PROGRESSIVE WEB APPS

BASICS OF PWA

KRYLAN FOR WEB DEVELOPERS

Basics of PWA

Author: Krylan

Visit the author's website at https://krylan.ovh/portfolio

First Edition (September 2019)

Table of contents

Preface

Krylan here! I'm a web developer, who started adventure with making websites more than ten years ago. Through this time I learned a lot, and worked mainly on my own projects (you can find them in my portfolio). Simultaneously, I'm also someone who love to share own knowledge and help others in learning new things.

Circumstances leading to creation of this book were following: I had to write Bachelor Thesis on my IT-economic studies (hence the language). I was interested in the area of Progressive Web Apps, but I didn't have complete knowledge on this topic. It was a great opportunity to learn more about that through writing and prepare comprehensive guide for other developers at the same time. Killing two birds with one stone.

I'm aware that at the time of publishing it, some parts of this book are outdated already. Web development is changing rapidly and PWA is no different than that. I still believe, that this piece of work can help some developers understand some basics of PWA, or organize the current knowledge. In any case, in the end it will become "History of Basics of PWA" eventually.

I plan to re-write some parts of this book in the future, to adapt it to more actual situation and change the language (to more informal and easier to read). As this book is distributed only digitally, if you find any critical (or not) mistake in this ebook, let me know. I will correct it and re-release. I really care about quality of my works.

Thanks for reading!

Krylan

Introduction

Internet is constantly and rapidly developing, providing many more possibilities for developers and users. As many everyday activities on personal computers and mobile phones were moved to the web, the demand for web applications has risen. Community contributing to the web standards thought about creating a common framework to use native app functionalities and design, while still basing on web languages. Term “Progressive Web App” started to be used to describe various features of web apps, which enhance user experience and make it similar to native applications.

The aim of this work is to introduce the reader to the topic of Progressive Web App and elaborate on techniques and technologies used to create fully functional web apps. Due to the novelty of this concept, the work is simultaneously trying to fill the gap in literature resources.

Topics of this work are divided into 5 chapters:

- Chapter 1 introduces the term of Progressive Web App, indicates differences between this type of application and other approaches, native and hybrid. It also gives an overview of specific requirements to create a Progressive Web App and how browsers and operating systems support them. Lastly, it provides an overview of application stores, where a developer can publish ready PWA,
- Chapter 2 covers specification of Web App Manifest file, needed to define an application, and draws attention at design matter, which differs from the normal web page design approach,
- Chapter 3 describes Web Workers and looks into technical details of creating a simple cache policy for an application,
- Chapter 4 presents possibilities of storing data for offline purposes, in key-value storages and NoSQL database built in the browser,
- Chapter 5 informs about popular tools, which are often used in creating and inspecting Progressive Web Apps;

Way of writing this work is adapted to people with no knowledge about modern web technologies, but also web developers, with some basic introduction to tools and concepts used in creating such. Research methods consist of examination of standards' specifications and statistical information about browser support for specific features, but also practical use of described code and tools on the author's projects. The author of this work is a web developer with several years of experience in this profession and was personally interested in the term "Progressive Web App" and related concepts. He created few web apps, which uses features of Progressive Web App and are highly noted in audits. He intends to expand his knowledge in this direction and follow new features available in the future. Moreover, the author wants to share his knowledge through various means of communication (for example, through an Internet blog or presentations at industry meetings) to increase awareness of this concept among other web developers.

The author made every effort to present the topic reliably.

1. Progressive Web Apps

1.1. Definition of Progressive Web App

One of the first appearances of the term “Progressive Web Apps” (in short PWA) can be found in Alex Russell’s blog post titled “Progressive Web Apps: Escaping Tabs Without Losing Our Soul”, published on June 15, 2015. Alex, who is Google’s Chrome developer, along with Frances Berriman, head of product in Netlify, came up with the name “Progressive Web Apps” for calling the already existing form of installable web applications, which are using operating system’s native functions.

The author of the post described them briefly: “*These apps aren’t packaged and deployed through stores, they’re just websites that took all the right vitamins.*”[1]. If a website meets certain requirements, the user can install the application through his web browser. In case of bigger engagement with a website (for example by frequent visits), the user can be prompted with a suggestion to add an application to his home screen. The biggest strength of PWA is that it can share the same codebase with the website, and it's not dependent on the platform. It would seem, that web application would need an Internet connection to be opened and working, but fortunately thanks to new technologies, developers can create an offline experience for users using a script called Service Worker. This work will look closer into this element of PWA in another chapter.

Year after the name debut, Alex published on his blog another post, titled “What, Exactly, Makes Something A Progressive Web App?”, where he focused more on the mentioned requirements for creating Progressive Web App. This concept is relatively new, so developers should remember, that these criteria are still changing, and new elements are required to create an installable app. One of the basic

[1] Russell Alex, “Progressive Web Apps: Escaping Tabs Without Losing Our Soul”, https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/ posted on 15 June 2015 (accessed on 18 December 2018)

elements required for PWA is serving website content over TLS (Transport Layer Security), with no active mixed content (if it's done right, the browser shows a padlock in the address bar and address itself contains "https" on the beginning). PWA from home screen should be possible to open anytime, even without an Internet connection: if Internet access is crucial for application working, then the user should receive appropriate notice inside the application. To provide that functionality, the website should have a Service Worker, which will cache every essential resource for application on the user's device memory. Registering Service Worker file is needed for creating complete PWA. The last requirement for it is web manifest file, which informs the browser about features of the application. Once these requirements are fulfilled, website visited by the user, when adding to the home screen, instead of adding a shortcut, will install the application on user's device[2].

The next contribution to the subject of PWA's name is Frances Berriman's blog post titled "Naming Progressive Web Apps", posted on June 26, 2017. The author recalls the thinking process behind the name "Progressive Web Apps" and why in the end it's called that. According to her explanation, it's mainly for marketing purposes, because "Web Apps" for company bosses and non-technical workers can be the same as "Native Apps", even if there are a lot of differences between them from the technical point of view[3].

1.2. Differences between native, hybrid and Progressive Web App

The main purpose of making an application is to provide the user with some new functionalities for his device. When planning an application, developers can

[2] Russell Alex, "What, Exactly, Makes Something A Progressive Web App?", https://infrequently.org/2016/09/what-exactly-makes-something-a-progressive-web-app/ posted on 12 September 2016 (accessed on 25 December 2018)
[3] Berriman Frances, "Naming Progressive Web Apps", https://fberriman.com/2017/06/26/naming-progressive-web-apps/ posted on 26 June 2017 (accessed on 26 December 2018)

choose from a wide range of languages and frameworks to adapt their program for a specific platform or operating system. Web languages were from the beginning focused on providing website development. Despite this, developers and companies repeatedly tried to bring them into the market of application development. The reason is that the Internet plays a bigger and bigger role in user's lives. People are using data stored in the cloud, making their work on websites through browser or playing online games. Web languages provide an easy way to create a communication client-server, which can be accessed only with an Internet browser. Companies like Adobe, Microsoft or Google were trying to make use of web languages to create installable applications working outside the browser (but often on its engine), but these ideas didn't become standard and in some cases, they were abandoned.

Before starting to develop an application, developers should think of tasks, budget, and target of the application, to choose the right development approach: native, hybrid or Progressive Web App. They are different and there is no "best" approach because its advantages depend from the factors mentioned above and should be considered individually.

A native application is the first development approach known for a long time. This type of application is written for a specific platform or device and, thanks to that, they can interact with operating system elements or have access to information about the device. It can be written using various programming languages, like C++, Java, Python. First ever created application for computers we can consider as "native". It's a popular and most used approach from these three mentioned, but in some situations, it's not a suitable one. It's the best way for companies, whose application needs strict communication with the operating system and foundations of their business are based on that application.

Developers had also a possibility to create native applications by converting their web files onto, for example, Java application. This type of application, called "Hybrid apps", is working in a simple way: it's compiled application, which contains frame (called WebView) displaying a website, written in web languages. One of the most popular tools for creating hybrid apps is Apache Cordova: it's mobile development framework, which wraps up HTML (Hypertext Markup Language), CSS (Cascading Style Sheets) and Javascript code into a native app for Android, iOS or

Windows. Thanks to hybrid apps, developers could use the same code for creating web application, as well as for native application for mobile devices or desktops. Hybrid app frameworks create a bridge between web apps and native apps, because, even if written in web languages, they can still use some potential of the operating system and be installable.

The last one and most recent approach is Progressive Web App development. This type of application comes to a user directly from a developers' website, likely many native apps, but these can also be distributed through aggregating platforms and application stores. Progressive Web App approach gains more and more popularity due to growing OS/Browser support as well as growing capabilities of this type of application. The biggest advantage of PWA is that developers don't need to work with languages other than web languages. Furthermore, they don't need to work with intermediary software/frameworks as well. The code can be common for the website (accessible from a browser) and Progressive Web App, installed on the user's device. In many cases, this type of application can significantly cut the costs of developing an app for the company. Despite this, PWA still can't do everything that native app could: that's why this approach is not the best in every case.

Capabilities of PWA are still changing along with browser and operating system support. Applications using web languages can use the device's camera or microphone, use geolocation, detect device motion and much more. Today's web browser capabilities can be checked using information from this website: https://whatwebcando.today/

1.3. Use of Progressive Web Apps in today's Internet

Starting from 2015, term PWA started to grow in minds of developers and browsers started to expand their support for PWA's features. It's hard to present its support in a simple way, as this type of application consists of many elements, and one browser can react for one of them, and ignore the other. Considering browser

support for PWA, developers should look at individual parts. Of course, this concept is very fresh, so things are changing from month to month.

Web App Manifest is a file with all fundamental information about the web application. A first public draft of this element's specification was published by W3C (World Wide Web Consortium) on December 17, 2013. In less than a year, the Chrome browser was first to support this idea. Following its steps, most mobile browsers for Android and iOS started gradually to support this feature.

Service Worker is a type of Web Worker which is needed to create a better offline user experience for PWA along with more features. While Web Workers, in general, are supported by more than 90% of browsers' usage, Service Workers has a few percent less (86%).[4] The reason is that Internet Explorer didn't get an update in that matter, as well as older versions of the biggest browsers, which are used today, don't support this element.

Looking at support graphs developers should think also about the future of the market. For example, Internet Explorer is still losing its market share, Microsoft is not developing Windows Phone anymore, so IE mobile will also lose. Moreover, Microsoft has also plan to abandon its newest browser, Edge, and create a new one, based on Chromium. That means, that things like web app manifest or Service Workers should work in the upcoming browser from Microsoft.

At the time of writing this work, new features related to PWA in Google Chrome browser were presented during the "Google I/O '19" event. In new versions coming to be released in 2019, Google Developers plans to give the possibility to access the Native File System, have unlimited quota (which actual limits are described later in this work), and many others. The consciousness of PWA existence among users can also increase thanks to the placement of a special install icon in Google Chrome Omnibox.[5]

[4] Can I use, "Service Workers"
https://caniuse.com/#feat=serviceworkers
(accessed on 31 March 2019)
[5] Google Chrome Developers, "Going Big: PWAs Come to Desktop and Chrome OS (Google I/O '19)"
https://www.youtube.com/watch?v=2KhRmFHLuhE
posted on 9 May 2019 (accessed on 12 May 2019)

1.4. Progressive Web App in stores

Well prepared Progressive Web App, which is fulfilling all requirements to be installable, is possible to install by users, when they open menu in their browser, or when they click on popup window on mobile, after some interaction with the website. It still requires the user to go directly on a specific domain to get the application. In many situations, people are searching for applications through app stores, like Google Play or Apple App Store. They are not searching for application branded by some company: they are searching by function application can perform or according to the topic it is associated with. In this situation, developers of PWA would not be noticed, since they are not listed in those stores, which have huge userbases.

It was clear, that developers and owners of app stores wanted to publish PWA on the common marketplace: developers because they would get a bigger audience, and stores' owners, for an increasing number of applications available in their stores. That's why leading giants of the mobile market did their best to create a possibility to publish PWA on their app stores. These possibilities exist, but they require some work and costs from developers.

Google Play store is the biggest distribution channel for apps on Android. To publish PWA there, the developer needs to use Android Studio software (available for free), in order to create an application with Trusted Web Activity. TWA gives the possibility to put together native code with web code and, using Digital Asset Links, it assures, that the website opened in the application, and the application itself are owned by the same person. This method could be compared to hybrid app approach, mentioned earlier, which opens a website inside a frame (WebView). The difference here is that application with TWA uses Google Chrome browser to display content of the web app. Thanks to that it gives numerous features: form autofill, web push notifications or background sync. Developer should also keep in mind, that publishing on Google Play store requires one-time fee, which is $25 per developer account, which could discourage some developers, who offers free services within their

applications. More information about TWA with link to guide of building such app can be found in the post on Chromium Blog:
https://blog.chromium.org/2019/02/introducing-trusted-web-activity-for.html

Apple App Store is an app store for iOS devices. It's worth noting, that Apple OS for mobile phones doesn't support all features of PWA: at this moment, for example, Background Sync API is not supported by Safari browser[6]. Like with Android, additional software is needed to create an application: Xcode is freeware integrated development environment (IDE) for macOS, but because of constraints of the platform, many developers could find this as an obstacle. Moreover, costs of publishing can be also unaffordable for some developers, because the fee for developer account on Apple App Store is $99 per year.

The third popular app store is the Microsoft Store. Although Microsoft abandoned mobile market with their OS, it still distributes applications through their store for it's most popular desktop operating system, Windows. Publishing in their store, unlike in the two previous ones, doesn't require any fee for a developer account (it can be related to the lack of applications in Microsoft Store, comparing to the other stores). To deploy PWA in Microsoft Store, the developer needs to create *.appx package. It can be done with additional software, Visual Studio, but there is also an alternative that doesn't involve any programs, except Internet browser. On this page of Microsoft documentation can be found a guide on how to create *.appx package and publish it in Microsoft Store:
https://docs.microsoft.com/en-us/microsoft-edge/progressive-web-apps/microsoft-store

As just mentioned, there is an alternative in creating *.appx package, which is helping also with publishing application to other stores. "PWA Builder" is a service by Microsoft, which, after providing URL to the website containing PWA, checks the correctness of app configuration and provides generated packages for different platforms ready to download. Unfortunately, while generated *.appx package for Microsoft Store can be used directly for publishing to the store, Android and Apple

[6] Can I use, "Background Sync API"
https://caniuse.com/#feat=background-sync
(accessed on 24 May 2019)

systems' packages still need additional software mentioned earlier. The tool can be found here:
https://www.pwabuilder.com/

Listed app stores are the biggest because they are owned by creators of platforms, on which these applications are addressed. But there are alternatives, providing "unofficial" marketplaces for apps. As PWA is getting more popular, thus there are special web app stores, created by enthusiasts of this type of application. One of the most prominent examples of such store is Appscope, providing service for Progressive Web Apps. Figure 1 shows the homepage view of Appscope.

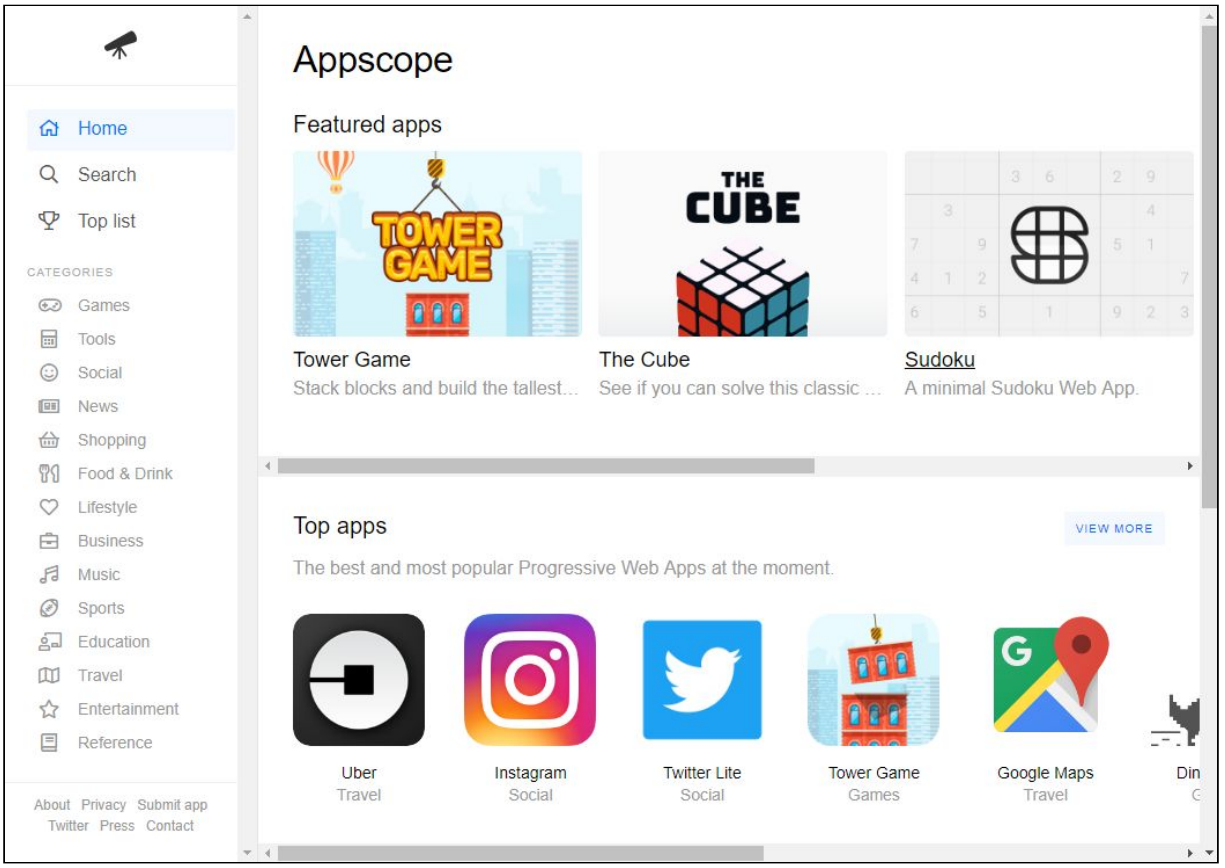

Fig. 1, Appscope homepage view,
Source: https://appsco.pe/
(accessed on 22 May 2019)

Appscope is a new app store, which is cataloging PWAs. It stores information about the app, along with screenshots presenting its design and functions, but the app itself is located on its domain: Appscope only links to it, so the decision on installing the app takes place after the user visits the developer's website.

Adding own app to the directory is simple: Appscope provides a form for submission, where the app can be added even without a developer account. The app can be submitted anonymously, by providing URL of a website with PWA. The system will check if the app is configured properly and meets the standards, using Lighthouse tool (the tool itself will be described in chapter 5.2. of this work) and then it puts the application in the directory. Next chapter will spotlight important file of every PWA, which is providing crucial information about the application, inter alia for Appscope.

2. Web App Manifest and application design

2.1. Web App Manifest

Web App Manifest is a simple JSON file, which contains all crucial information about the Progressive Web App available on the website. It's one of the basic things to create installable PWA. As website caniuse.com reports, this functionality is supported mainly by mobile browsers, but also by Chrome for PC, having 77% of global support in total.[7] Nevertheless, its aim was initially targeted into mobile users, who, instead of downloading simple applications from the app store (like Google Play or Apple's App Store), can just add a shortcut to their home screen, which after opening will behave like a native mobile application.

To make Web App Manifest, the developer should create a new file. While W3C's specification tells that its extension should be *.webmanifest, browsers support also *.json extension, so developers can use any of those two. However, in the same specification can be found a record of MIME media type intended for this file, and it's "application/manifest+json".[8]

Let's take a look at simple web manifest file content presented by W3C:

```
{
  "name": "Donate App",
  "description": "This app helps you donate to worthy
causes.",
  "icons": [{
    "src": "images/icon.png",
    "sizes": "192x192"
```

[7] Can I use, "Web App Manifest"
https://caniuse.com/#feat=web-app-manifest
(accessed on 18 February 2019)
[8] W3C, "Web App Manifest"
https://www.w3.org/TR/appmanifest/
(accessed on 12 February 2019)

```
  }]
}
```

This web manifest contains only the most basic information about web application: its name, description, and icon. Name of the application can be provided in two variants, one as “name” and second as “short_name”, but it’s recommended to use both of them as browsers can display them depending on the place (for example on home screen below the icon would be better to show short name of application). In the case of missing names, the browser can retrieve the name from other elements or display default text when nothing can be used. The “description” should reflect the purpose of the application in short words. “Icons” attribute is an array of objects, where each of them is representing the icon of different size or format so the browser can use these, which are the most suitable for a particular device or system. Developers can also provide “purpose” and “platform” attributes for each image to specify their targeted device group or place, for which it was designed.

Other crucial attributes, which can be used by developers to customize their web app, are “display” and “orientation”. The first one is accepting four values: "fullscreen", "standalone", "minimal-ui", "browser". Each of them is a different level of User Interface (UI) presence (for example “browser” value will open the application in the web browser with the address bar and other functionalities, like a normal website). Another attribute, “orientation”, gives developers the possibility to lock application orientation on mobile, for example for portrait or landscape mode.

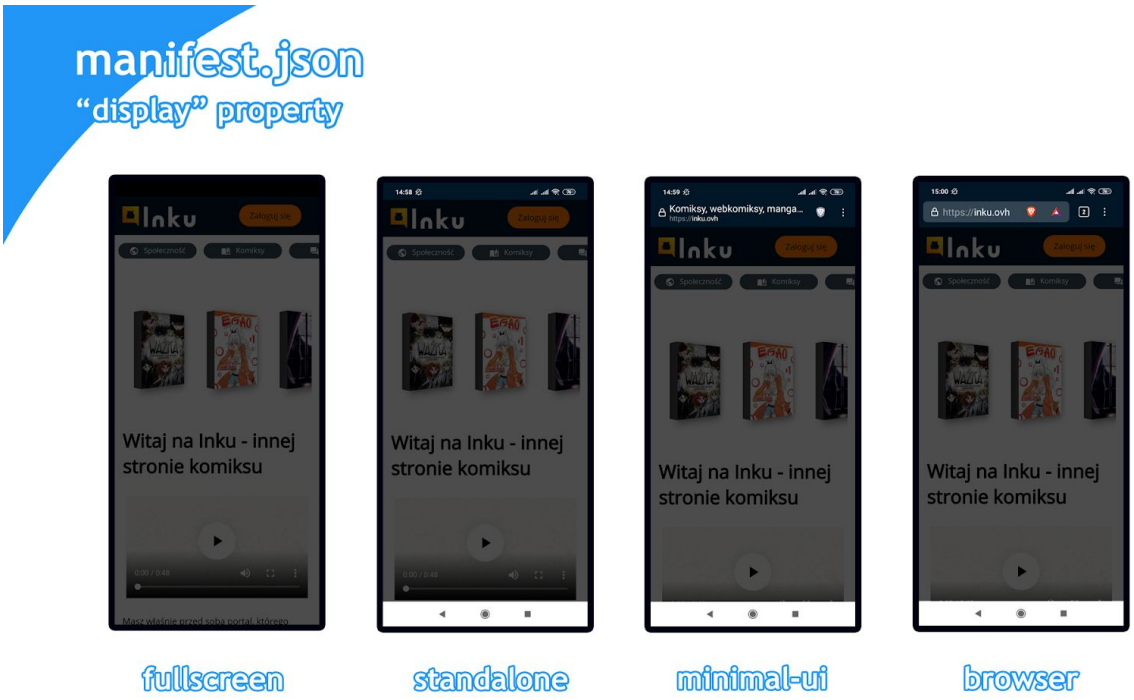

Fig. 2, Display property,

Source: own elaboration

The specification also foresees place for PWA hitting storefronts, giving developers attributes to enhance their app description. While listed above elements, like name, description or icons, can be also used for creating an application page in the application store, there are attributes targeted especially in this scenario. With “screenshots” attribute developers can provide multiple images presenting the actual design and functioning of the application. Attribute “categories” should contain a string with listed categories. These could be used by app stores to catalog apps and simplify user search experiences. Categories in this string are case insensitive, but specification and community encourage developers to use lower-case category names. Also, there is no specified list of categories, which can be used in this attribute, so it’s not limiting developers (but app store could have their strict list of categories). Nevertheless, there is a list of categories, which are commonly used in web app manifest of Progressive Web App:
https://github.com/w3c/manifest/wiki/Categories

Developers can also find attributes, which lets them customize their web app. With “theme_color” developers can set a default color scheme for their application. However, this one element can be used also outside the app manifest, in website HTML code. Meta tag “theme-color” in the head of HTML document is giving the same purpose, but, according to specification, its value can be overridden if a color value is provided by app manifest.

Attribute “dir” is defining reading direction for the most important manifest elements: “description”, “name”, “short_name”. It takes one of the three values: “ltr” (left-to-right text), “rtl” (right-to-left text) and “auto”. There is also the “lang” attribute, which is responsible for defining web app manifest’s (not application itself), above mentioned, elements language. Its value needs to be appropriate to IANA Language Subtag Registry. It can contain language as a two-character shortcut, but also (separated by “-”) country, to specify pronunciation.

There could be a possibility, that the developer wants the application to start from a certain address, which could be different from the home webpage. Using the “start_url” developer can define URL, from which application will start, after being opened by the user. W3C specification beware, that browsers can ignore this value or let the user decide, from which URL he wants to start the application.

2.2. Application design and shell architecture

Before the developer starts to create an application, it needs to be designed first. Under the hood of the Progressive Web App, there are languages and techniques used in website development, but from the design point of view, it should more resemble application, than a website. It means, that some elements of navigation, user interface, should be tailored to an application interface.

One of the most prominent examples of such design would be presenting a back button on derivative pages from the homepage. While a PWA doesn't open in a browser window, users don't have a navigation bar and option to control browse history. That's why such button should be taken into account when designing a web application. As mentioned earlier, PWA should be more Internet connection independent, that's why, if it is necessary, the application should inform the user with an appropriate and clear statement.

To help create a consistent and native-like application, developers can use existing frameworks, which have prepared CSS and JavaScript files to resemble the Material Design interface, which is commonly used for native Android applications. One of the known examples of such framework would be Material-UI[9]. Google, which is the creator of Material Design, also shares libraries called Material Components, which include simultaneously CSS and JavaScript files with ready elements for creating a web application.[10]

Having this topic around, it's worth to introduce the "Application shell architecture" term. It is a concept, which describes crucial HTML, CSS, JS files needed to ensure, that the app will show its structure on opening, even without an Internet connection. It should contain all necessary navigation elements, which are downloaded at the first visit, and then they are retrieved from device's memory on the next opening of the installed application. This can be achieved by using Service

[9] "Material UI"
https://material-ui.com/
(accessed on 18 March 2019)
[10] "Material Components for the web"
https://material.io/develop/web/
(accessed on 18 March 2019)

Workers, which will be described in detail next chapter. Figure 2 shows the difference between application shell and content of the application.

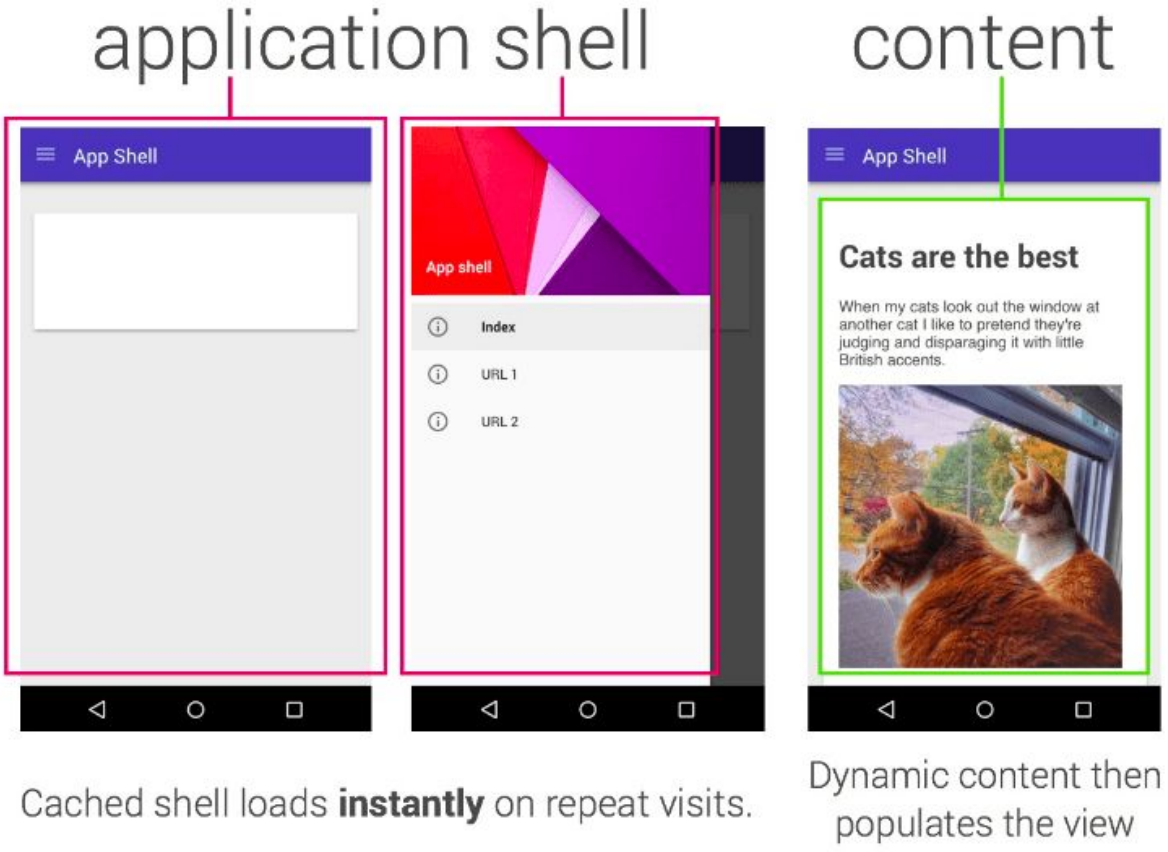

Fig. 3, “The App Shell Model”,
Source: https://developers.google.com/web/fundamentals/architecture/app-shell (accessed on 13 May 2019)

Image files being part of application content (for example photos of blog or news posts) are not considered as a part of application shell.

3. Web Workers and Service Workers

3.1. Definition of Web Workers

Web Worker is a Javascript file, which is running in the background. In other words, all scripts called in <script> HTML tag are executed on the main thread, altogether with the website, sometimes causing a slowdown on delivering content to the user. But Web Worker is a script called to be executed in parallel, so it's not interrupting the works of the website, which is especially profitable if the script contains code using a lot of computing power.

A developer can easily register a worker in Javascript, by pointing to a file of the script on a webpage in the following way:

```
var worker = new Worker('worker.js');
```

The provided script file will be executed on another thread, in the background. It's worth noting, that this script doesn't have access to the DOM tree on the website, so it can't directly change HTML elements. However, a website can communicate both ways with worker script, exchanging information, for example, the results of computing.

As mentioned earlier in another chapter, Service Worker is considered to be the type of web worker, to be more specific, a shared worker. Its most known feature is fetching website resources to cache and delivering them on the next user's visit, or when he will lose Internet connection. It has also different interesting features, like push notifications, which can be delivered even when the user doesn't have application opened at the time.

Each Service Worker has its lifecycle and its simplified schema is presented on Google's Developer Guide webpage, which is presented in figure 3 of this work.

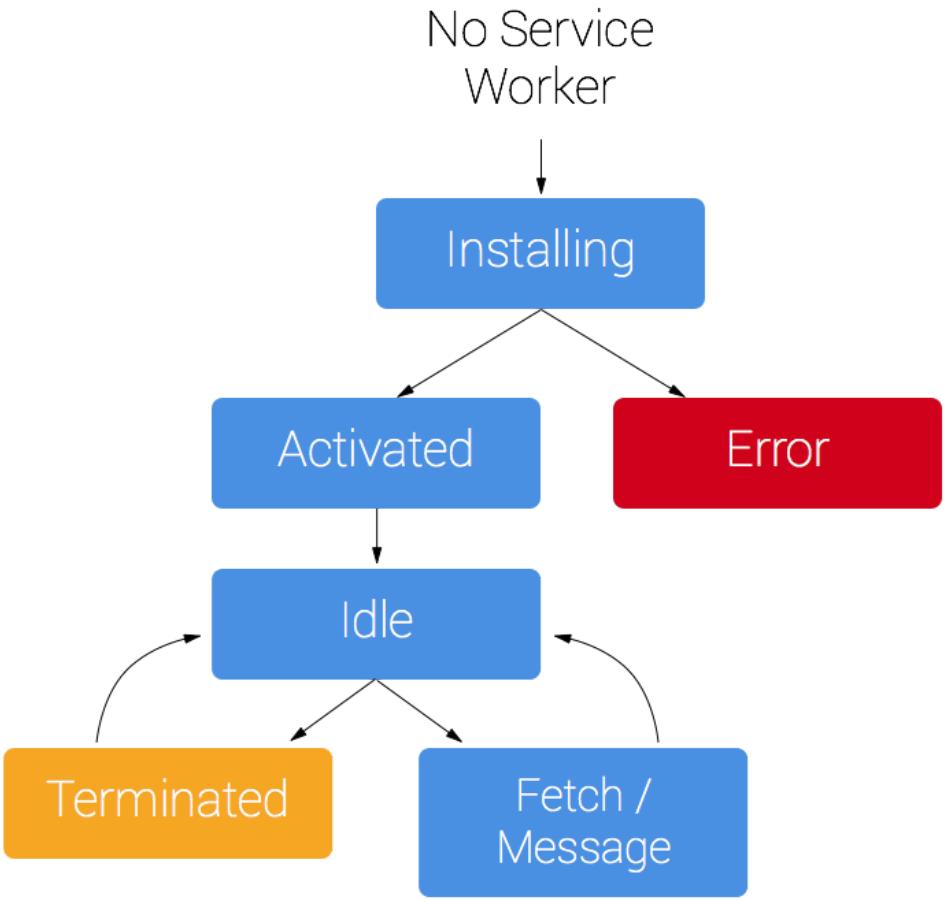

Fig. 4, “Service Worker lifecycle”,
Source: https://developers.google.com/web/fundamentals/primers/service-workers/ (accessed on 06 April 2019)

Further this chapter the work will present a simple practical cache setting for PWA through Service Worker with an explanation of each step.

3.2. Demonstration of Service Worker in JavaScript

The very first part of implementing a Service Worker in Web Application is creating an empty script, which will contain all instructions to do “in the background”. In this example, this file will be named “sw.js” and put in the main folder of the

application. Next, in the bootstrap file of the application such line is required to register Service Worker:

```
navigator.serviceWorker.register('/sw.js');
```

Function above returns promise, which can be expanded on success/error handling with “then()”/”catch()”. Before executing this code it’s also worth to check if user’s browser is supporting Service Workers, by checking if “ServiceWorker” exists in the “navigator” object. On the first user visit, the browser will register a given Service Worker and try to install it. After the installation process, if it won’t fail, the browser will activate SW and let it function in the background. In this example, work will present a way for creating offline cache for an application (and some of those files can be an “App Shell”, which was mentioned earlier).

In “sw.js” file, on the first line there will be a declaration of cache name, which will be used by Service Worker during caching operation and fetch event:

```
const cacheName = 'cache';
```

That’s how typical listener on “install” event could look like:

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          'style.css',
          'script.js',
          'offline.html',
        ]
      );
```

```
    })
  );
});
```

It should be noticed, that Service Worker is referenced in code through the “self” keyword.

The code above is opening a cache with the name delivered earlier and adding files listed in the array. In the case of adding only one file to the cache, a developer can use the function “add” with a single file path as an argument, but in most cases, there would be need to save more files. What is worth noting, when using “addAll”, every file needs to be successfully cached: otherwise, Service Worker will fail to install.

The characteristic thing about Service Worker is that it’s behaving like a “good” middleman between browser and server. When the user wants to visit the website, the browser is sending requests to the server to get the necessary files. With Service Worker, these requests go through it and SW can decide to respond to this specific request with another file, for example, this cached one, on behalf of the server. This operation can be handled with “fetch” event, where file names from cache are compared to these from requests:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);
    })
  );
});
```

In this listener, “cacheName” was used again, because SW should use files from the same place there were stored in the previous listener.

The whole code is responsible for storing into cache HTML, CSS and JS files and, on another visit or refresh, retrieve these files when a browser will request them. This helps in both situations: when the user has an Internet connection, it will use fewer Internet data to load the website, thus it will be faster, but also without such a connection browser will be able to load app resources from cache. However, there are also different approaches for fetching files from the cache. It's because in some situations developer would like to use cached resources when a user doesn't have an Internet connection, but simultaneously deliver actual files from the server, when such connection is established ("Network falling back to the cache").

Object "caches", which was used in examples above, is accessible not only from Service Worker. Caching techniques can be also used directly on website/application in Javascript code. It can be really useful in creating "save to read later" buttons on the website, which will save the article with resources on the user's device so he could read it anytime, regardless of Internet connection state. Caching can be also done while fetching resources: every time resource is going from the server through SW, it can copy the file to opened application cache, to retrieve it on the next visit. Google's Web Developer Guide is covering the topic of different cache techniques and how each of them should be written.[11]

[11] Google, "Caching Files with Service Worker"
https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker
(accessed on 27 April 2019)

4. Storing data in Progressive Web App

4.1. Web Storage

There are times when a web developer needs to store some data on users' devices to provide better services. One of the common conceptions for that purpose are HTTP cookies. Cookies are files, which store small amounts of data and can be used, for example, to authenticate user visiting the website (because the web server doesn't distinguish users who request files). Cookies from the domain are sent to the web server with every request, even for images or other non-server interpreted resources. It can often create unnecessary network traffic because cookies don't have to be used when considering requested resources. There is even a workaround for this situation: static resources like images or CSS files can be served from another domain (or subdomain) where the user doesn't have any cookies registered.

In the time, when websites started to be something more than client-server communication, there was an obvious need to store the data only for client purposes, but with better features. Cookies were designed to store very small (around 4 KB) amount of data needed only for some time (cookies have an expiration date) to deliver them to the server. Expanding usage of web development languages for applications and games created need for storing bigger data, used by users every time when they open the application (for example save settings of an application or save game progress). Cookies couldn't fulfill these requirements, especially, that in case of online-offline applications all of them would be sent to the server with every request, causing big and unnecessary network traffic.

Response to that needs was Web Storage, which started to be supported in first versions of Firefox browser in 2006, now exceeding 93% of global browser support[12]. This feature consists of two elements: LocalStorage and SessionStorage, which are similar in their way of working.

[12] Can I use, "Web Storage - name/value pairs"
https://caniuse.com/#feat=namevalue-storage
(accessed on 19 January 2019)

Local Storage is based on key-value data storage paradigm: each element has only a unique identifier (called key or index) and a value, which in case of Web Storage occurs in the form of strings. The quota limit for stored data is much bigger than in cookies, but it's not specified by the W3C specification and it's dependent on specific browsers (the most common capacities are 2.5 MB and 5 MB per domain)[13]. Data stored in Local Storage doesn't have an expiration date and it's also device persistent, which means, that even after closing the browser, data will be kept and possible to use in another session. Local Storage data is accessible through javascript, with functions for getting, setting, and removing particular values from the storage.

```
localStorage.setItem('myCat', 'Tom');
```

Using function "setItem" on object "localStorage", developers can set a pair of key/value, giving strings (in case of other value type it will be converted into a string) into two corresponding arguments.

```
var cat = localStorage.getItem('myCat');
```

Passing a pair's key in function "getItem" will return the value of it. In the case of non-existing pair, null will be returned.

```
localStorage.removeItem('myCat');
```

In "removeItem" function developer can remove key/value pair by giving its key.

[13] W3C, "Web Storage (Second Edition)"
https://www.w3.org/TR/webstorage/
posted on 19 April 2016 (accessed 19 January 2019)

```
localStorage.clear();
```

Local Storage “clear” function will remove all this type stored data for this domain.

Developers can set and get items also in another way, treating localStorage as usual Javascript array, but it’s not recommended due to problems which can happen when plain objects are used as a key-value store (for example functions can be accidentally overridden).

Session Storage works on the same principles and shares the same features with Local Storage: the only difference is that data stored in Session Storage is only session persistent, which means, that all the data will expire after browser window/tab will be closed. The purpose of this kind of storage is to manage bigger data needed only temporarily in a particular user visit.

All shown above functions are applicable also to this type of storage, but it should be invoked on the “sessionStorage” object instead.

4.2. Indexed Database (IDB)

In some cases, Web Storage, described above, wouldn’t be enough. It also has much-limited space and is used to store only simple values in a global array. The demand for a bigger amount of more complex data was still increasing, especially when web apps were starting to be developed in the direction of offline working.

Data on websites very often come from databases. To give user possibility to access this data without a persistent Internet connection, the developer should copy it and save to the client side. In some cases, saving it in Web Storage would be cumbersome and not efficient. The response for that demand was WebSQL, which short lifecycle started in 2009. W3C stopped works on it, making WebSQL deceased

year later. On Working Group Note concerning WebSQL was left warning message, bewaring developers from using this feature and explanation of abandoning it:
"*This document was on the W3C Recommendation track but specification work has stopped. The specification reached an impasse: all interested implementors have used the same SQL backend (Sqlite), but we need multiple independent implementations to proceed along a standardisation path.*"[14]

Indexed Database (IndexedDB or IDB in short) has taken place of unrealized standard. It started to be supported experimentally (with browser-specific prefixes) by Mozilla Firefox and Google Chrome in early 2011. Right now, this storing data method is supported 92% users' browsers[15], although IndexedDB 2.0 (it started to be W3C Recommendation on 30 January 2018), which contains improvements to the standard, has more over 81% of support.[16] At this moment, Indexed Database API 3.0 is in the work as Editor's Draft.

IndexedDB is a NoSQL, object-oriented database. It's a different architecture approach from relational databases, which are table-oriented. Moreover, as a NoSQL database, and contrary to the earlier mentioned WebSQL, it doesn't use Structured Query Language (SQL) for managing data. Everything in IndexedDB is happening inside a transaction, like in relational DBMS. The advantage of this model is preventing concurrent scripts to interrupt each other (for example when the application is opened in two windows), altering data in the same time, as well as providing consistency in case of sudden browser disruption.

Considering storage size there is no constant value of space intended for use IndexedDB. In most cases, it's dynamically calculated basing on free hard disk space. According to Mozilla documentation on this topic, Firefox browser has 50% of hard disk space available for all stored data and it's called a "global limit". From that

[14] W3C, "Web SQL Database"
https://www.w3.org/TR/webdatabase/
posted on 18 November 2010 (accessed 22 January 2019)
[15] Can I use, "IndexedDB"
https://caniuse.com/#feat=indexeddb
(accessed on 22 January 2019)
[16] Can I use, "IndexedDB 2.0"
https://caniuse.com/#feat=indexeddb2
(accessed on 22 January 2019)

value, each group limit (which is composed of the domain and its subdomains) has 20% for use, a minimum 10 MB, but no more than 2 GB of disk space.[17]

[17] MDN Web Docs, “Browser storage limits and eviction criteria”
https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria#Storage_limits
(accessed on 22 January 2019)

5. Tools for building Progressive Web App

5.1. Developer Tools

Google, as one of the biggest Internet companies and owner of the most popular browser, Chrome, is very often involved in developing standards and facilitating the work of web developers. On Chrome Dev Tools, available together with the browser (keyboard shortcut for opening/closing it is on F12 key), developers can find a lot of information and tools helpful in creating complete Progressive Web App.

The basic source of information about “appiness” of the created website will be tab “Application” in dev tools. All described in this work elements can be found there: detected web app manifest, active Service Worker, Local and Session Storages, IndexedDB, as well as Cache Storage and more. In subtab “Clear storage” developers can also find how much space for cache and IDB the application is currently using and how much storage quota is assigned on this specific device, together with the option to clear particular elements of storage. Figure 4 shows how app manifest subtab looks like in the developer tools, on the sample of the author’s project.

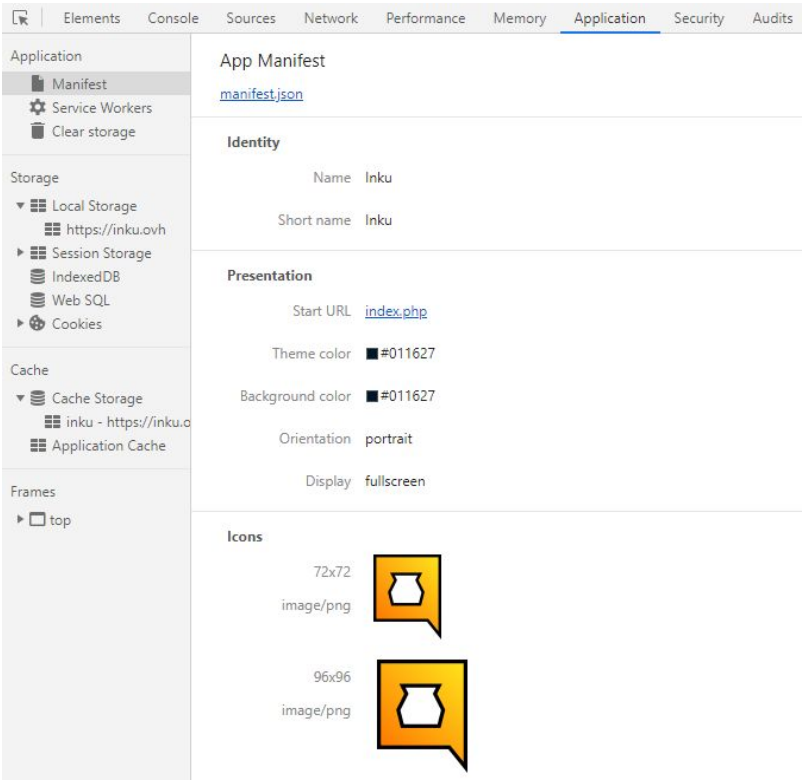

Fig. 5, Sample of Manifest subtab in Chrome Dev Tools,
Source: own screenshot

In subtab “Service Worker” developer can find information about the status of registered SW and tools which let perform certain interactions to test its features. For sure, one of the most useful elements in this subtab would be checkbox “Update on reload”. When checked, every time when the developer will refresh the website with developer tools opened, the Service Worker will be downloaded again, forcing all changes to be in effect with the current load. Checkbox “Offline” will also come in handy when a developer wants to simulate the lack of Internet connection. In this place service workers from all domains can be updated, unregistered or tested in means of push and sync functions. Content of this subtab is presented in figure 5.

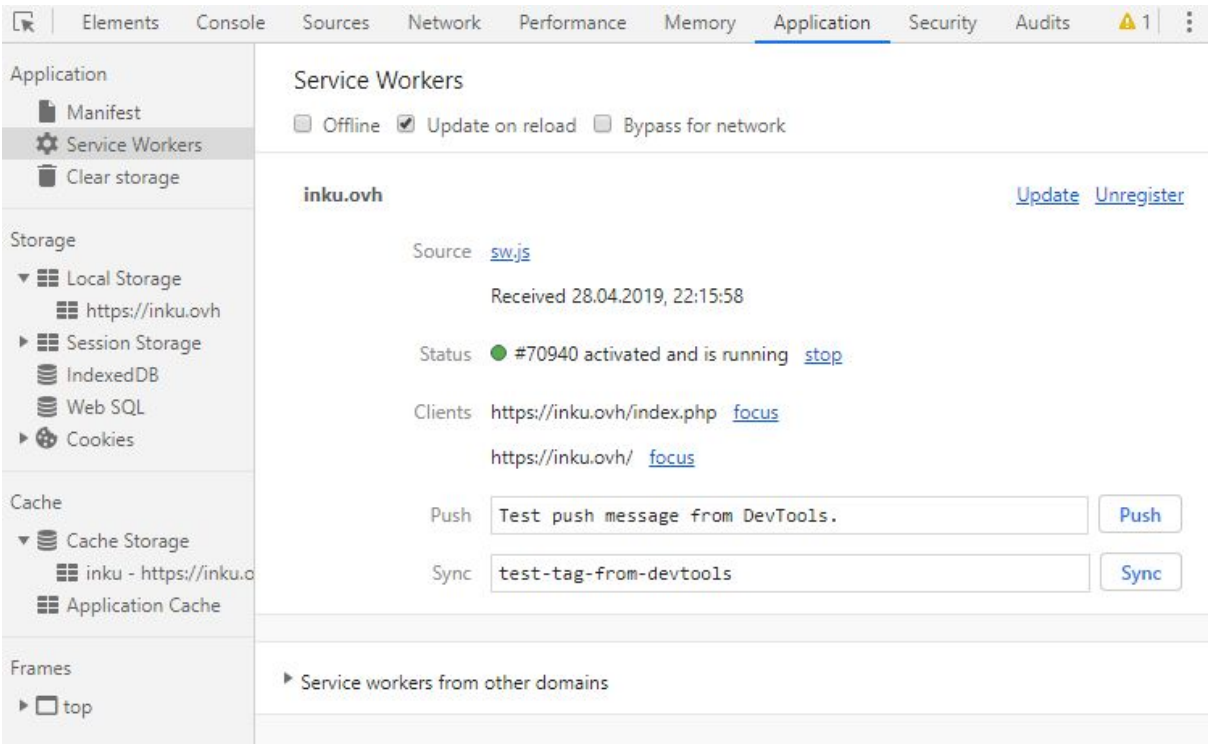

Fig. 6, Sample of Service Workers subtab in Chrome Dev Tools,
Source: own screenshot

Considering key-value storage, like Local Storage, Session Storage, and Cookies, developers should keep in mind, that all values of them can be altered by the user using developer tools in browser and also by executing JavaScript code in the developer console. Therefore, it shouldn't contain any sensitive, or important data for security. The application should also have instructions on what to do in case of unexpected data from storage, in the case when someone would edit it.

The main tab "Network" also contains relevant and useful information about resources, which actual website uses. When recording is set on, below waterfall chart of loaded resources through time there is a table with information about all those files. From this table, developers can not only check the load time of a file (very useful for optimizing the total load time of application) but also read, from what source file was loaded. When it was downloaded from the network, dev tools will

display in “Size” column number with filesize. Otherwise, it will specify the source in brackets, for example “(from disk cache)” or “(from memory cache)”.

5.2. Lighthouse

Web Developers are not ideal in terms of knowledge, which in this domain is vast. It’s easy to forget about some elements on a newly created website and it’s hard to properly measure the performance of the page. Making a list and checking it every time manually would be troublesome and time-consuming, that’s why these processes are automated thanks to different tools, which are analyzing the website and showing results in form of an audit. One of such tools is Lighthouse.

The mentioned tool is created and maintained by Google, which encourages other developers to contribute to it in the form of the open-source repository on Github service.[18] Even though it’s owned by the developer of the most popular web browser, Lighthouse can be run not only from Chromium developer tools but also from NodeJS command line or the website (https://web.dev/measure). The last source is relatively new and inaccurate and, moreover, doesn’t check PWA factors, which from the perspective of this work are the most important. Considering the above, example screens in this sub-chapter will come from developer tools, from the “Audits” tab. A received audit can be exported to JSON format, and then imported to Lighthouse Report Viewer, which is available at Github service:
https://googlechrome.github.io/lighthouse/viewer/

Lighthouse audit gives developer ratings about the single web page from 5 categories: performance, accessibility, best practices, SEO (Search Engine Optimization) and Progressive Web App. Scores are given on a 0-100 point scale, except for PWA, which results are split through three subcategories. Figure 6 shows example results of audit, while figures from 7 to 9 present subcategories with criteria according to which Lighthouse were checking the website.

[18] “Lighthouse”,
https://github.com/GoogleChrome/lighthouse
(accessed on 07 May 2019)

Fig. 7, Sample of audit summary in Chrome Dev Tools,

Source: own screenshot

The first subcategory, “Fast and reliable”, checks if the webpage is adjusted to mobile network connection (which is generally slower) in terms of performance and data amount. It also checks if the webpage responds with a suitable status code (200) when it’s offline. Even if the application is not designed to work offline, it should return a page with an appropriate message to the user when he doesn’t have a persistent Internet connection.

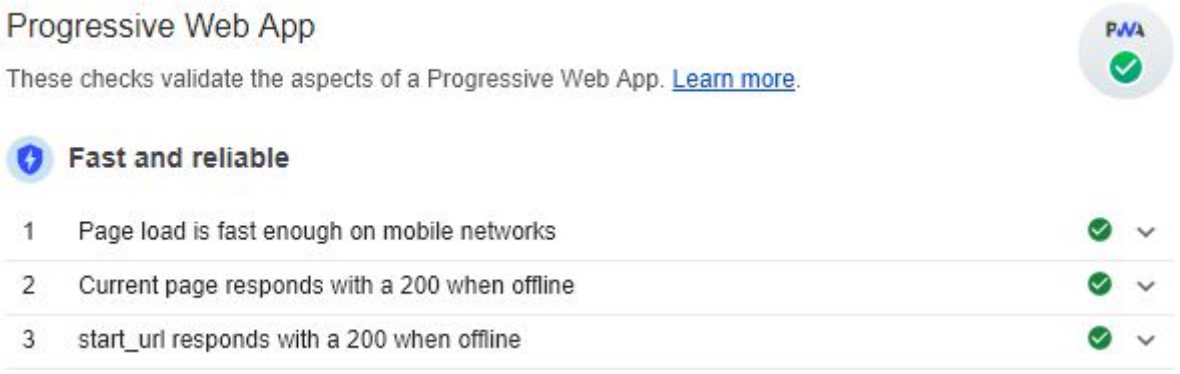

Fig. 8, “Fast and reliable” subcategory in Chrome Dev Tools,

Source: own screenshot

"Installable" is the second subcategory. It checks whether web page fulfills requirements to become an installable app (i.e. it's served from the safe origin with HTTPS and registers a Service Worker).

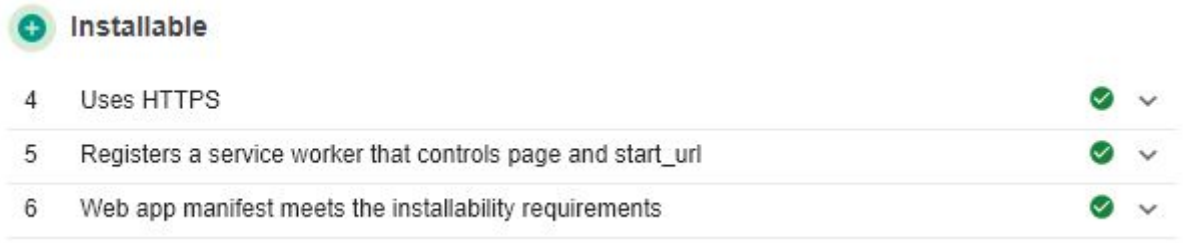

Fig. 9, "Installable" subcategory in Chrome Dev Tools,
Source: own screenshot

The last subcategory, "PWA Optimized", includes checks various additional features of PWA and page responsiveness. There is also another list below, which contains elements for manual checking by the developer because they cannot be examined by Lighthouse.

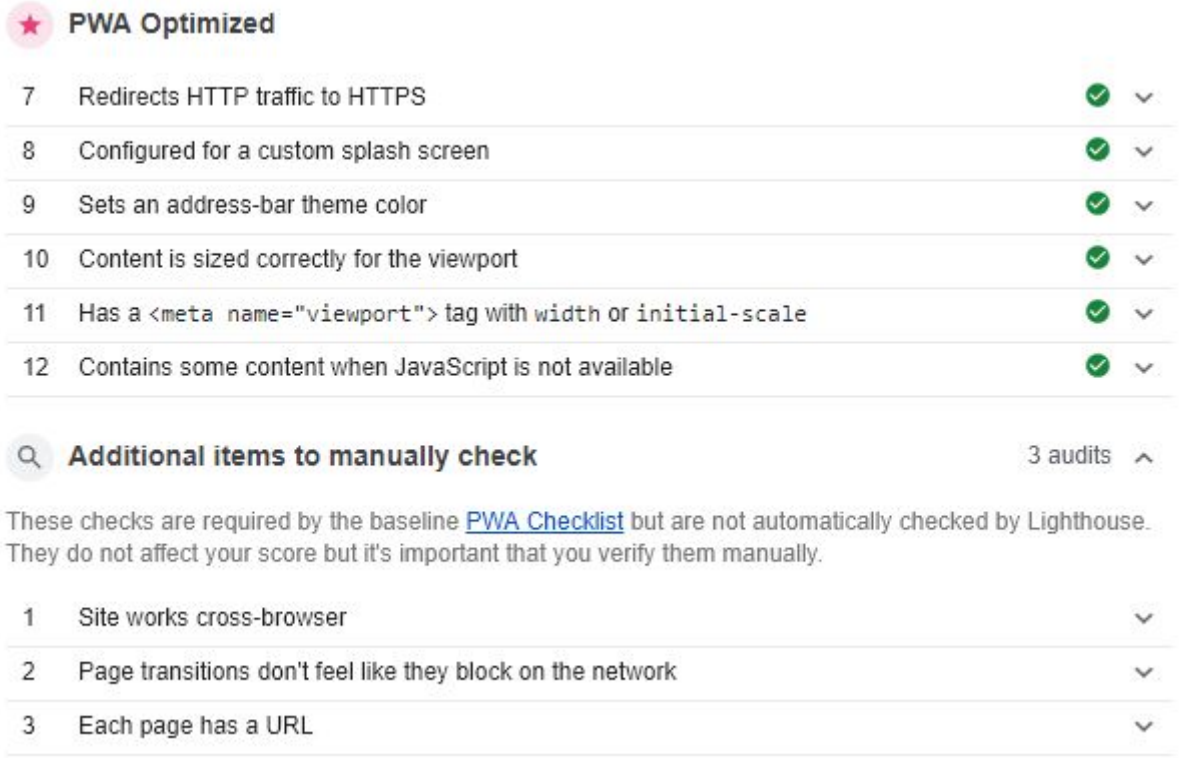

Fig. 10, “PWA Optimized” subcategory and manual checklist in Chrome Dev Tools, Source: own screenshot

It’s good to test web page with this tool, not only in terms of PWA (audit gives information about UI accessibility, details about performance, tips on SEO, etc. which are also important). On the other hand, they don’t contain other things, on which developers should pay attention to when building their website.

5.3. Workbox

Workbox is a product of Google, which helps in building PWA faster, focusing on Service Worker features. It consists of libraries and nodeJS modules, which imported into application make things easier to build.

The first thing in order to use Workbox in the project is importing script in the Service Worker file:

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/4.3.1/workbox-sw.js');
```

The script will create a workbox object, which imports other libraries depending on needs. One of the frequent use cases of Workbox is to prepare and execute a caching strategy for the application. Workbox is using two caches for storing application files: precache and runtime cache. Despite the fact, that they have default names generated by itself, developers can create custom one in the following way (with example names):

```
workbox.core.setCacheNameDetails({
  prefix: 'pwa',
  suffix: 'v1',
  precache: 'precache',
```

```
  runtime: 'runtime'
});
```

With this, on live precache will be named “pwa-precache-v1”, while runtime cache “pwa-runtime-v1”. Precache store is used, when it’s inserting files on the “install” event.

Example code below will recreate cache policy presented in chapter “Web Workers and Service Workers” (named “cache first”):

```
workbox.precaching.precacheAndRoute([
     'style.css',
     'script.js',
     'offline.html',
]);
```

Also, with Workbox developer can prepare “routes” for cached files and apply different cache strategies to them. For example, this code will cache (in runtime store) all JS files encountered through “fetch” event, and then on next visits it will try to download it from the network, but when it fails (for instance user doesn’t have an Internet connection), then it will be loaded from cache:

```
workbox.routing.registerRoute(
  new RegExp('.+\\.js$'),
  new workbox.strategies.NetworkFirst()
);
```

Conclusion

Throughout this work, there were presented different aspects of building Progressive Web App. Individual chapters fully realized the tasks set out in the introduction. As mentioned earlier, this concept is relatively new and it's rapidly developed from various sides: specification (W3C and WHATWG documents), technical support (browser and operating system updates), and community (frameworks, libraries, and other tools helping with the development of PWA). For this reason, it's difficult to present every element of this type of application and follow it up to date.

The subject of PWA has a bright future, as it's supported by the biggest Internet companies and many developers in the community feel enthusiastic about this idea. In the opinion of the author of this work, the biggest challenge at this moment is low level of awareness among Internet users. A similar situation can be observed about RSS: despite the more than 20 years of existence of this standard and its frequent occurrences on websites, it's still not commonly used by the users, even though it would simplify subscribing and getting notified about any new content on their frequently visited websites. In the case of PWA, Google is trying to increase awareness of these applications among users by displaying messages or placing special buttons in important places of user interface. This can help developers in increasing userbase of their web applications and make PWA more common in the next years. Looking at the rate at which web technologies and online services are developing, it can come to the situation, where native applications will become unnecessary and users would be using only web applications.

Developers should be on guard because requirements for making installable PWA can change and some parts of the specification are not finally determined. The author of this work hopes that Progressive Web Applications will be more often used in the future and they will help in building open and modern web, for everyone. The author itself is creating PWA and wants to spread the knowledge about this type of application among other web developers.

List of References

1. Appscope,
https://appsco.pe/
(accessed on 22 May 2019)

2. Bar Adam, “What Web Can Do Today”,
https://whatwebcando.today/
(accessed on 05 January 2019)

3. Berriman Frances, “Naming Progressive Web Apps”,
https://fberriman.com/2017/06/26/naming-progressive-web-apps/
posted on 26 June 2017 (accessed on 26 December 2018)

4. Can I use, “Background Sync API”
https://caniuse.com/#feat=background-sync
(accessed on 24 May 2019)

5. Can I use, “IndexedDB”,
https://caniuse.com/#feat=indexeddb
(accessed on 22 January 2019)

6. Can I use, “IndexedDB 2.0”,
https://caniuse.com/#feat=indexeddb2
(accessed on 22 January 2019)

7. Can I use, “Service Workers”,
https://caniuse.com/#feat=serviceworkers
(accessed on 31 March 2019)

8. Can I use, “Web App Manifest”,
https://caniuse.com/#feat=web-app-manifest
(accessed on 18 February 2019)

9. Can I use, “Web Storage - name/value pairs”,
https://caniuse.com/#feat=namevalue-storage
(accessed on 19 January 2019)

10. Chromium Blog, “Introducing a Trusted Web Activity for Android”,
https://blog.chromium.org/2019/02/introducing-trusted-web-activity-for.html
posted on 5 February 2019 (accessed on 21 May 2019)

11. Gaunt Matt, Google, “Service Workers: an Introduction”,
https://developers.google.com/web/fundamentals/primers/service-workers/
(accessed on 18 May 2019)

12. Google, “Caching Files with Service Worker”,
https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker
(accessed on 27 April 2019)

13.Google Chrome, “Lighthouse”,
https://github.com/GoogleChrome/lighthouse
(accessed on 07 May 2019)

14. Google Chrome, “Lighthouse Report Viewer”,
https://googlechrome.github.io/lighthouse/viewer/
(accessed on 18 May 2019)

15. Google Chrome Developers, “Going Big: PWAs Come to Desktop and Chrome OS (Google I/O ’19)”,
https://www.youtube.com/watch?v=2KhRmFHLuhE
posted on 9 May 2019 (accessed on 12 May 2019)

16. Microsoft, “Progressive Web Apps in the Microsoft Store”,
https://docs.microsoft.com/en-us/microsoft-edge/progressive-web-apps/microsoft-store
posted on 27 June 2018 (accessed on 23 May 2019)

17. Microsoft, “PWA Builder”,
https://www.pwabuilder.com/
(accessed on 23 May 2019)

18. Osmani Addy, Google, “The App Shell Model”,
https://developers.google.com/web/fundamentals/architecture/app-shell
(accessed on 18 May 2019)

19. Russell Alex, “Progressive Web Apps: Escaping Tabs Without Losing Our Soul”,
https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/
posted on 15 June 2015 (accessed on 18 December 2018)

20. Russell Alex, “What, Exactly, Makes Something A Progressive Web App?”,
https://infrequently.org/2016/09/what-exactly-makes-something-a-progressive-web-app/
posted on 12 September 2016 (accessed on 25 December 2018)

21. Web.dev, “Measure”,
https://web.dev/measure
(accessed on 18 May 2019)

22. W3C, “Categories · w3c/manifest Wiki”,
https://github.com/w3c/manifest/wiki/Categories
(accessed on 18 May 2019)

23. W3C, “Web App Manifest”,
https://www.w3.org/TR/appmanifest/
(accessed on 12 February 2019)

24. W3C, “Web Storage (Second Edition)”,
https://www.w3.org/TR/webstorage/
posted on 19 April 2016 (accessed 19 January 2019)

25. W3C, “Web SQL Database”,
https://www.w3.org/TR/webdatabase/
posted on 18 November 2010 (accessed 22 January 2019)

List of Images

PROGRESSIVE WEB APPS

Progressive Web Apps is the term describing web applications, which uses features and design known from native applications for PC and mobile. Thanks to that, developers can use web languages to create a fully functional application, installable by the user on his device.

Krylan is a web developer with several years of experience. This book was initially written as Bachelor Thesis for IT-economic studies, but from the beginning, it was intended to release for other developers in the form of eBook.

Visit author's website
https://krylan.ovh/portfolio/