressive Web Apps

2.1 What is PWA?

PWA abbreviation stands for Progressive Web Application. PWA is a hybrid between a traditional web page and a native application[1], meaning it is a web page that implements some of the functionality that was traditionally widely seen only in native apps. There is no particular set of features that the web site has to implement to be considered as a PWA, but the most common ones are the ability to be installed to the home screen, the ability to work offline and to persist data between user sessions. Then based on the use case of the app, the app usually enriches the user experience by push notifications or access to device hardware, such as the camera.

2.1.1 PWA Principles

Any Progressive Web App should be reliable, fast and engaging. These general principles can be described by more specific characteristics: [1]

1. Reliable:
   - **Progressive** - the PWA should work in any browser, even when it does not support PWA features (this approach is called progressive enhancement)
   - **Responsive** - the PWA should have a layout that adapts to any screen size (desktop, mobile, tablet, etc.)
   - **Safe** - the PWA has to be served over a secure connection (TLS)
   - **Connectivity Independent** - by implementing a service worker as a client-side proxy, the PWA can control caching and can provide resources even when the Internet connection is not available
2. Fast:
   - **Fast load times** - the PWA can implement a service worker to cache files and serve them without waiting for a remote request

[1] The term *native application* refers to an app written specifically targeting one platform or device

- **Fresh** - the PWA is always up to date, there is no need to open the app store to update the app as it is in the case of native apps

3. Engaging:

   - **App-like** - the PWA offers an immersive fullscreen experience without browser tabs and bars. The web feels like a native app.
   - **Re-engageable** - the PWA can reach users with push notifications even after they have closed the browser, which directly leads to increased user engagement and higher customer conversion (i.e. converting a site visitor into paying customer).
   - **Installable** - users can add the PWA to the home screen directly from the browsers without opening the app store. The PWA can control this process with `Web App Manifest` and modify the app name, icon or splash screen or screen orientation
   - **Linkable** - the PWA should be deep linkable and should provide a unique URL for every route in the app, which can be shared as a link with other users

In general, the main intention behind PWA is to bring *app-like* experience to the web browser. It is necessary to mention that PWAs are not restricted to be used only in mobile devices. As the PWA is still a website and its core is a JavaScript file, it can also be used in any desktop environment with a web browser. The main reasons why to implement PWA on desktops are push notifications and offline capabilities.

In this work, I will look closely at their features and implementation for mobile devices, but a majority of these principles can also be applied to the desktop environment.

2.1.2 Difference between PWAs and Hybrid Applications

There is an emphasis on the word "progressive" in the PWA abbreviation because of an intention to distinguish PWAs and hybrid applications.

Hybrid application has a "shell" written as a native app encapsulating a web view with a web page, which is written as a conventional website. These apps use native code to implement platform-specific functions or functions that are not accessible from the web browser. These functions are usually more tightened to the operating system of the device, for example, push notifications or access to the device file system. Also, the native part of the app can, after user permits them, read data of other apps (contacts, call history, etc.). Native code can additionally take advantage of the accelerometer and other communication peripherals such as Bluetooth or NFC. Because of the native shell with an embedded browser, users have to download them from the application store of their platform (App Store for Apple and Google Play for Android).

On the other hand, PWAs progressively take advantage of new web technologies - if they are not present, the app operates as a conventional website

(or at least it should). Moreover, PWAs are entirely written in JavaScript, which slightly limits their capabilities, but gives them a huge benefit - users can use them without downloading any app from the app stores.

Hybrid Apps:

Pros:

- the content of the web-view can be updated easily only by updating the website on the server
- development of a hybrid app is faster in comparison to native apps, which also results in a lower cost of the app
- the web part of the app is shared between multiple platforms

Cons:

- it is necessary to write native code, which means the development team has to have more comprehensive knowledge or larger amount of members
- if the app lacks enhanced functionality over the encapsulated website, Apple App Store can reject this app during approval process, because of one of the store requirements: "Your app should include features, content, and UI that elevate it beyond a repackaged website." [2]

Progressive Web Apps:

Pros:

- they have all the advantages of hybrid apps - fast development, shared code between platform, instant distribution of updates
- they target all platform without a necessity to write any native code

Cons:

- capabilities of these apps are limited
- the support of some features can vary between web browsers

2.1.3 Transition from Websites to Modern Web Apps

In the past, traditional web pages were used only to display content served from web servers, and the client-side was intended to be plain, leaving all logic functionality on the server. Since users started using more powerful machines to open those webpages, it became apparent that the website owners wanted to bring more of the computations and logic to the client-side to lower the requirements on the server.

The other factor that strongly helped to bring more logic to the client-side was the rise of JavaScript in recent years. JavaScript frameworks like

React or Angular help developers to quickly build web applications with rich functionality. In these frameworks, we even skip writing the plain HTML code, and the HTML elements are rendered using JavaScript. By this approach, we can easily bind functions to these elements and react to various events, including user click and touch or any other possible events. Also, because the UI is completely rendered on the client-side, there is no need to fetch another HTML file when the user navigates through the website. All HTTP requests are made asynchronously without refreshing the browser tab using `XMLHTTPRequest` or `Fetch API`. This approach is also called SPA (Single Page Application).

The other area that has changed a lot is the user interface. As more and more users browse websites using mobile devices (in fact, according to statistics, nowadays, the number of mobile visits has exceeded the number of desktop visits [3]), the design of websites had to change. Websites and native applications got more similar to each other, and many websites started to be designed in a mobile-first approach. Responsive design[2] became a standard in modern web development. This is accomplished by using `@media-queries` in CSS (Cascading Style Sheets).

So after this shift, there are users having devices with enough computational power, who access our websites, that are precisely designed for various mobile sizes, the rich functionality of the app is provided by JavaScript code. It is clear that if we push this modern website a little bit further, we can give users app-like experience on the web.

2.2 Service Worker

The core JavaScript file of a PWA is called a Service Worker.

> *A service worker is a script that can be registered to control one or more pages of your site. Once installed, a service worker sits outside of any single browser window or tab. Events such as requests for files from the web can be intercepted, modified, passed on, and returned to the page.* [4]

Key aspects of service workers:

- Service Worker is a JavaScript file
- Service Worker is a layer between the web page and remote server, which can listen to events and modify responses
- This script is not dependant on browser tab - it works even when the tab is closed
- It cannot access the DOM directly

[2] Term "responsive design" means rendering webpages based on size and orientation of the screen

- It runs in its own context (separated from window global context)

Service Workers use-cases [5] include acting as a client-side network proxy enabling caching using Cache Storage API, or as a receiver for push notifications.

2.2.1 Service Worker Lifecycle

Before a Service Worker is active, it goes through a few states [6]. Service Worker is event-based and can listen to various events triggered during its lifecycle. These states and corresponding events are shown by Figure 2.1.

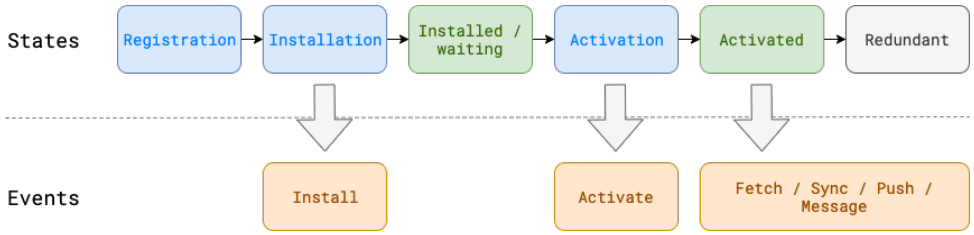


Figure 2.1: Service Worker Lifecycle

Registration

First, it is necessary to check if the browser supports Service Workers, then the Service Worker can be installed. Supported browsers will use an enhanced site with Service Workers, while others will display the website without PWA functionality.

```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.register("/serviceworker.js")
}
```

Scope of Service Worker
Service Worker can listen and modify only events coming from its scope. The scope is defined by origin and a folder, where the Service Worker is placed. When registering, the scope can be set to a smaller subset, but can not be extended.

Installation

Installation happens only if the browser considers the Service Worker as new. This will happen only under these conditions: there is either no installed worker, or there is a byte difference between the new and the current worker.

Activation

After installation, the service worker is activated, only if the current Service Worker is no longer controlling any page. If there are any pages controlled

by an old service worker, the new service worker will stay in `Waiting` phase, until all tabs are closed. By this approach, we can be sure that only one version of the Service Worker is installed at a time. However, this logic can be overridden and the Service Worker can force to control all pages by calling a function `clients.claim()`.

2.2.2 Listening for Events

When the worker is `Activated`, it controls all pages within its scope and it listens to events triggered by the controlled page. Listener can be registered in a very similar way as in window scope:

```
self.addEventListener("eventName", function(event) {
   // Perform some tasks here
})
```

List of Possible Events:

- **install** - Service Worker can typically precache all necessary files during installation, so it can rely on their availability even when the device goes offline. Install phase can be extended by calling `event.waitUntill()` untill all resources are fetched.
- **activate** - During this event, the Service Worker typically deletes old unnecessary caches created by previous instances of the Service Worker.
- **message** - Because the Service Worker can not access the controlled page directly, `Channel Messaging API` has to be used. After registration, the page receives a reference to the Service Worker and can use `postMessage()` function to send messages to it. Listener to `message` event is the place, where the Service Worker will receive this message.
- **fetch**: Service Worker can listen to every request sent from the page. It can pass the request to the remote untouched, or it can retrieve the resource from a `Cache Storage` instead of from the remote server using one of the cache strategies discussed in *Caching Strategies (subsection 2.3.1)*.
- **push** - PWAs can use `Push API` to receive notifications from remote server. When the event is triggered, the Service Worker can react to this event, typically by showing a notification using `Notification API`. Example of this functionality is shown in *Re-engageable PWA - Web Push Notifications (subsection 2.5.3)*.
- **sync** - PWAs can use `Background Sync API` to defer actions until the user has stable connectivity. At that moment, the listener to the `sync` event is triggered, and the Service Worker performs the deferred task. Example of this functionality is shown in *Reliable PWA - Background Sync (subsection 2.5.4)*.

2.3 Cache Storage API

Caching is the most common use-case of Service Workers. By using a client-side cache, we can deliver resources without the need to send HTTP requests through the network. The response from the cache is faster and can be returned even when the client is offline, which is a key part of PWA functionality.

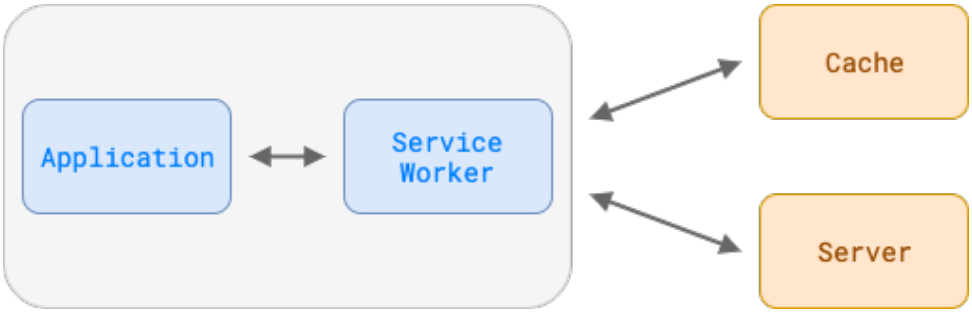


Figure 2.2: Service Worker with a Cache

> *The CacheStorage interface represents the storage for Cache objects. It provides a master directory of all the named caches that can be accessed by a ServiceWorker or other type of worker or window scope. It maintains a mapping of string names to corresponding Cache objects.* [7]

Methods of CacheStorage Interface [7]:

- **CacheStorage.open(cacheName: string)**[3] - returns a Promise resolving to a reference to a newly created `Cache object` with the passed name or to an existing one
- **CacheStorage.delete(cacheName: string)** - deletes a cache by its name. Returns a Promise which resolves to true, if the cache has been deleted, or to false otherwise
- **CacheStorage.keys()** - returns a Promise resolving to an array of cache names
- **CacheStorage.has(cacheName: string)** - returns a Promise resolving to true, if any cache with passed name exists, or to false otherwise
- **CacheStorage.match(request: Request)** - returns a Promise resolving to a corresponding cache entry in any of the Caches in CacheStorage, otherwise the promise is rejected

[3] From this point, for better clarity of functions parameters, I will use TypeScript-like notation in form of (parameterName: Type)

Methods of `Cache` stored objects are similar to `CacheStorage`. We can `put()` a request to the Cache, or we can use `add()` method which is equivalent to calling `fetch()` and then `put()`. A cache entry can be deleted from Cache by calling `delete()`. Methods `match()` and `matchAll()` retrieve items from the cache. These methods return a Promise.

The `CacheStorage` is heavily used by Service Workers. The typical use-case is:

1. Open new or existing cache during the install phase and add to it all resources, which should be prefetched. The ServiceWorker is not active until all these resources are saved in the cache.
2. Listen to `fetch` events and according to a selected caching strategy, return the response from a remote server or return prefetched file.

2.3.1 Caching Strategies

Based on the requirements for the cache, we can choose one of cache strategies. If we get the request from the cache, it may not be the newest version, but the response time is almost immediate. On the other hand, if we always get the up-to-date version from the remote server, our app has to wait until the request is made. In one Service Worker, it is common to combine multiple of these strategies for different types of resources.

List of Caching Stategies [7]:

- **Cache only** - Resources are usually cached during the install phase. Then they are returned without sending the request to the remote server. This method is used for static, non-changing assets.
- **Network only** - Resources are fetched from the remote without querying the cache. This method is used for logs and page views analytics.
- **Cache, falling back to network** - Resource is queried from the Cache, and if it is not found, it is fetched from the network.
- **Network, falling back to cache** - Resource is fetched from the network, and if the request fails, then the cache is used as a fallback.
- **Cache, then network** - Resource is queried from the Cache and returned immediately. At the same time, a network request is sent to get an up-to-date resource and the response is put to the cache instead of the old one. This approach can require some modification in the app to send two requests - one to get resource from cache and the second to get the new version.
- **Generic fallback** - If the resource is not available in the cache and the remote server can not be reached, then a generic fallback can be used. In a case of images, a gray image can be used as a fallback.

2.3.2 Storage Quotas

Browser storage space is limited per `Origin`[4] and is shared between `Cache Storage API` and other storage options such as `IndexedDB` or `Local Storage`. The quota is not defined in the specification and varies between browsers and is dependant on storage condition.

In general, it ranges between 50MB and 20GB [9]. Some browsers will notify the user when certain limits are overreached (for example desktop Firefox at 50MB). The most strict is mobile Safari limiting the storage to 52MB, but in this case, the `IndexedDB` is excluded from this amount.

2.4 Web App Manifest

Web App Manifest is a file in JSON[5] format describing how a PWA should behave when it is installed on the home screen. Though the prefered extension is `.webmanifest`, the `.json` extension is also allowed. In both cases, it is recommended to label the file with `"application/manifest+json"` MIME Type.

This file should be linked from the Head of HTML document by `Link` tag:

```
<link rel="manifest" href="/manifest.webmanifest">
```

2.4.1 Manifest Properties

List of manifest allowed properties [11]:

Language Style:

- **dir** - ["ltr" | "rtl" | "auto"][6] - this property specifies the base text direction of other manifest properties and will affect how they will be displayed. It should be set consistent with the `lang` property. Text in LTR languages is displayed from "left to right", RTL languages will be displayed from "right to left".
- **lang** - [string] - this property specifies the language of other manifest properties in format "en-US"

App Names:

- **name** - [string] - represents the application name. This property will be used in "Add to Home Screen" dialog (even though this property is mandatory, if it is not provided, the name of the app will fallback to short_name, and the "Add to Home Screen" dialog will appear)

[4] Web content's origin is defined by the scheme (protocol), host (domain), and port of the URL used to access it. [8]

[5] JSON (JavaScript Object Notation) is a lightweight data-interchange format containing a collection of name/value pairs and an ordered list of values. [10]

[6] Values in square brackets specifies the form of allowed values. It can be either a set of allowed values, or a type of the value.

- **short_name** - [string] - this name will be used as the app name next to the app icon on the home screen

App URLs:

- **start_url** - [string] - the PWA will launch on this route when the user opens it from the home screen. This means that our app has only one entry point regardless of the subpage, where the user was located when he installed the PWA.
- **scope** - [string] - sets the scope of current manifest. If a user navigates outside of this scope, the app will change to the normal browser window. The scope is relative to the manifest file (`/folder`), or can be set as absolute URL (`https://example.com/folder/`).

App Style:

- **icons** - [array of objects { src: string, type: string, sizes: string }] - this icon will be used as the app icon on the home screen (it can be different from the favicon)
- **orientation** - ["any" | "natural" | "landscape" | "landscape-primary" | "landscape-secondary" | "portrait" | "portrait-primary" | "portrait-secondary"] - this property specifies the orientation of the display
- **display** - ["fullscreen" | "standalone" | "minimal-ui" | "browser"] - this property specifies the preferred amount of browser's UI controls around the app.
- **background_color** - [string] - the browser will set this color as a background until the main stylesheet is loaded. On Android, this color is used for the Splash Screen
- **theme_color** - [string] - this property changes the color of UI elements of the browser (especially Android status bar)

Other allowed properties:

- **categories**, **description**, **iarc_rating_id**, **screenshots** - categorization properties, they are intended to by used by PWA stores in the future
- **related_applications**, **prefer_related_applications** - specifies the equivalent native apps with the same functionality, the browser may suggest installation of them to the user
- **shortcuts** - describes key tasks in the PWA, where the operating system can guide the user, for example, "start playing the first song from my favorite playlist". OS can suggest these actions to the user, for example, after a long press of the app icon.
- **serviceworker** - defines Service Worker location

2.5 Selected PWA Features

2.5.1 Installable PWA - Add to Home Screen

Probably the most significant feature PWAs have, is the possibility to have an icon on the home screen or desktop. Then on most of the platforms, the app is indistinguishable from a native app. Users are more likely to open the app again if they see the icon on their screens everyday than if they have to type the URL to the browser.

If the browser should allow users to open the "Add to Home Screen" dialog, it has to recognize these features on the website:

- **Secure origin** - the website has to be delivered over TLS (HTTPS) to ensure, that the content of request was not modified between the browser and the server
- **Service Worker** - it has to be registered within the scope of `start_url` specified in Web App Manifest, and it has to return some response even when the device is offline (the full offline functionality of the app is not needed).
- **Web App Manifest** with a minimum configuration of these properties: `name` or `short_name`, `start_url`, `display` (set to other value than `browser`), `icons` (at least one square .PNG icon of size at minimum 192x192 px)

If all these conditions are met, and the browser supports the service workers, then the user can tell the browser to install the PWA.

Prompting the user to install the PWA

The process of manual installation of the PWA to home screen differs between different platforms and usually is not really user-friendly. Also, the users may not be aware that the website is a PWA and can be installed. Browsers provide a solution for this - if the app can be installed, an event `beforeinstallpromp` on window element will be fired. The app can attach a listener to this event, which will receive a `deferredPrompt` as a parameter and can decide whether to trigger a `prompt()` function on this `deferredPrompt`. The `prompt()` function can be fired only once on the `deferredPrompt` and can be called only from a click handler, so if the user dismisses the prompt, the next prompt can be shown only after page refresh.

Also, the prompt should not disturb the user, and it should be used only if the user can benefit from it. For example, the app can suggest the user to install the app after his first purchase, or there can be a banner with a sign for example *"Install the app and read our articles while offline"*. By this approach, the user will see the benefits of installing and will be more likely to click on the install button.

A simple PWA implementation is demonstrated in an example project: `https://github.com/rychnovsky/pwa_examples/tree/master/01_installable_pwa`.

2.5.2 App-like PWA - Fullscreen Experience

If a user opens a PWA from an icon on his home screen, he expects the app to be fullscreen and not to be in a browser window. This particularly concerns mobile devices, PWAs should look like native apps.

On Android

On Android (Google Chrome) the support of Web App Manifest is high, and we can use its properties `theme_color` to set status bar color and `background_color` to set splash screen background.

On Safari

Apple Safari uses its own proprietary meta tags to ensure that developers will customize all asset to match Apple ecosystem [12].

- webmanifest display property is partially ignored. Both `Fullscreen` and `standalone` values results in fullscreen view, and both `minimal-ui` and `browser` results in the classic Safari browser UI.
- webmanifest icons are ignored completely, an extra meta tag has to be used to set home screen icon instead
  `<link rel="apple-touch-icon" href="apple-icon.png">`
- splash screen (also called launch screen in Apple termininology) can be set by meta tag. If it is not provided, a screenshot of the app will be used.
  `<link rel="apple-touch-startup-image" href="/launch.png">`
- status bar can be set to `default | black | black-translucent`. `Default` stands for white bar with black text, `black` is the default value and stands for black bar with black text (so the text is not visible) and `black-translucent` sets the bar to the same color as body background with white text.
  `<meta name="apple-mobile-web-app-status-bar-style" content="black">`

Further Customization

On both platforms the app can be further restricted by disabling zooming (by meta tag `viewport` set to `user-scalable=no`) or by disabling user gestures (by applying CSS rule `-webkit-user-select: none;`).

Manifest property `orientation` is ignored in Safari and the only workaround is to apply a rotation transformation to whole body by CSS rule.

On both platforms, we can check by JavaScript if the web is installed as PWA and viewed in a native-like window. In Safari, the check can be done by accessing property `window.navigator.standalone` (boolean), in Chrome the check has to be done by checking display mode:
`window.matchMedia('(display-mode: standalone)').matches`
To apply CSS rules only for when the app is in standalone mode, the `MediaQueries` can be used:

```
@media all and (display-mode: standalone) { /* rules here */ }
```

A simple PWA implementation with app-like styling is demonstrated in an example project:
`https://github.com/rychnovsky/pwa_examples/tree/master/02_app_like_experience`.

2.5.3 Re-engageable PWA - Web Push Notifications

For some businesses, it is crucial to keep users engaged even after they have left the website and target them with banners and rich advertising. In other use-cases, the app wants to notify users or deliver a small amount of information to them in real-time. For these short real-time updates, push notifications should be considered as a solution.

These web push notifications appear in the standard native notification center of the device, having the same form as any other notification from any other native app. Notifications can be triggered locally from the website or can be pushed from a remote server and they will appear in real-time even when the website is closed. To manage notifications, two APIs are used: `Notification API` and `Push API`.[13]

Notification API [14]

This API lets us display notifications on the device. To prevent any visited website from showing dozens of push notifications and flooding user's device with unwanted advertisements, the website has to ask for permission first. Some browsers such as Mozilla Firefox even block all notifications from websites that show notifications before they ask for permission.

```
Notification.requestPermission(result => {
   if (result === 'granted') {
      // notifications can be shown
   }
})
```

After granting the permissions, a notification can be shown using `Notification` constructor.

```
new Notification(title: string, options: NotificationOptions)
```

This notification can be modified using `options` parameter. Support of these options varies between browsers. The commonly used ones are:

- **body** - [string] - text of the message in the notification
- **icon** - [string] - URL of an icon which will appear in the notification
- **image** - [string] - URL of an image which will appear in the notification
- **actions**: [Array<action: string, title: string, icon: string>] - specifies action buttons that are visible next to the notifications. These actions (when clicked) are passed to the service worker, which can react to them.

All major browsers support this API, except iOS mobile Safari and Internet Explorer[15].

Notification API also introduces new functionality for the Service Workers, which can be used while the website is closed or in background:

- **showNotification(title: string, options: NotificationOptions)** - shows a notification. Accepts the same arguments as Notification constructor does. This function is used to show notifications in `push` event handler in SW.
- **getNotifications(options: {tag: string})** - returns a `Promise` resolving to an array of all notifications shown by the current origin
- **onnotificationclick** - this event handler will receive an event when the notification has been clicked and can react to it, for example, open an address in the browser. If the user clicked on a certain action in the notification, this action would be available as `action` property of the event

None of this background functionality is available in desktop Safari, which means that this API cannot be used to show Notifications from the background there. However, apart from this standardized API, there is Apple proprietary solution called `Safari Push Notifications`[16]. It uses `Apple Push Notification service (APNs)` and solely works in the desktop Safari browser and can be used there to show push notifications in the background.

Push API [17]

The Service Worker, once registered and active, can receive push messages even when the website is not in the foreground. First, it needs to subscribe to these messages, and then it can attach a listener for these messages:

- **PushManager.subscribe(options)** - accepts an options object containing `applicationServerKey`[7]. Returns a `Promise` that resolves to a `PushSubscription` object containing `endpoint`, which is an URL, where the app can send messages to this subscription. This URL should be

[7] This key is a public server key used for an identification of the message sender. When specified, a signed JWT token with the private key has to be send with every push message. This is called VAPID (Voluntary Application Server Identification).

passed to the application server, which will send these messages. It should be kept secret, otherwise, other apps would be able to use this subscription to send notifications to the device.

- **onPush** event - this is the event handler, where Service Worker receives the content of push messages and reacts to them. Typically, it shows a notification, but the content of the messages can also be saved to `IndexedDB`.

Permissions for using `Push API` are shared with `Notifications API` - once the user allows `Notifications`, the `Push API` can be used as well. Push API is not supported in both desktop and mobile Safari browsers and Internet Explorer.

These two APIs together form the required steps to receive push notifications on a website:

1. Ask for a permission to use notifications
2. Register a service worker
3. Subscribe to messages
4. Save the subscription on the application server
5. Send a message from the application server to a push service server
6. Receive the message in the Service Worker in `onpush` event
7. Show the notification

The process of sending push messages using push messaging service is illustrated in a sequence diagram *Figure 2.3*.

Push Messaging Services

To send notifications to multiple platforms (web, native iOS, native Android, etc.), push messaging services are used. They offer a unified and simple API and handles the delivery of messages to all these platforms. These services vary in pricing, but for many use-cases, they are free. The choice usually depends on other services provided by these services, for example hosting, databases, or machine learning functions. Examples of these services are:

- Firebase Cloud Messaging - `https://firebase.google.com/docs/cloud-messaging/`
- Amazon Simple Notification Service - `https://aws.amazon.com/sns/`
- IBM Push Notifications - `https://www.ibm.com/cloud/push-notifications`

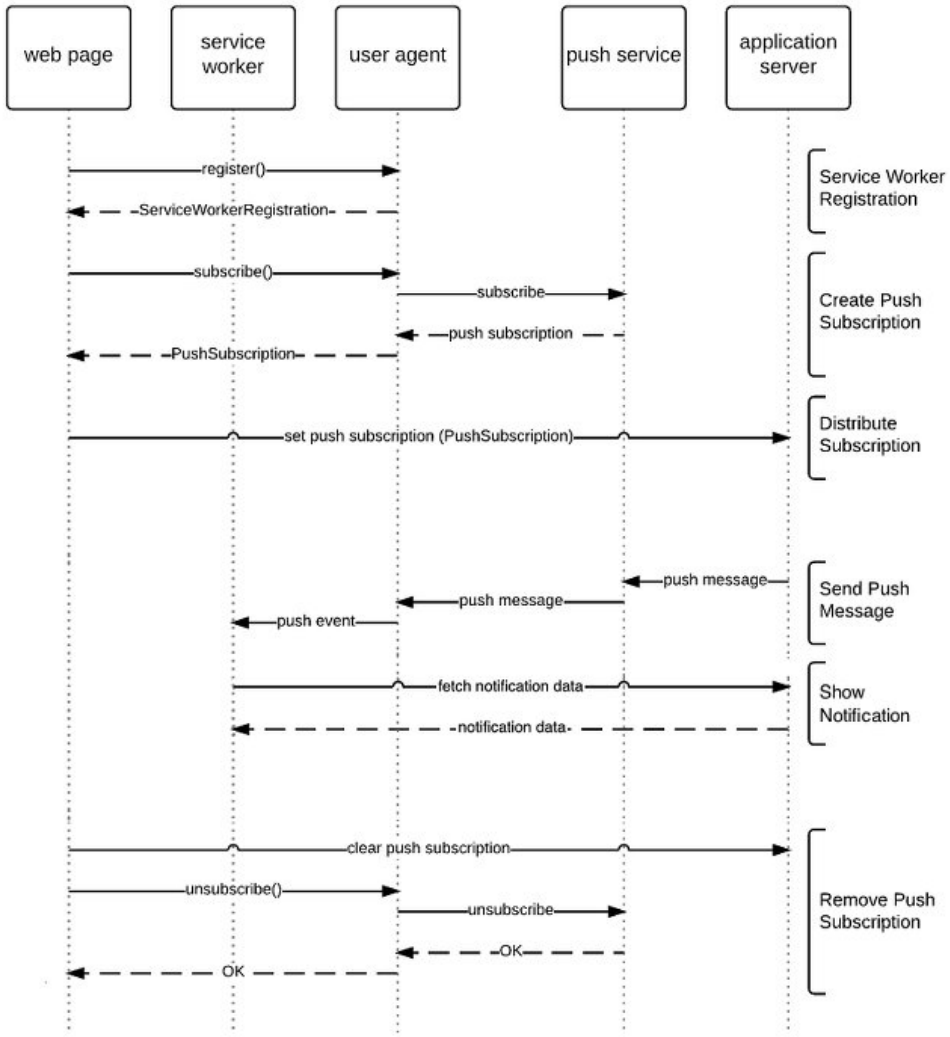


Figure 2.3: Web Push Sequence Diagram [18]

Push Notifications Alternatives

The above-discussed web push notifications allow real-time delivery of updates to the users in a form that is familiar and intuitive to all of them, but their most significant drawback is a lack of support in iOS mobile browsers. To deliver information also to this group of users, we can use other alternative methods and channels, each offering different balance between price, audience width, and speed of delivery:

- SMS - available to all mobile platforms, but higher price
- EMAIL - the user has to have an email client installed, free
- Facebook Messenger - not installed on every device, free

2.5.4 Reliable PWA - Background Sync

Background sync is a new web API that lets the app defer actions until the user has stable connectivity. This is useful for ensuring that whatever the user wants to send, is actually sent. [19]

The process of using the background sync is following:

1. Register a new `sync` event with any name. This name represents the desired action. If any data are supposed to be passed to the Service Worker and to be used there, they have to saved in `IndexedDB`.

   The sync event can be simply registered using:

   ```
   navigator.serviceWorker.ready.then((swRegistration) => {
      return swRegistration.sync.register('nameOfSyncEvent')
   })
   ```

2. In the Service Worker, add an `eventListener` for the `sync` event. This event is triggered when the Internet connectivity is restored.
3. Do some tasks based on the event name, for example send some data to the app server. Additional data can be retrieved from `IndexedDB`.

Background Sync Use-cases

The background sync API can be used to enhance offline functionality, for example to send messages in chatting app, after the app was closed. Other use case can be to ensure requests reliability. In a case the server is unreachable (it is down or not responding), the request can be repeated after some time.

It is necessary to mention, that this API is supported only in Google Chrome, Opera and Edge browsers. [20]

2.6 Security Considerations

Because a PWA is a website, it is targeted by the same security threats. Examples can be Cross-Site Request Forgery (CSRF) or Cross-Site Scripting (XSS) [21]. Many other threats are in concern of the backend part of the app.

In terms of safety of the user, the PWA runs inside of a browser, so it is encapsulated from other apps. The PWA has to ask the user for specific permissions before it can access the device hardware such as camera or microphone.

The PWA can register the Service Worker only if it is served over a secure connection (HTTPS - Hypertext Transfer Protocol Secure). The Service Worker can modify any request and response coming out of the page, so the browser has to be assured that the Service Worker was not modified by a third-party and is not malicious.

2.7 Performance Optimisation

A fast load is a crucial part of the user experience when browsing the web. When a browser displays a web page, it has to go through a `Critical Rendering Path (CRP)` [22]. It consists of creating the `Document Object Model (DOM)` from HTML file, then fetching JavaScript files, that can modify the `DOM`. Then CSS resources are requested, and a `CSS Object Model` is created. These two are combined into a `Render Tree`, which represents a tree structure of all visible elements on the page. Then a `Layout` is created, and it is `Painted` on the screen.

This process implies one critical point - the browser has to postpone the rendering until all CSS and JavaScript files are downloaded. We can optimize this process to reduce the load time:

1. Reduce the amount of the resources and their size by bundling (or splitting) and minification
2. Tell the browser not to wait for a particular resource using `defer` and `async` attributes
3. In the case of a PWA, cache these resources using Service Workers.

Using the Cache to Optimize PWA

From the perspective of PWA, using the caching is the most relevant. By using this local cache, possible issues with Internet connectivity, server downtimes or overload, can be overcome. We can be sure that the response from the cache will be immediate and will not delay the rendering. Based on the type of the resource, we have to find the most suitable caching strategy, this topic was discussed in the *Caching Strategies (subsection 2.3.1)*. Also, the cache will affect only renders after the Service Worker is active, so for optimization of the first render, we have to use other techniques.

In an extreme case, when the user opens the page for the first time, we can prefetch all possible files, images, and resources. Then every upcoming rendering will be very fast, but we can easily exceed storage limits (described in the *Storage Quotas (subsection 2.3.2)*), and also users have limited cellular data plans.

Other Optimization Techniques

Since the PWA works as a standard website, the same performance optimization principles apply, and reducing the length of `CRP` is only one of them. There are many other optimization techniques, such as image optimization or optimizing execution of JavaScript, but these topics are out of the scope of this work.

2.8 PWA Compatibility

To have a general idea of how many users we can target with PWA functionality, we can look at stats of Service Worker support, which is, as described above, the core of the PWA.

Service Workers are supported by 95% of mobile devices and 90% desktop devices [23]. This proportion gives us a huge audience we can target with our enhanced functionality. The only downside is that we can not rely on all browsers to support the same features and functions as other browsers do. One example of how the functionality of a particular API differs across browsers and their versions can be `Push API` enabling us to send push notifications to a user device. Google Chrome fully supports this API since version 50 released in April 2016, Safari for desktop has a custom implementation of this API since 2016, while Safari for iOS does not support this API at all. Some of other issues are reported by PWA Police on their Git repository[8].

PWAs consist of many features whose support differs even across versions of a single browser, so it is necessary to test each of the functionality before using it. This approach is named progressive enhancement. The users who use unsupported browsers will experience a standard website. It should work, but they can notice that they lack some functionality.

2.9 PWA Disadvantages

PWAs bring to the web many exiting features, but it comes with some downsides:

- They are not distributed through app stores, and users have to open the website via browser first
- Limited access to device features such as NFC, accelerometer, contacts
- Limited offline and background functionality
- Support varies between devices and browsers, for example:
  - It is not possible to prompt the user to install the PWA on every platform
  - Push Notifications do not work on iOS

2.10 Real-World Examples

PWAs are used widely across almost every industry. Here are a few examples of successful PWA implementations:

[8] PWA Police repository: `https://github.com/PWA-POLICE/pwa-bugs`, (accessed: Nov 24, 2019)

https://app.starbucks.com/ - Starbucks has built a PWA as a supplementary tool to their native app. This PWA features advanced caching mechanisms, and customers can browse the menu offline. Users also benefit from smaller app size, which is 99.84% smaller than the native iOS app (233Kb compared to 148MB). From a technical perspective, the app is built using React JS Framework, it uses GraphQL API and Redux, and the service worker is built on top of the Workbox Framework. [24]

https://m.uber.com/ - Uber wanted to allow customers to order rides even on a slow connection and on devices unsupported by their native app. Their PWA is built on Preact (a smaller and faster alternative to React). The core app is bundled in a just 50KB file and takes less than 3 seconds to load on 2G networks. [25]

https://www.pinterest.com/ - Pinterest has built their PWA using React JS + Redux stack with a Service Worker built on top of Workbox Framework. They have focused on the performance (especially fast load time) and ended up increasing the number of users who spend more than 5 minutes on the website by 40% in comparison to the previous old website experience. This improvement has also gained them a 44% increase in user-generated ad revenue. [26]

Chapter 3

Technologies for PWA Development

PWA is an application served through web browsers, therefore, it uses traditional web technologies like HTML, CSS, and JavaScript. The core of PWAs is a Service Worker, which is also a JavaScript file. It means that it is not necessary to use any additional programming language, but there are some options, which can be used instead of writing plain JavaScript and HTML. After compilation and bundling, all these options result in HTML, JavaScript, and optionally CSS files.

These options can be divided into two parts:

- using TypeScript to improve the development process
- using a SPA Framework to not to write the code from scratch

3.1 Typescript

The implemented prototype in upcoming chapters will use Typescript.

> *TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.* [27]

TypeScript makes using JavaScipt better, especially for large-scale applications. The code written in Typescript is generally more readable, the debugging is easier and the development time for large-scale applications is generally shorter.

As it is a superset of JavaScript, any code written in JavaSript is also a valid Typescript code. The filename extension is `.ts`.

Typing

It adds static typing to JavaScript, which ensures type-integrity through the app. Here is an example of a function taking two arguments: the first is a number, the second is a string. The function returns a string.

```
function foo(a: number, b: string): string {
   // some code here
}
```

If we omit the typing, the compiler will infer the type. For example, if our function returns "abc", typescript knows, it is a string. Types can also be declared by declaration files with extension `.d.ts`.

3.2 SPA JS Frameworks

The PWA usually contains a complex client-side logic in JavaScript, so in many cases, a JavaScript framework is used. Generally, in these frameworks, the entire client-side application is written in JavaScript. I will look closely at the two most popular SPA (Single Page Application) JavaScript frameworks - React and Angular.

3.2.1 Angular

Angular is a SPA Framework developed and managed by Google, written in TypeScript. All necessary functions, such as data binding and form validation, are included in the Angular itself. This is a considerable contrast to React, which covers only the UI and other libraries for other parts of the app have to be used.

Angular is a complex framework using advanced programming techniques:

- **Components and two-way data binding:** The key elements in Angular are Components, which use two-way data binding - properties are bound to the template, and events are bound the other way.
- **Dependency Injection (DI):** The shared app functionality should be delegated to service classes. Ideally, components should be concerned only about the view. Dependency Injection is used to get instances of these services inside components.

Angular architecture combines MVVM (Model, View, ViewModel) and component-based approach, meaning that View and Controllers are described within one component: [28]

- **Model:** it describes entities used within the app. They are specified in Service files together with functions associated with these entities.
- **View:** it consists of `.html` and `.css` file, so-called templates. By using directives (e.g. `{{}}`, `ng`), we can describe two-way data binding between a view and a controller.
- **ViewModel:** it is described in a `.ts` file. Here, DI is used to get instances of Services, and properties are specified to be used in the templates. If the property changes, the change is instantly propagated to the Template.

3.2.2 React

React is a JavaScript library for building user interfaces, that is developed and maintained by Facebook [29]. The main concepts are:

- **Declarative** - There is a designed view for each state of the application, React will update and rerender the components efficiently, if the data of the app changes.
- **Component-Based** - Components are encapsulated from each other, can have their own state, and can be composed into tree structures.

In contrast to other complex MVC[1] frameworks, React targets only View. It means that in almost every React application, other frameworks and libraries, such as libraries for routing, state management, or communication with APIs, have to be added.

Rendering Elements [30]

In the HTML file, there should be a div element, where the app will be rendered. In the case of `<div id="root"></div>`, the app can be rendered by calling a render function, where `App` is the main component of the app:

```
ReactDOM.render(App, document.getElementById("root"));
```

Components [30]

Components are written in JSX - a HTML like syntax. The component is then compiled to `React.createElement()` functions. These "raw" functions can be also used, but JSX makes the code much more readable.

The main concept of the component is to render a view based on passed props from the parent component. These components can be nested and composed into the full view of the app. In React, data flows down from the parent component to its children.

The component can have a state - for example, if the component has been clicked. If the component does not have an internal state, we refer to it as to a *"Stateless Component"*.

The component can be written as a function returning a JSX Element, or as a Class with a render method returning a JSX Element.

Component Lifecycle [31]

React Class Components goes through certain stages during the mounting process. The main lifecycle methods during a component mount are:

- **constructor()** - is called during initialization. The initial component state is set here.

[1] MVC = Model View Controller, a type of application architecture

- **render()** - returns an element based on props and component state. The render method should be pure, i.e. it does not modify the state of the component.
- **componentDidMount()** - is called only once after the component has been mounted. API calls are made from here.

Hooks API

React Hooks let us use state and other React features in functional components without writing a class component.[32]

It is basically a functions, whose name starts with "use" (for example `useState`), and that returns the desired part of the functionality. This function is called from the component and should not be triggered inside loops or inside `if` conditions.

The basic hooks are:

- **useState** - brings local state management to functional components
- **useEffect** - similar to `componentDidMount` function on class components, manages component lifecycle
- **useContext** - enables sharing of state between nested component without passing props through all of them

The hook API has been taken over by many other React packages and dependencies, and now they offer similar functionality.

3.2.3 Choosing the Right Framework

Both of the above-mentioned frameworks can be used to build a PWA. Both support rich client-side functionality and are very efficient both in terms of development time and runtime performance. Angular is more complicated than React, so the learning time is longer. Typescript has to be used with Angular, but it is optional in the case of React.

In the upcoming implementation, I decided to use React in combination with TypeScript because of these (partly biased) reasons:

- I have worked with both of these frameworks and React suited me better thanks to the broader possibilities when choosing particular libraries and approaches to certain parts of applications
- Pingl (the PWA I will implement) uses React in other parts of the platform, so using the same framework make sense in terms of code sharing and reusability. Also, all these parts of the platform are simultaneously developed, so switching between projects is easier if they use the same framework.
- Other team members have no experience with Angular, so choosing Angular can have a negative impact on future development

3.3 Google Workbox

Google Workbox is a set of libraries, that can be used to easily implement functionality in Service Workers. These libraries are distributed either as .js files through CDN, or as node modules and javascript packages and can be bundled directly into the Service Workers.

Basically, these libraries allow developers to implement in a few lines of code many of PWA features, such as routing, caching strategies or background synchronization.

3.3.1 Routing & Caching

Using `registerRoute` function, a route (i.e. URL of a request) can be easily mapped to a handler. The route can be distinguished by its full name, a regular expression or by a callback function. The handler can intercept the request and use some of the cache strategies, or can perform any other tasks and return a `Response` wrapped in a `Promise`.

Bellow, there is an example, that opens a cache called "images" and saves there last 200 images for 30 days. Then, all images are returned from this cache using `CacheFirst` strategy:

```
workbox.routing.registerRoute(
   /\.(?:png|gif|jpg|jpeg|svg)$/,
   new workbox.strategies.CacheFirst({
      cacheName: 'images',
      plugins: [
         new workbox.expiration.Plugin({
            maxEntries: 200,
            maxAgeSeconds: 30 * 24 * 60 * 60, // 30 Days
         }),
      ],
   })
)
```

It is important to remember, that some requests reaches third-party sources (other origins). These cross-origin requests are opaque to the JavaScript, which means, that some details of the responses are not readable. Especially in Service Workers, it is not possible to check status codes of the responses. This means, that even if an error response comes from the server, it could be saved in the cache and our app will continue to receive this error response. Because of that, by default, Workbox disables `CacheFirst` strategy for these cross-origin requests and allows only `NetworkFirst` and `StaleWhileRevalidate` strategies, which ensure frequent updates of the cache.

3.3.2 Precaching

Very important feature is preloading of resources. This is typically done by function `precacheAndRoute`. It accepts a list of objects containing URL of the resource and its revision number:

```
workbox.precaching.precacheAndRoute([
   {url: '/index.html', revision: '383676...233' },
])
```

This list of resources is often generated automatically by Workbox CLI interface or WorkboxBuild Node module and is injected into the Service Worker file using `workboxBuild.injectManifest()` function.

3.4 Testing with Lighthouse

To test if the website has implemented PWA functionality correctly, Lighthouse tool can be used [33]. This tool can be run from the command line, or it is available in Google Chrome browser DevTools. It audits several aspects of the website, such as performance, SEO (Search Engine Optimisation), best practices, accessibility, and PWA aspects. In the case of PWAs, it namely tests, if the web page has registered a Service Worker, works offline or has an optimized view for different viewpoint sizes.

Chapter 4

Pingl.app - PWA

To demonstrate PWA functionality, I will develop a part of Pingl application. Pingl is a product of the same-named company Pingl s.r.o., which I co-founded in February, 2019.

> **Pingl.app description:** *Thanks to Pingl, gastro business customers can order and pay without waiting, whether they want to order a takeaway for a certain time or they are already sitting at a restaurant table.*
> *For restaurants, Pingl means a new efficient sales channel - Pingl will bring new customers through takeaway orders while reducing service demands.* [34]

4.1 Project Scope

One of the key parts of the Pingl platform is a customer-facing app.
This part can be divided into two sections:

1. **Choosing the restaurant to eat** - the current design is not sufficient for current needs. It has to be changed to better showcase restaurants, which users can visit.
2. **Choosing a meal and placing the order** - this part of the app is already implemented and has a huge logic, which makes it hard to realize in this project. I will omit this part of the app in this project.

Currently, the customer-facing part of the application is a website built in React. It lacks any PWA functionality, and it is in question if the app should be implemented as a PWA or as a native app. This work aims to test the first option - implement a PWA and conclude whether this approach was efficient, and the rest of the app should also be converted into a PWA.

Out of scope functionality:

This project will be limited to the PWA itself, so only a frontend side will be implemented. If any requirement for the backend side will occur, it will be implemented by other persons in the Pingl team.

Meal ordering and payment are not implemented in this project. The main focus is on the PWA functionality in a narrow part of the application.

4.2 Requirements

4.2.1 Functional Requirements

FR_01: As a customer, I want to see a list of all available restaurants, so I have an overview of all of the possibilities to eat.

FR_02: As a customer, I want to sort restaurants by my preferences, which include distance or type of served food, so I can easily choose a place to eat.

FR_03: A a customer, I want to see restaurant on a map, so I know, where they are located.

FR_04: As a customer, I want to see restaurants offering special discounts.

FR_05: As a customer, I want to see specific meals around me, so I can decide by available meals, not only by restaurant names.

FR_06: As a customer, I want to be able to scan QR codes on tables in restaurants, even if I don't have a dedicated scanning app.

FR_07: As a customer, I want to login to save my preferences, so I will have a personalized experience while browsing the app.

FR_08: As a customer, I want to use the app in multiple languages, mainly in Czech and English.

FR_09: As a customer, I want to see a list of restaurants even when I am offline.

4.2.2 Non-Functional Requirements

NFR_01: The app is compatible with modern browsers, including both desktop and mobile platforms.

NFR_02: The app is responsive and adapts to different screen sizes, ranging from 4-inch mobile phones to large desktop monitors.

NFR_03: Users can add the app to their home screens.

NFR_04: The app is written in React to match other parts of the platform.

NFR_05: The app uses the current backend, which provides a GraphQL API.

NFR_06: The app design is in accordance with Pingl design guidelines - the logo and brand colors are correctly used.

Chapter 5

Pingl.app - User Interface

5.1 Screens

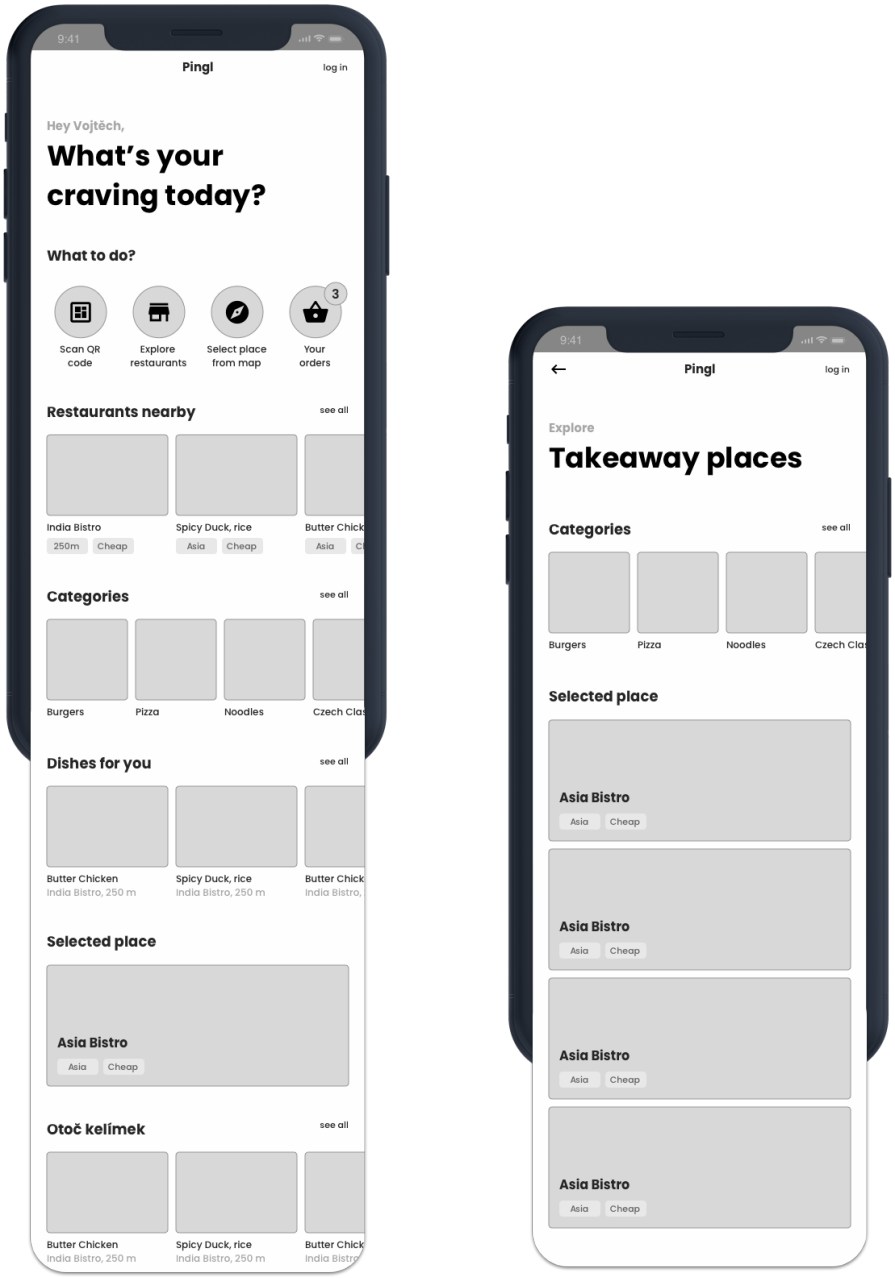
All screens that will be in the app are listed bellow:

- **Landing Screen** - to browse a list of meals and restaurants
  Wireframe: *Pingl.app Mobile Wireframes - part 1 (a)*
- **List of Restaurants** - to see and explore restaurants by categories
  Wireframe: *Pingl.app Mobile Wireframes - part 1 (b)*
- **Map View** - to select a restaurant from a map
  Wireframe: *Pingl.app Mobile Wireframes - part 2 (c)*
- **QR Code Scanner** - to scan QR code when the user is in a restaurant
  Wireframe: *Pingl.app Mobile Wireframes - part 2 (d)*
- **User Profile** - to see basic information about user, to enable notifications

5.2 Wireframes

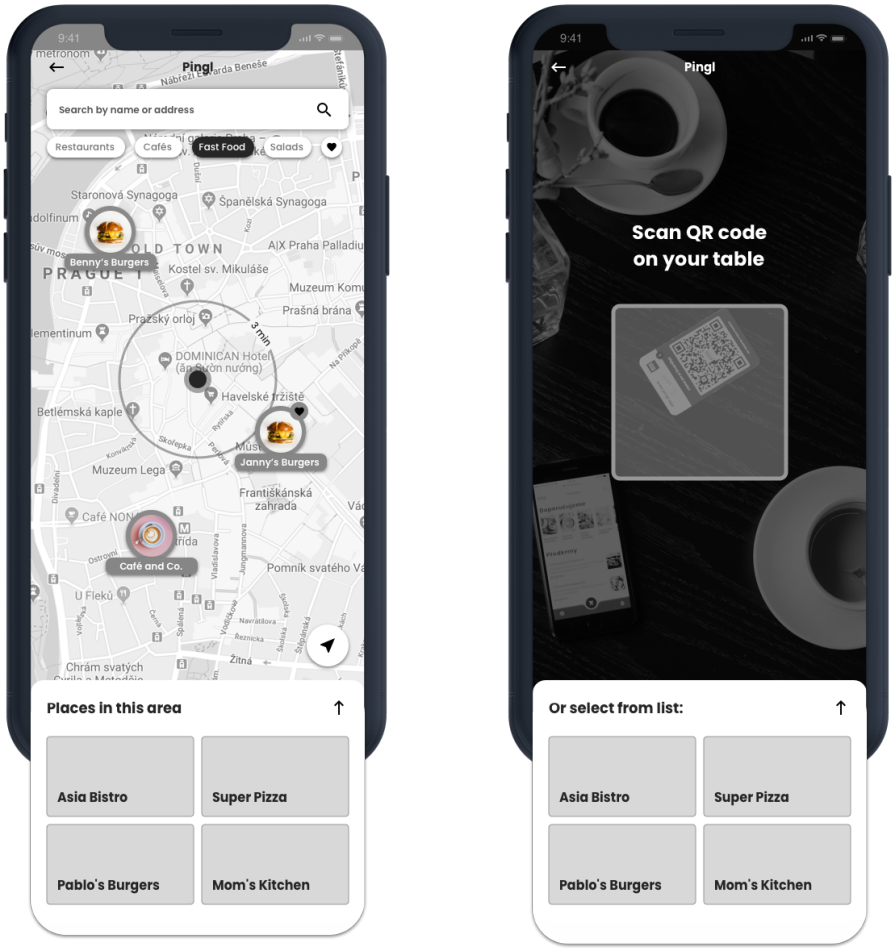
To demonstrate the app design and functionality, I have created a few wireframes. A wireframe is a bare-bones depiction of all the components of a page and how they fit together.[35] These wireframes are designed in Sketch, which allows rapid prototyping directly to the mobile phone. Since the majority of users will use mobile phones to access the app, I have used a mobile-first approach. For the desktop version, the intention was to keep as many elements as possible same with the mobile version, to minimize changes between both versions.

I have used a combination of high fidelity prototyping in grayscale colors with some more concrete design elements, such as icons and shadows. The result wireframes and clickable prototype can be considered almost as the final design without colors. Lack of colors and images in these wireframes allows me and beta testers to focus more on the layout and action flow.



(a) : Landing Screen **(b) :** List of Restaurants

Figure 5.1: Pingl.app Mobile Wireframes - part 1



(c) : Map View **(d) :** QR Code Scanner

Figure 5.1: Pingl.app Mobile Wireframes - part 2

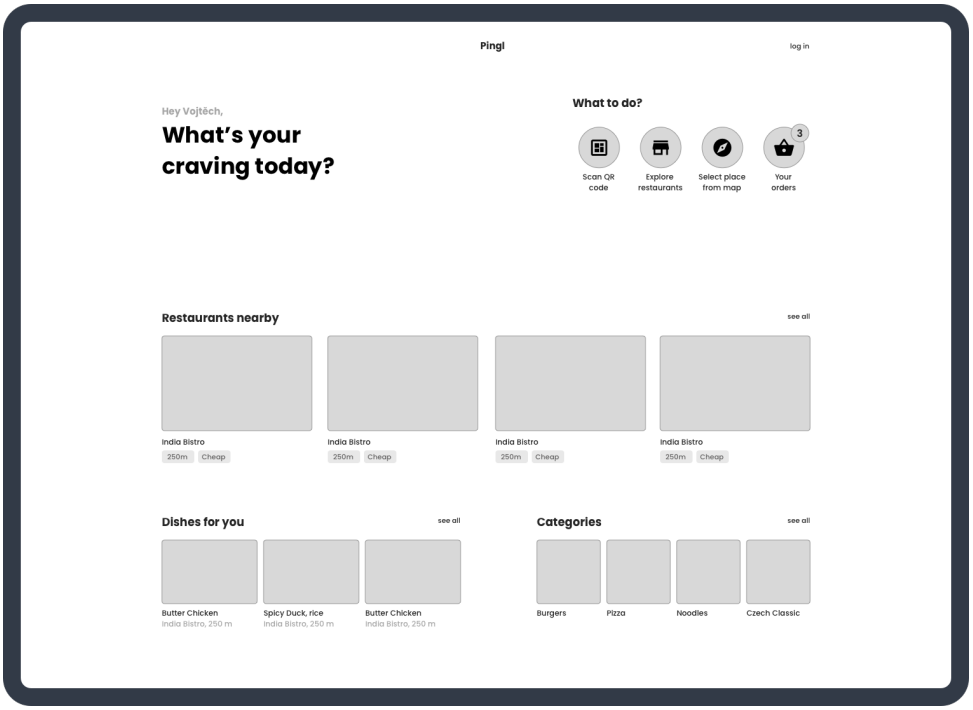


Figure 5.2: Pingl.app Desktop Wireframe

Chapter 6

Pingl.app - Implementation

I have developed a production-ready part of Pingl.app as a PWA. The code of implemented prototype is available in the enclosed CD or in a GIT repository: `https://gitlab.com/vojtech.rychnovsky/pingl-marketplace`.

The project consists of:

- a setup of React project based on TypeScript, with an emphasis on code quality tools, and with CI/CD pipelines - *Project Setup (section 6.2)*
- a GraphQL client capable of offline work - *GraphQL API Client (subsection 6.3.3)*
- a set of responsive pages and components allowing users to see restaurants in a list or on a map, filter them by categories, search by their name or to select them by scanning a QR code - *Implemented Functionality (section 6.4)*
- a PWA, that is reliable (works offline), fast (uses advanced caching) and engaging (has push notification) - *PWA Features (section 6.5)*
- testing - *Testing (section 6.6)*

Compared with the designed user interface, page "Your orders" has not been implemented. The reason is, that as placing of new orders is not in the scope of this projects, this page would be always empty.

Steps to run the React project locally:

1. `npm` (Node Package Manager) has to be installed on the device
2. open a command line window or a terminal window in the project root folder and run these commands:
   - `npm install`
   - `npm run start`
3. open `http://localhost:3000` in any browser, the website is ready to be viewed there

6.1 Deployment Architecture

6.1.1 Backend

The current Pingl backend in a development environment is used. During the project, there were raised some new requirements for the backend, such as a new query for sending test push notifications to user's devices. These changes on backend were implemented by other members of Pingl team.

6.1.2 Frontend

The frontend part consists of one PWA client app. As some parts of the PWA have to be tested on a real domain with HTTPS connection, I have prepared a separate domain for this app. As a hosting, I have chosen Firebase Hosting for the following reasons: the PWA itself is compiled to static files and can run here in a free tier with no costs, and also other Firebase products are used in this project, so I wanted to keep the project unified in one environment. The app is deployed here: `https://thesis.web-rychnovsky.com/`.

6.2 Project Setup

6.2.1 Used Frameworks and Technologies

The project uses React as the main framework. The base configuration comes from the `Create React App`[1] template. Static type checking is provided by TypeScript. I have paid attention to type all functions and objects correctly to fully benefit from TypeScript type checking.

As mentioned in the React analysis, it is necessary to add other libraries for particular parts of the functionality. These libraries come as dependencies from `NPM` (Node Package Manager) repository. After installing, they are listed in `package.json` file in the project root. As some of these dependencies are written in JavaScript, this requires additional libraries with TypeScript types definitions. Below are listed the most important and interesting ones:

- **Routing** - As a Router, `BrowserRouter` from `react-router-dom` package is used. This type of router distinguishes routes based on the page URL (address in the address bar in the browser) and, based on the URL, renders a correct component.

- **Global State** - The global state of the app is managed using `react-redux`. This state is persisted in the LocalStorage in the browser. More in *Redux Store (subsection 6.3.2)*

[1] Create React App: `https://github.com/facebook/create-react-app`, (accessed: Jan 02, 2020)

- **Async tasks** - For more complex asynchronous tasks in the app, `redux-thunk` is used. In thunks, more `redux actions` and `GraphQL API` calls can be chained to build the app logic. More in *Redux Store (subsection 6.3.2)*
- **API** - To connect to the remote GraphQL (gql) server, `apollo` client is used. More in *GraphQL API Client (subsection 6.3.3)*
- **Configuration** - The app configuration, such as backend URLs, depends on the environment. These environment variables are listed in `.env` files and are loaded by `env-cmd` package. To ensure that all of these variables are presented in the `.env` file, they are wrapped into a config object, which checks for missing variables during compile time.
- **Styling** - `styled-components` defines the base `HTML` elements with their corresponding styles. Some components, such as icons and inputs fields, are used from `@material/core` framework and are customized to match the design of the app.
- **Forms** - `formik` together with `yup` validation is used to manage forms submission and validation
- **Testing** - for running tests, `Jest` package is used. More in *Testing (section 6.6)*

6.2.2 Code Quality Tools

To ensure a high standard of code quality through the app, static code analysis is used. These tools, together with TypeScript, check for issues and problems in the code and minimize runtime errors.

- **ESLint** - checks for problems in JavaScript and TypeSript code
- **StyleLint** - checks for problems in `CSS` code in `StyledComponents`
- **Prettier** - ensures uniform code formatting across the whole app

All these tools are configured to be run as git commit hooks. I have configured git hooks by `Husky` package. Using this tool, the hooks can be defined in `package.json` file and can be shared with the development team. This process ensures that every commit is checked that it follows all these rules. The rules are, if it is possible, auto-fixed.

6.2.3 CI/CD Pipelines

For deployment and testing, automated pipelines on GitLab are used. They run on every push to `dev` and `master` branches with following stages:

- **Build** - installs dependencies and builds the app
- **Test** - runs all tests defined in the app
- **Deploy** - deploys compiled files to Firebase Hosting (only master branch)

6.2.4 Folder Structure

- `src/components/` - contains simple reusable components
- `src/containers/` - contains more complex components with side-effects
- `src/constants/` - contains route names and server error codes
- `src/data/` - contains GraphQL queries and Redux Store: actions, selector, reducers and thunks
- `src/pages/` - contains individual components for every page
- `src/static/` - contains static files such as images and translations
- `src/styles/` - contains global app styles, animations, colors and typography
- `src/utils/` - contains small utility functions and helpers, for example for distance meassuring, notifications or Local Storage
- `src/@types/` - contains types declarations
- `serviceWorker/` - contains Service Worker file and build script for its compilation
- `public/` - contains the base `index.html` file, PWA manifest and favicons

6.3 Frontend Architecture

6.3.1 Components

Each component is located in its own folder containing two files. The component logic is placed in `index.tsx` file, the styles are separated in `styled.ts` file. Every component is written as a Functional Component typed as `React.FC` with a heavy usage of `React Hooks`. Except of standard React Hooks (such as `useState`, `useEffect`, `useMemo`), they are used, for example, to get an instance of translator function, to access the app history, to dispatch Redux `actions` or to select information from the Redux Store.

The main App component contains a switch between individual pages. Each page is wrapped in `AppRoute` component responsible for protection of private pages. The entire app is wrapped in an `ErrorBoundary` component, which catches errors in the underlying component tree. If an error occurs, a fallback component is shown - there is an option to reload the page or to clear the `Local Storage`. By these two buttons, the user can help the app to recover from many errors.

Layout Components

To facilitate creation of new pages (routes) with unified layout, there are three base layout components. They can be used by any page in the app, all of them are fully responsive and offer further customization by passing different props:

- **PageLayout** - a basic page with header and container with a maximum width, used in the main app screen
- **HalfPageLayout** - a page with a big image on the left and a column with a content on the right. On mobiles, the image is hidden, and the right column is full-width. This layout is used in the login page
- **TwoColumnPageLayout** - a page, which has two columns, one of them becomes a bottom bar on mobiles and can be expanded by swiping up. This layout is used in the map and scanner page.

6.3.2 Redux Store

The Redux Store is divided into four sub-stores based on their responsibility:

- **App** - stores location of the device together with a timestamp of this location
- **MenuItems** - stores an array of specially-offered menu items from each restaurant. These arrays are stored in an object indexed by ID of the restaurant
- **Restaurant** - stores an array of all available restaurants and all available tags (categories). Also, information about selected restaurant, meals and discount is kept here.
- **User** - stores information about the current user

Each of them contains a `constants` file defining allowed action types, `actions` file defining actions, which can be dispatched in the components, and `reducers` modifying the store based on the action types. Store `selectors` are also defined in their own file, they simply retrieve values from the store and sometimes make data transformation.

Asynchronous actions are handled by `Redux-thunk` middleware. The most common use-case in this app is to try to fetch data from the API, to dispatch an action to save the data to the store and to handle any possible error. To ensure correct typing of responses returned from the thunks to the component, custom hook `useReduxDispatch` is used. This hook returns `dispatch` function, that is able to run a thunk and return its return value correctly. This process is illustrated by diagram *Asynchronous data fetching with Redux thunks and GraphQL API (Figure 6.1)*.

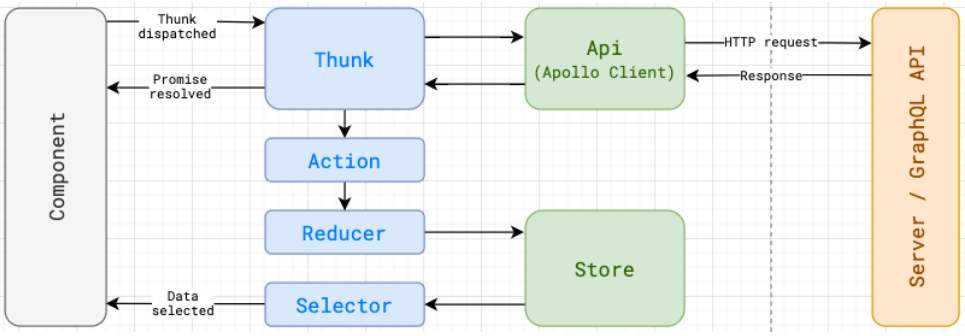


Figure 6.1: Asynchronous data fetching with Redux thunks and GraphQL API

6.3.3 GraphQL API Client

Each of the individual Redux Stores also contains `GraphQL queries`, `mutations` and `fragments` related to its scope. From these queries, typescript types are generated using `Apollo client` (sripts `npm run schema:fetch` & `npm run schema:generate`). By this process, all these queries are validated against current application server.

The query and generated types are then combined together to form a fully-typed API method, starting with "request" as a naming convention. There is a generic api function, which makes this process very easy - only return type and type of variables has to be passed. This is an example, how this can be done for sign in request. As a result, the function `requestSignIn` now returns Promise, that resolves to an object of the same type as the return type of `gql` mutation:

```
import { signIn, signInVariables } from './gql/__generated__/signIn'
const SIGN_IN = gql`
   mutation signIn($username: String!, $password: String) {
      signIn(username: $username, password: $password) {
         // ... returned attributes
      }
   }
`

export const requestSignIn = async (variables: signInVariables) => {
   return (
      await gqlApi.apiMutateRequest<signIn, signInVariables>(
         SIGN_IN,
         variables
      )
   ).signIn
}
```

Apollo Client Middlewares

Internally, these API calls use `Apollo Client` to form and send HTTP requests. Each request is process through several middlewares (also called

`links`). The order and composition of these links is illustrated in diagram *Composition of Apollo Links (Figure 6.2)*. Each link is responsible for handling different task:

- **AuthLink** - adds authentication token to the header of each request
- **HttpLink** - connects to the app server using HTTP requests (for mutations and queries)
- **WebSocketLink** - connects to the server using web sockets (for subscriptions)
- **RetryLink** - retries the request N-times in case of network error
- **QueueLink** - if the client is offline, saves the request in a queue and sends it when the client is online
- **split** - a basic "if" condition, can split chain of links to two branches

These links combined can be used to send requests while offline. Based on the configuration of each request, it can wait until the client is online (e.g. an unimportant request such as fetching restaurants on the homepage), or can return an error (e.g. an important request such as sign in)

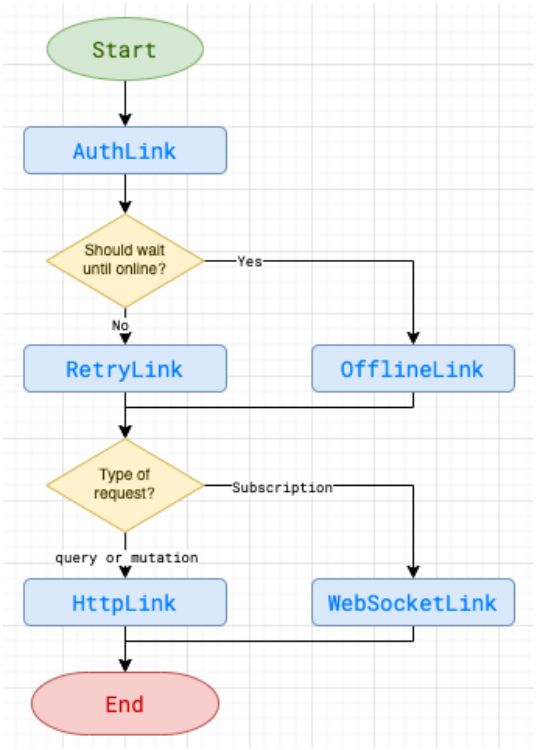


Figure 6.2: Composition of Apollo Links

6.4 Implemented Functionality

The implemented features correspond to the functional requirements defined in *Functional Requirements (subsection 4.2.1)*. Generally, the app offers a nice, intuitive and responsive interface for restaurant selection.

User Authentication

The app features standard login and registration functionality. The user is authenticated by email and password, these credentials are validated on the app server, and if they are correct, an `access_token` with limited expiration is returned. This token is saved in `LocalStorage` and then is send as a `Bearer token` in the `Authentication` header with every request to the server.

The app routes are split to private a public routes. All of them are wrapped into `AppRoute` component. If the route is marked as private, the page is rendered under two conditions:

- `access_token` is available in local storage
- `user` object is saved in redux store

At the same time, the `access_token` is validated against the application server. If the token is valid, it is refreshed and new user data are fetched. Otherwise, the user is logged out and redirected to the homepage.

The login and registration forms are split between two separate pages. There is a link to the registration form in any page header. If the user has ever been logged in, the link directs to the login form instead of to the registration form.

Logged in users can access a profile page. There they can see their name, email, and available payment methods (these methods are filled after the user places an order in the real Pingl.app).

There is a switch between available languages. Currently, the app supports Czech and English using `react-i18n` package. The translations are separated from other code in `JSON` files, so it is straightforward to add new languages or to edit these strings by external translators. The default language is set to browser default language with a fallback to English.

Users can also enable location services there. The app asks for permission, and if it is granted, the user location is saved to redux store. This location is then in the next 24 hours used to sort restaurants by distance.

The last setting is related to Notifications. The user has an option to enable them and to send a test notification. If the user has denied permission to send notifications, a prompt to enable them in browser settings will appear. The technical aspects of how are notifications implemented, are discussed in *Push Notifications (subsection 6.5.2)*.

There is also a logout button that deletes `access_token` from the `LocalStorage` and resets redux store, then the user is redirected to the main page.

Restaurant & Meals Lists

Two layouts were developed to showcase restaurants and meals. There is a basic grid component, which can be customized by passing a number of columns for mobile and for desktop view. The second component is a horizontal carousel built on top of `react-slick` library. Similarly, the number of columns can be specified by props. Users can go to the next slides by swiping on mobile or by clicking on arrows on desktop.

These two components, grid and carousel, accept an array of restaurants or meals thumbnail, that makes them easy to reuse in any part of the app. Moreover, in the future, if there will be a need to show a different list of restaurants, for example, sorted by customer rating, this can be done very quickly by reusing these components. Now they are used to present a randomly selected restaurant, a list of all restaurants, a list of nearby places, as well as a list of special meal offers from all nearby places.

To show details of meals or restaurants in the grids or carousels, thumbnail components are used. There is visible a name, an image, tags, conditionally a distance. Using only one component to show a restaurant or a meal in the app ensures consistency of the design and the code reusability. These components are ready in three sizes. The thumbnail components also encapsulate the logic of selecting a restaurant. If the user clicks a thumbnail with "to table" orders, a dialog window with input for table number appears. Otherwise, the user goes directly to the restaurant detail. On this restaurant detail page, there are visible the selected data - restaurant, table, and meal. This page is there only for testing purposes and will be replaced later after merging with the other part of the Pingl app responsible for placing orders. All selected data there are saved in the store, so the connection to restaurant detail, where a meal will be optionally selected, is straightforward.

Usage of these components is illustrated by image *Sections of the Main Page (Figure 6.3)*.

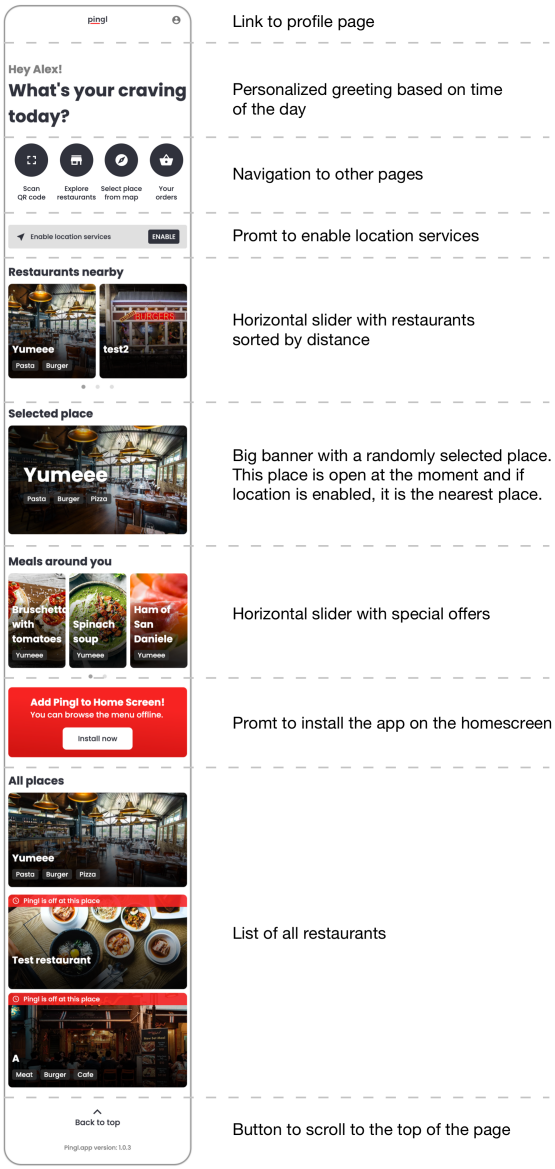


Figure 6.3: Sections of the Main Page

QR Scanner

This component uses `react-qr-reader` package to handle image processing, then the scanned data are passed to a parsing function. There are currently 3 types of QR codes illustrated by *Possible types of QR codes (Figure 6.4)*:

- **table** - containing table ID and a discount code
  `https://example.com?id=xxxx&discount=yyy`
- **marketing** - containing specification of the marketing source (these values are not saved, they are used only for web traffic analysis)
  `https://example.com?referral=name&type=takeaway_promo`

- **restaurant** - containing a short name of the venue
  `https://example.com/yummeee`

(a) : Type "Table" **(b) :** Type "Marketing" **(c) :** Type "Restaurant"

Figure 6.4: Possible types of QR codes

Values from these codes are parsed and validated, then saved in the redux store. In case the user opens these URLs directly in a browser, the app handles the parameters also in the main App component and performs the same actions. It means that the codes can be scanned with third-party scanners, including native camera apps, with no difference in functionality.

If the user is offline, the QR code is validated against cached restaurants data. If the code is found invalid, a message that Internet connection is needed, will appear. This can happen, for example, if a new QR code was placed on a table today, but the cached data are older and does not contain this particular table. In this case, a server validation is needed. But for the vast majority of cases, codes can be scanned offline.

On the side of the scanner page, there is also a list of restaurants filtered to show only places offering table ordering, so in a case users have problems with using the scanner, they can select the restaurant by clicking on it.

Map

The map page shows the locations of all restaurants. `Google-maps-react` package is used to show a fully functional Google map component. There is a custom "locate me" button to get the current location from Navigator. Restaurants are placed on the map as pins with their images. In the middle, there is a circle showing a radius from the map center with a distance label, so it is easy to see the distance between restaurants. There is a pulsing `CSS` animation on the center mark, and the circle radius is animated as the map is dragged.

If the user is offline and the map cannot be loaded, a message about this issue will appear. Since the map doesn't show only one location, but many of them, static preloaded image of the map doesn't make sense.

Next to the map, the list of nearby restaurants is grouped into two parts - one for the places visible in the map, the other to show places outside of the

map borders. This list is sorted from the closest places to the map center to the furthest. As the map moves, this sorting and grouping are updated.

6.5 PWA Features

The implemented website is a Progressive Web App. It meets the prerequisites for installation on the home screen. Users are asked to install the app by the browser, or have an option to install the app in various places in the app - for example, if the browser can install the app, a banner between carousels with restaurants on the main screen will appear.

The app has also all the necessary visual elements to display on mobile and desktop platforms correctly. For example, iOS splash screens (loading screens) can be generated using `pwa-assets-generator` package (`npm run splash:generate`), the resulting images (more than 40) are placed in the `/public` folder.

In this section, I will cover some of the PWA functionality in more detail.

6.5.1 Service Workers

Create React App has its own service worker based on `Google Workbox`, a framework for building service workers[2]. Currently, there are no possibilities for configuring this default service worker in `CRA` project, so I had to create another build script, which basically creates a new file and replaces the old file. The service worker is compiled from TypeScript to JavaScript using `Babel`, then `Workbox` is used to inject static files urls, which will be cached by the service worker. This build process can be run by `NPM` scripts and is part of deployment pipelines.

The main task for this service worker is to (pre)cache files and assets of the app using different caching strategies[3]:

- **index.html and JS bundle** - all the app static assets are injected to this worker during build process with coresponding revision hashes. These files are precached using `Workbox precacheAndRoute` method. As this app is a SPA (single page app), the application shell (index.html) is used by multiple app routes, so all routes are mapped to this one index.html file.
- **fonts** - cached using cache-first strategy
- **restaurant images** - cached using stale-while-revalidate strategy
- **meals images** - cached using stale-while-revalidate strategy
- **other images** - cached using cache-first strategy

[2]Detailed description of how Workbox works, can be found in *Google Workbox (section 3.3)*

[3]Different caching strategies has been described in *Caching Strategies (subsection 2.3.1)*

- **other .js and .css files** - cached using stale-while-revalidate strategy

The process of distribution of updates of the app and service worker is simplified into these steps:

1. with a new app build, a new service worker is generated with a different revision number for each of the resources
2. when the user opens the app, this service worker file is downloaded, compared with the old one and as it is different, it is installed and page reload is forced
3. because revision numbers of the resources have changed, they are removed from the cache and downloaded again
4. finally, the app is ready to be used, with full offline support

There is also a second service worker, responsible for receiving push messages, that is registered by Firebase Cloud Messaging.

6.5.2 Push Notifications

Push notifications are powered by Firebase Cloud Messaging (FCM). The subscription process uses `firebase/messaging` package. After the firebase messaging is initialized using API keys, the whole process of establishing the subscription is abstracted under one function - `getToken()`. This function is responsible for asking for the permission, for registering a new service worker and for starting the subscription. It also saves the subscription details on the FCM server and simply returns a FCM token. This token is then used as an "address" of this device. Then a request to Pingl application server is send to save this token, paired with current user id. As one user can have multiple devices, an unique device ID is generated and it is saved together with this token. From this moment, the application backend can send push notifications to this device.

To receive the messages, two handlers are used - one in the service worker receives the messages while the app is in background, the second handles the push messages while the tab with the app is visible and focused. In these handlers, when the message is received, push notifications are shown. Currently, 3 push notification types are supported (this type comes in a payload of the message):

- **TEST** - a test notification, it can be triggered from user profile
- **ORDER_READY** - a notification when an order is done and ready for pickup
- **NEW_RESTAURANT** - a notification after a new venue is added to Pingl (not implemented on backend yet)

This implementation will be probably changed in the future. Now the title and body of the notification is hard-coded in the frontend based on the message type, which makes it hard to implement new message types or to add personalization. Better approach would be to receive these texts for the notifications directly from the backend.

6.5.3 Offline Support

The service worker caches static JavaScript and HTML files, Google Fonts and images of restaurants in `CacheStorage`. The state of the app is persisted in local storage, so the app is fully accessible even without Internet connection. Users can open the app and see the list of restaurants while being offline. However, behaviour of certain parts can differ to the online state:

- **List of restaurants & meals** - restaurant data may not be completely up to date, but the list is visible even with restaurant images
- **QR Scanner** - if the client is online, the codes are validated against the app server, however, during offline state, they are validated against cached data in redux store
- **Map** - map itself is replaced by an offline message, the list of restaurants works
- **Profile** - button for sending test notifications does not work
- **User registration** - works only online

As the user navigates through the app, many HTTP request are triggered, such as fetching of all restaurants, fetching of user data, or verification of user credentials. As it was described in section *Apollo Client Middlewares (subsection 6.3.3)*, some requests are postponed until the client is online. This means, that the user will see old data from cache, but once the Internet connection is restored, the data will be updated. Other requests, such as submission of sign in form, will skip this queue and will be send immediately, with 3 repeats in case of error response.

These two techniques are used to overcome temporary connection issues and to ensure, that the app is usable without the Internet connection.

6.6 Testing

This section refers to the "technical" aspects of testing. Testing of user interface and user experience is describe in a dedicated part of this project: *Pingl.app - User Testing (chapter 7)*.

6.6.1 Testing with Lighthouse

As it was mentioned in previous sections (*Testing with Lighthouse (section 3.4)*), the Lighthouse Chrome extension is very valuable source of information about the page performance and possible issues. This tool helped with fixing various issues related to accessibility, SEO (Search Engine Optimization) and best practises, such as missing link labels or small buttons for mobile devices.

The only area in this test with insufficient results is performance - 20/100 rating means that that are several problems regarding page load time, image compression or precaching of images. These issues can be caused by preloading high amount of resources in service workers for offline use, but this has to be further investigated, which is done in *Performance Testing (subsection 6.6.3)*.

6.6.2 Unit Testing

Certain parts of the code are unit tested. These tests are based on `Jest` testing framework. They can be run locally by script `npm run test`, and they are part of CI/CD pipelines. The tests are divided into 10 test suites based on the tested files (one test suite for one file), together they form 35 tests. Namely, the tested parts of the apps are:

- **Location Utils** - tests functions for measuring the distance between two places and restaurant sorting
- **Notification Utils** - tests functions for generating device ID
- **QR Code Utils** - tests parsing data from URL, that is scanned from QR codes
- **Time Utils** - tests calculation of the next close time of a restaurant
- **Redux Store** - All reducers are tested if they return the right result for a passed action. There are also tests for selectors and thunks.

These functions were chosen for the testing because they form the core functionality of the app. Additionally, from my experience, these are the functions that cause a lot of bugs and issues and, at the same time, can be very easily tested.

6.6.3 Performance Testing

The implemented PWA was also tested in terms of performance. There are two main areas users are concerned about - the initial load time and the smoothness of application.

App Smoothness

In this app, there are no heavy computations or complex animations, so the slowest parts of the app are HTTP requests to the app server. The average server response time is under 250ms, so even this is not an issue for the user.

Page Load Time

The more important area is the load time. The PWA features advanced caching mechanisms, so it is important to test, whether this caching affects the resulted experience.

The tests were conducted in a consistent environment: desktop Google Chrome browser was used, with an emulated "fast 3G" connection. The first load time was measured with disabled browser cache and with cleared other browser data (such as service workers, local storage). This is how the app would feel like for a new user. The second load time was measured with enabled browser cache and with the content of local storage and other parts untouched after the first load. This simulates a returning user.

For comparison, the app was tested with and without the service workers. Each test was run 5 times and times were averaged.

		first load	second load	improvement
no SW	load time [s]	3,51	1,71	51 %
	finish time [s]	13,99	5,81	58 %
with SW	load time [s]	3,54	1,00	72 %
	finish time [s]	13,10	2,68	80 %

Table 6.1: Results of load time testing

These results are also illustrated in a chart *Page Load Times (Figure 6.5)*.

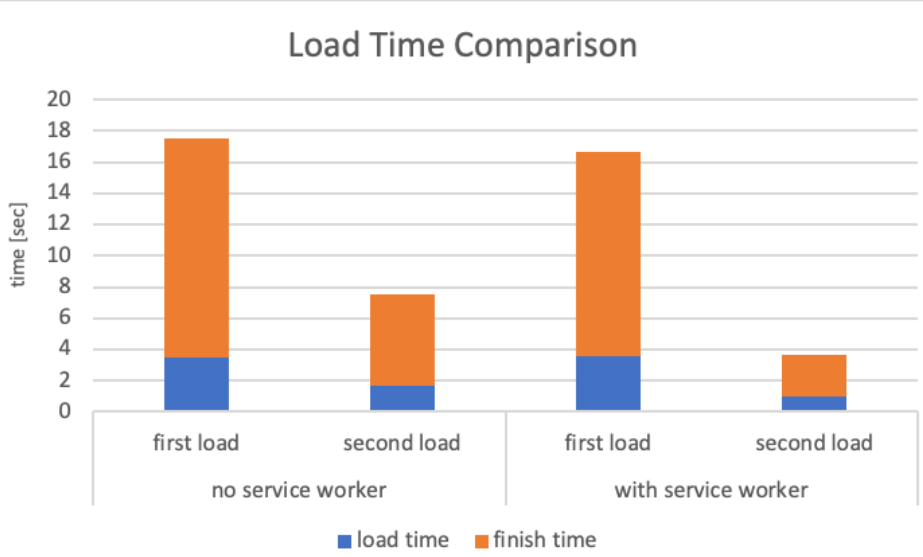


Figure 6.5: Page Load Times

The performed tests show that there is no difference in the load time of the first visit between the versions with and without the the service worker.

However, for the second visit, both versions shows a significant improvement.

In the version without the service worker, the images are cached in the browser cache and the app loads 58% faster. With the service worker enabled, even more assets are cached and there is another improvement - the page loads **80% faster**. This clearly shows that service worker brings another acceleration of the page load.

If the user leaves this testing environment and uses full Internet connection, the average load time is around **1 second**.

Chapter 7

Pingl.app - User Testing

The resulting app has been user-tested. In particular, usability tests has been performed with a group of 5 participants in an informal environment.

7.1 Preparation

7.1.1 Target Group

Participants of this test have been chosen in accordance with the target audience of Pingl app. These users should be in age of 15 - 50 and should use mobile phones on a daily basis.

7.1.2 Environment

The tests have been run in an informal environment over online video calls or in person. Each of the participants used their own smartphone, because this is exactly how they would use the app in the real world. They were asked to fulfill simple tasks in the app, while their thoughts and impressions were noted down.

7.1.3 Pre-Test Screening

Before running the test, each participant had to fill a screener to prove his/her qualification for the test and to collect some demographic data. The results are shown in table *Results of the survey before user testing (Table 7.1)*. Number in a separated column represents quantity of each response.

Question	**Answers**					
gender	male	4x	female	1x		
age	15-25	4x	25-35	0x	35-50	1x
restaurant visits per week	0-1	2x	2-3	2x	4+	1x
have ever ordered food online	Yes	4x	No	1x		

Table 7.1: Results of the survey before user testing

7.2 Test Cases

Each of the participants was asked to perform one by one these actions:

CASE_01: Open the website and add it to the home screen of your device.

CASE_02: Create a new account and check your profile, that you are correctly logged in.

CASE_03: Change the language of the app.

CASE_04: Tomorrow you will go to Prague. Check, if there are any places, where you can eat. Find your favourite.

CASE_05: Enable location services and find the nearest restaurant to your location and select it.

CASE_06: Scan QR code on your table.

CASE_07: Check, whether the restaurant called "Yumeee" offers burgers.

CASE_08: Logout from the app.

CASE_09: Leave the app and disable Internet connection. Go back and try to navigate through the app. Try to reload it. Does it work offline?

7.3 Test Report

Participant 1

Information about the participant: Co-Founder of Pingl, 23 years old, with a strong technical background. Testing device: iPhone 7.

Found issues: After registration and after log in, he expected to see a success response. When he opened the page with the map, he expected to be asked for his location immediately, so the map would be centered to his position. Also, the bottom bar (in the map page and scanner page), when it is opened, should be closable by clicking on its top edge. When he had the app installed, the QR scanner was not working.

Post-test discussion: Generally, he likes the app. As a comparisons to the old Pingl interface, he finds attractive multiple carousels and lists with the restaurants.

Participant 2

Information about the participant: Co-Founder of Pingl, 21 years old, with a strong technical background. Testing device: Samsung Galaxy 9 Plus.

Found issues: When he opened the app, the browser proposed installation on the home screen, he confirmed it, but nothing happened. He was not sure

if that was only a temporary issue, but the in-app banner with a button to install the app worked right after with no problems.

Post-test discussion: He loves the experience from the app. He thinks there are many improvements in comparison to the current layout and functionality in the Pingl app. He considers offline support as a very handful feature.

Participant 3

Information about the participant: 17 years old student, with experiences in design and online marketing. Testing device: iPhone 6.

Found issues: As the testing phone has a smaller screen, some elements of the UI seem to be bigger than expected. In particular, the homepage has a lot of white space in the top section, and restaurant thumbnails have too big titles and some of them are overfloating their containers. Also, there is an arrow icon on the top edge of the bottom bar, and it should change its direction, when the bottom bar is expanded. In addition, the full text search should also filter the restaurants by city or by address.

Post-test discussion: He finds the app quite nice, even though there are some details, that would need further improvements. Especially to tweak padding of some elements and their sizes on all screen sizes.

Participant 4

Information about the participant: 20 years old student without any specific technical knowledge. She is an ordinary daily smartphone user. Testing device: iPhone 8.

Found issues: For her, it was hard to find a button to save the PWA on the homescreen. Even though she knew, what a PWA is, Safari browser is not very user friendly in this area and this task was not manageable for her. Nevertheless she finished every task without much problems. During using the map, she thought that the center mark is her location and she was surprised, that it moved as she dragged the map.

Post-test discussion: In general, she considers the app to be intuitive and simple.

Participant 5

Information about the participant: 55 years old business man, daily visitor of restaurants, interested in the app as he might use it to speed up his lunch routine. Testing device: Huawei P30.

Found issues: When he was in the map, at first, he couldn't find the button to center the map to his location, but then he realized, how it works. He is satisfied with large buttons and touchable elements.

Post-test discussion: He likes that the app can be saved on the homesceen and is quickly accessible.

7.4 Results

This usability testing brought positive results. In general, participants found the app to be intuitive and user friendly and managed to finished all critical test cases. In the future, this user testing should be done on a wider audience and also should be done periodically, as some issues will be fixed and a new functionality will be added.

However, there are some problems, that were either found by more users or have a great impact on the app functionality. These issues have to be addressed and fixed:

- Investigate issues with the QR scanner, when the app is installed on iOS devices
- Add a success response after user is logged in
- Test the layout on very small phones and fix minor styling problems
- Improve bottom bar to be closable by click and to change direction of the arrow icon

Chapter 8

Conclusion

Progressive Web Apps offer enhanced functionality over classic web sites and can stand as a second option next to native apps when it comes to a decision, whichever of them to implement. A current company website or web app can easily be turned into a PWA and can offer at least some of the possible functionality. The main drawback of PWAs is the inconsistency of support among browsers and platforms. Also, some users can open the PWA using an old, unsupported browser, so using progressive enhancement technique is essential. The PWA should not limit its audience by not offering classic web experience to these users.

Pingl web app is an unambiguous example, where PWA makes sense. During this project, a fully functional PWA has been developed. Its functionality covers all the required functional requirements, mainly focusing on the part of the app, where users select a restaurant, where they want to order meals. This PWA can be installed on a variety of devices thanks to its responsive design. The resulting app is intuitive, which was proven by user testing. Furthermore, the app itself is unit tested. It was shown that the current backend and API is suitable to be used with this new PWA. As a result this project is a production-ready part of Pingl.app. To emphasize this, some parts of the functionality, such as notifications, map, location services, and basic caching powered by service workers, have already been implemented in the production Pingl application and is used by real customers.

In terms of the future of this work, the next step is to merge this project into production Pingl.app and replace current insufficient restaurant selection flow. Thereafter, the resulting app should be more user-tested, improved and released to all customers.

Bibliography

[1] Alex Russell. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/. (posted: June 15, 2015, accessed Nov 9, 2019).

[2] Apple Inc. *App Store Review Guidelines*. URL: https://developer.apple.com/app-store/review/guidelines/#minimum-functionality. (accessed: Dec 8, 2019).

[3] StatCounter. *Desktop vs Mobile vs Tablet Market Share Worldwide*. URL: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#monthly-200901-201911. (accessed: Nov 24, 2019).

[4] Tal Ater. *Building Progressive Web Apps: Bringing the Power of Native to the Browser*. O'Reilly Media, 2017. ISBN: 978-1-491-96165-0.

[5] Google Developers Contributors. *Progressive Web Apps Training: Introduction to Service Worker*. URL: https://developers.google.com/web/ilt/pwa/introduction-to-service-worker. (accessed: Dec 7, 2019).

[6] Alex Russell, Jungkee Song, Jake Archibald, and Marijn Kruisselbrink. *W3C Editor's Draft: Service Workers Nightly*. URL: https://w3c.github.io/ServiceWorker/. (accessed: Dec 7, 2019).

[7] MDN Contributors. *MDN web docs: CacheStorage*. URL: https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage. (accessed: Nov 30, 2019).

[8] MDN Contributors. *MDN web docs: Origin*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Origin. (accessed: Nov 30, 2019).

[9] Addy Osmani. *Offline Storage for Progressive Web Apps*. URL: https://medium.com/dev-channel/offline-storage-for-progressive-web-apps-70d52695513c. (accessed: Nov 30, 2019).

[10] *Introducing JSON*. URL: https://www.json.org/json-en.html. (accessed: Dec 6, 2019).

[11] Marcos Caceres, Kenneth Rohde Christiansen, Mounir Lamouri, Anssi Kostiainen, Matt Giuca, and Aaron Gustafson. *W3C Working Draft: Web App Manifest*. URL: https://www.w3.org/TR/2019/WD-appmanifest-20191204/. (accessed: Nov 31, 2019).

[12] Apple Inc. *Safari Web Content Guide: Configuring Web Applications*. URL: https://developer.apple.com/library/archive/documentation/AppleApplications/Reference/SafariWebContent/ConfiguringWebApplications/ConfiguringWebApplications.html. (accessed: Nov 31, 2019).

[13] Joseph Medley. *Google Web Fundamentals - Web Push Notifications: Timely, Relevant, and Precise*. URL: https://developers.google.com/web/fundamentals/push-notifications. (accessed: Dec 8, 2019).

[14] MDN Contributors. *MDN web docs: Notifications API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API. (accessed: May 2, 2020).

[15] Can I Use Contributors. *Can I Use: Notification*. URL: https://caniuse.com/#search=Notification. (accessed: May 2, 2020).

[16] Apple Inc. *Notification Programming Guide for Websites: Configuring Safari Push Notifications*. URL: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/NotificationProgrammingGuideForWebsites/PushNotifications/PushNotifications.html. (accessed: May 2, 2020).

[17] MDN Contributors. *MDN web docs: Push API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Push_API. (accessed: May 2, 2020).

[18] Ruadhán O'Donoghue. *Web push notifications*. URL: https://mobiforge.com/design-development/web-push-notifications. (accessed: May 2, 2020).

[19] Jake Archibald. *Google Updates - Introducing Background Sync*. URL: https://developers.google.com/web/updates/2015/12/background-sync. (accessed: May 10, 2020).

[20] Can I Use Contributors. *Can I Use: Background Sync*. URL: https://caniuse.com/#search=Background%5C%20sync. (accessed: May 10, 2020).

[21] OWASP Foundation. *OWASP Top Ten 2017 Project*. URL: https://www.owasp.org/index.php/Top_10-2017_Top_10. (accessed: Dec 5, 2019).

[22] MDN Contributors. *MDN web docs: Critical rendering path*. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path. (accessed: Dec 5, 2019).

[23] Can I Use Contributors. *Can I Use: Service Workers*. URL: https://caniuse.com/#feat=serviceworkers. (accessed: Nov 24, 2019).

[24] Inc. Formidable Labs. *Formidable Starbucks, Universally Accessible Ordering for Established and Emerging Markets*. URL: https://formidable.com/work/starbucks-progressive-web-app/. (accessed: Nov 16, 2019).

[25] Uber Technologies Inc. Angus Croll. *Building m.uber: Engineering a High-Performance Web App for the Global Market*. URL: https://eng.uber.com/m-uber/. (accessed: Nov 16, 2019).

[26] Addy Osmani. *A Pinterest Progressive Web App Performance Case Study*. URL: https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154. (accessed: Nov 16, 2019, posted Nov 29, 2017).

[27] Microsoft Inc. *TypeScript Documentation*. URL: https://www.typescriptlang.org. (accessed: Dec 14, 2019).

[28] Lukas Marx. *Is Angular 2+ MVVM?* URL: https://malcoded.com/posts/angular-2-components-and-mvvm/. (posted: Dec 06, 2016, accessed: Dec 21, 2019).

[29] Facebook Inc. *React GIT Repository*. URL: https://github.com/facebook/react. (accessed: Dec 11, 2019).

[30] Facebook Inc. *React Documentation*. URL: https://reactjs.org/docs/. (accessed: Dec 11, 2019).

[31] Robin Wieruch. *The Road to learn React: Your journey to master plain yet pragmatic React.js*. Independently published, 2018. ISBN: 9781720043997.

[32] Facebook Inc. *React Documentation: Hooks API Reference*. URL: https://reactjs.org/docs/hooks-reference.html. (accessed: May 5, 2020).

[33] Google Developers Contributors. *Google Tools for Web Developers: Lighthouse*. URL: https://developers.google.com/web/tools/lighthouse/. (accessed: Dec 3, 2019).

[34] Pingl s.r.o. *Pingl.app company website*. URL: https://poznej.pingl.app/en/. (accessed: Dec 14, 2019).

[35] Jesse James Garrett. *The Elements of User Experience: User-Centered Design for the Web and Beyond*. New Riders, 2010. ISBN: 0321683684.

Appendix A

Content of the enclosed CD:

```
/
├── project.pdf ..... pdf version of this text
├── text ..... source LaTeX files of this thesis
├── PWA Examples ..... examples of PWA functionality
│   ├── 01_installable_pwa
│   └── 02_app_like_experience
├── wireframes ..... images of the wireframes
│   ├── 01_landing_desktop.png
│   ├── 01_landing.png
│   ├── 02_scanner.png
│   ├── 03_explore_nearby.png
│   └── 04_map.png
└── implementation ..... source code of the React project
```