Technology
Arts Sciences
TH Köln

Progressive Web Apps - Costs, Benefits and Tradeoffs

Masterarbeit

ausgearbeitet von

Yasa Yener

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt an der

Technischen Hochschule Köln
Campus Gummersbach
Fakultät für Informatik und
Ingenieurwissenschaften

im Studiengang

Informatik / Computer Science
Schwerpunkt: Software Engineering

Erster Prüfer: Prof. Dr. Stefan Bente
Technische Hochschule Köln

Zweiter Prüfer: Dr. Christian Willmes
Universität zu Köln

Gummersbach, April 13, 2018

Adressen: Yasa Yener
Homarstraße 24
51107 Köln
yasa.yener@smail.th-koeln.de

Prof. Dr. Stefan Bente
Technische Hochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
stefan.bente@th-koeln.de

Dr. Christian Willmes
Geographisches Institut
Universität zu Köln
Albertus-Magnus-Platz
50923 Köln
c.willmes@uni-koeln.de

Abstract

Many tasks that humans perform in their daily lives are computer-assisted. The actual devices are, however, often very heterogeneous in nature and differ in their architecture (x86/x64, ARM), operating system (Windows, MacOS, Linux, Android, iOS), but also other factors like performance or network availability. Those factors can strongly increase the complexity for software engineers that want to address those platforms.
Progressive web apps try to apply a subset of requirements, that are usually expected from native applications, to the web, by using modern, and in some cases experimental, technologies. This thesis investigates which trade-offs remain and which efforts have to be made in order to fulfill most of these requirements.
It shows that progressive web apps are the most approachable type of application that is currently available, but have noticeable limitations when it comes to accessing most OS-level APIs. They do prove to be a viable substitute for native mobile applications as shown in multiple case studies. The biggest challenges in building progressive web apps seems to resolve around managing user interaction and keeping up with the vast majority of technologies, frameworks and patterns that keep emerging.

Contents

1. Introduction

Humans perform many tasks in their daily lives that are computer-assisted. The underlying computing systems are often heterogeneous in order to fit their niche in their respective context. Developing software for those devices can quickly become a complex task, as software engineers have to address different operating systems and system architectures, while also deal with varying processing performance and network availability. Since each specific platform requires its own body of knowledge, development of such cross-platform software will often require hiring multiple experts and/or teams of their respective domain in order to possibly deliver a consistent experience. This increases development costs as well as making it harder to come to a general consensus, since more parties are involved.
Netscape developers back in 1999 identified two main approaches to cross-platform development[CY99]. One was developing a separate version for each platform, which would cause a lot of code to be rewritten from scratch every time. The second approach would attempt to write most of the software as generic, platform-independent code with preferably less to no code specific to a platform. They also found some costs or *penalties* to cross-platform development. Netscape engineers estimated 15-20% human effort and time penalty in design and coding for cross-platform versions of their product[CY99, P. 77]. They're also describing additional challenges in staffing the platform-specific experts, at one point leaving them with only one expert for a specific division, which ended up being a bottleneck. Additionally, integrating and testing all these different versions produced significant extra costs. A more recent study from 2013 focuses more on the variety of mobile platforms and the challenges that arise when developing cross-platform software for those [Smu12]. While platform-native APIs are often feature-rich, they usually force a specific language (e.g. Java on Android, Objective C/Swift on iOS) upon the developers. These developers need a development license in some cases, while having to subject to the respective terms of service to publish their application on the platform-specific ecosystems.
Web applications, on the other hand, are for the most part unaffected by these cross-platform challenges, as they are available on all internet-connected devices with a browser. *Progressive web apps* introduce a new class of web applications which attempt to close the gap between web apps and native apps by introducing new technologies and focusing on load times and UI performance. This thesis is driven by two key points: The first point is about investigating the viability of web applications as a whole and how the usage of *progressive web apps* can be useful. Secondly, *good practices* should be identified in order to assist in building a *progressive web app*.

1.1. Viability of Web Applications

In order to have merit in real-world applications, investigation has to be made about the exact capabilities of *progressive web apps*, especially compared to native applications and the requirements those usually fulfill. Web applications allow usage across any platform with a working browser, thus eliminating the need to acquire experts of many strongly varying fields. Another factor that benefits web applications is their ease of access, as no installation setup is required. Less trust is needed for users to try a web application, since every modern browser sandboxes the web application away from the actual filesystem, minimizing the risk of malware. Web applications do, however, bring their own share of disadvantages. They usually require a working network connection to start with and only function while opened in a browser. Web applications don't integrate into the operating system at all as their URL has to be typed out or accessed by operating the bookmarking/history UI of the browser, or even searched for using a search engine. Web applications are downloaded every time they are accessed and can cause further delays depending on the resources that need to be retrieved from the network. The apparent problem with web applications lies within being perceived as seemingly less capable, slower or of less quality than using native applications.
Progressive web apps try to address those disadvantages by introducing new technology like service workers, webworkers and web manifests among new concepts like PRPL, App Shell and others.

1.2. Identifying *good practices*

Another goal of this work is the identification of possible *good practices* for building progressive web apps. Traditional websites followed a rather simple pattern. The User requests a URL (often by typing manually or visiting a bookmark/history entry), the browser will then issue a HTTP GET Request to a webserver. That webserver would then serve an HTML document. The Browser then parses that document and checks for further requests to be made (usually further GET's to stylesheets, images and scripts). Visiting a new subpage on a site would often repeat this cycle, leaving the user waiting and the browser displaying a blank white screen in meanwhile. Since the browser loads resources asynchronous, bigger images would usually pop into the page slightly delayed, shifting contents around, which can be a quite jarring user experience.
Todays web applications evolved from back then, by moving most of the application logic to the frontend presented by the browser. Instead of wiping the current webpage with all its resources out of existence every time a request to a webserver is made, only the affected module of a webpage is updated on the spot. As a consequence, a lot of new concerns entered the world of frontend development. While, traditionally, developers wrote HTML to declare contents, JavaScript to modify said contents and CSS to change the layout, todays development often involves different languages which need an additional build step before the application can be deployed. Tooling presents a rather big, and seemingly overwhelming[Cle15] part of todays frontend development[Jan17]. In this case, *good practices* would suggest which practices, concepts, tools and patterns are useful and why, and where decisions should be made according to the underlaying

project.
Another aspect are architectural concerns, which, while assisted by tools, are ultimately decided by the software engineers at hand. This work aims to illustrate possible patterns and caveats when planning and implementing a progressive web app. *Good practices* entail structural, functional and usability concerns.

1.3. Approach

The approach to answer these questions is broken down in three parts. In the first part, a general overview over the current state of technology will be given, which will go into detail how modern web applications are structured before introducing technologies and patterns that are specific to progressive web apps. Finally, the overview will go into tooling and how the JavaScript ecosystem works.
The second part focuses on the concrete implementation of a progressive web app and illustrates decisions that may go into building one. For this, a case study will be held in which the CRC806 Database[Wil16] - an existing web application that hosts a variety of research data - will be rebuild as a progressive web app. The key challenges that appear during the implementation will be the major focus of this second part.
The third and final part is the evaluation of the practices that went into building a progressive web app. This will factor in suggested practices from different sources, as well as the findings that surfaced during the actual implementation. The evaluation process will then lead into the conclusion that aims to answer the key points stated earlier (see previous two sections 1.1 and 1.2).

Part I.

State of Technology

This chapter attempts to establish a theoretical foundation to achieve a better understanding on how web applications are build and what *progressive web apps* bring to the table. In order to achieve this, the chapter is broken down into three aspects.
The first chapter, *Building Applications in JavaScript*, will take a closer look into the technologies that are available on the web and how they can be utilized. If additional Browser-plugins are to be exempt, JavaScript is the only way to implement application logic in the browser.
The second chapter, will look further into progressive web apps and which technologies and concepts come with those.
Finally, in the third chapter, *Tooling*, we will explore the tooling that comes with modern frontend development and how they can improve or ease the development process.

2. Building Applications in JavaScript

JavaScript is an interpreted scripting language that allows developers to implement application logic within the browser. In the past, JavaScript was far from being as feature-rich as it is today. Technologies like Adobe Flash, Microsoft Silverlight and Java Web Applets were attempts to close that gap, but they brought their own issues. These technologies always required some sort of browser plugin to be installed and regularly updated in order to avoid security issues or incompatibilities. Contrary to the usually open web technologies, software based on those technologies, were often proprietary in nature and, in the case of Adobe Flash, needed commercial software to build upon. Running these plugins cross-platform would in some cases (e.g. Microsoft Silverlight) need workarounds or unofficial, community-maintained software.
With the advent of HTML5 and increased support by browser makers, JavaScript has become a real asset to modern web development. It is important to note that, while JavaScript itself is not standardized, it is heavily based on ECMAScript[Ecm17], which is a continuously evolving standard. The following chapters will illustrate how Applications can be build using todays JavaScript. For a better grasp on some of the newer ECMAScript additions, please refer to section 1 in the appendix.

2.1. Prototypical Inheritance

Object-oriented programming is an important tool as it allows developers to break down complex problems into small, solvable packages, hiding the complexity away behind abstractions. Object-orientation also greatly improves code reuse and modularity, which, as a consequence, allow for easy replacement of components. It is hard to argue against object-oriented programming, as it is the de facto standard in most modern applications. ECMAScript uses prototypical inheritance, which can be used very similar to traditional inheritance. Traditional inheritance uses classes as some kind of blueprint for potential objects, only upon instantiation they become working objects. Prototypical inheritance uses, as the name suggests, prototypes instead of classes to build these blueprints. The important distinction to make here is that prototypes are already fully functional objects before any instantiation happens. This can result in unwanted side effects when prototype-functions are used directly (see fig.2.1), but is rather easy to avoid by using prototypes after instantiation, just like classes in traditional inheritance.
Another effect of prototypical inheritance to be aware of is how object instances share the prototype they are based on. When the first object instance modifies a property that is also an object, any consecutive object instance will now share this modified property[Yen17]. This can be avoided by instantiating possible object-based properties inside the constructor and not inside the prototype. For a closer look into ECMAScript classes and how they work, please refer to section 2 in the appendix.
Looking at the principles of object-oriented programming, encapsulation, abstraction,

```
> Array.prototype.push("lol")
< 1
> var empty = [];
< undefined
> empty[0]
< "lol"
```

Figure 2.1.: Side effect of using a prototype function directly, opposed to using it on a properly instantiated array[Dai17]

inheritance and polymorphism[Ana16], ECMAScript (and in extension, JavaScript), seems to be capable of most of them. Inheritance is demonstrated in the appendix (see section 2 in the appendix) and abstraction comes down to using objects to break down a complex problem. Polymorphism is reached by overriding methods of the parent-class if needed. Only encapsulation is hard to use, since ECMAScript does not provide any *public*/*private* or *protected* keywords to limit visibility of object properties. Instead, workarounds have to be used where supposedly private properties have to be hidden behind function scopes and closures (see [Cro01]), which is a lot less intuitive compared to more traditional, object-oriented languages. The module pattern illustrates one attempt to solve the visibility problem (see section 3 in the appendix).
Another popular approach is the usage of pseudo-private properties where every property is prefixed with an underscore to signify how it must not be accessed from outside, similar to the *__proto__* where direct access is heavily discouraged[MDN17]. However, this does not in any form enforce visibility, but rather illustrates some kind of coding guideline in order for other developers to better understand which properties should or should not be accessed from the outside. There is no protection to any misuse of that convention.

2.2. Testing

Testing tries to evaluate the robustness of a system, especially when it goes through multiple iterations. Tests can be automated (e.g. through a build pipeline) in order to notify software engineers quickly after a test-breaking change has been made. This can make removing a bug or updating a test to reflect changes easier as the chances are high that the engineer is still within the mindset of that specific code change. Tests are usually distinguished by three categories.

Unit Tests Testing of individual functions, where a specific input has to be correctly mapped onto the correct output

Integration Tests Testing a set of modules in order to ensure their interoperability

Functional Tests Applying a test scenario on the system as a whole without regard of the systems specifics

Unit and integration tests respect the actual system and its components, and are thus considered as *white box testing*, opposed to functional tests which are considered as *black box testing* since the systems internals are not taken into account.
A study from 2016 [Shi16] shows a low popularity of JavaScript-testing among web developers (see fig.2.2). The entry to web development is relatively low, only needing

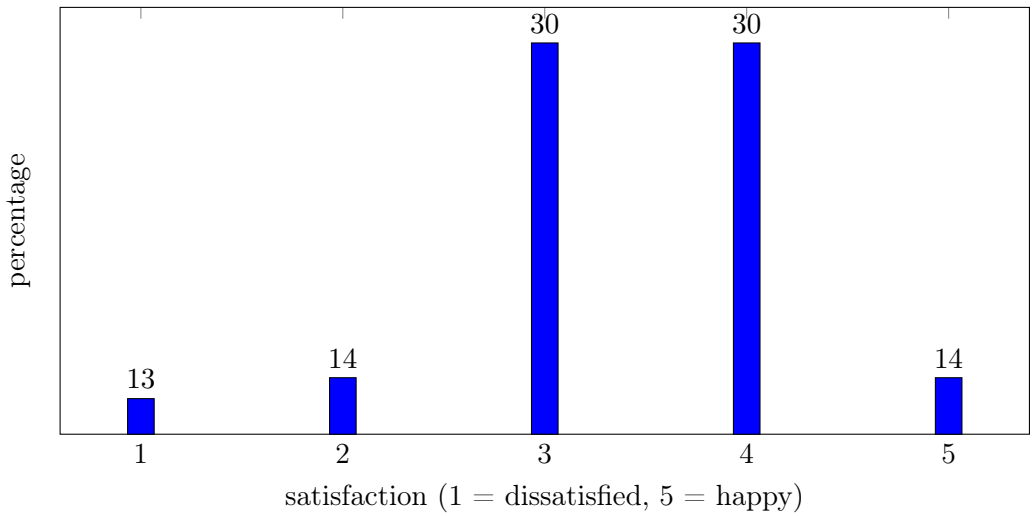

Figure 2.2.: Developer-Satisfaction of JS testing-tools from 1 to 5. Data-Source: http://2016.stateofjs.com/2016/testing/

an editor and a browser in regards to tools. For effective testing, a lot more tools (see chapter 4) need to be introduced and learned by developers which might be a possible reason for the low popularity. Vitali Zaidman listed common functionalities of testing tools in his article about JavaScript testing[Zai17]:

Provide a test environment A testing environment encapsulates itself from the actual in-production system in order to create a safe environment to run tests on without affecting the production system.

Provide a testing structure Testing structures help unifying how tests are structured, with *behavior-driven development* (BDD) being one of the more popular approaches among the established tools. It is often seen as a continuation of *test-driven development* (TDD) which usually operates on technical specifics to be tested on. Dan North describes BDD as an '"*outside-in*" methodology', which '...starts at the outside by identifying business outcomes, and then drills down into the feature set that will achieve those outcomes. Each feature is captured as a "story", which defines the scope of the feature along with its acceptance criteria.' [Nor07]. Agile development processes often also work with user stories to model software requirements. BDD illustrates a viable approach to translate those stories into the actual process of writing code. Testing

frameworks usually come with a way to define stories for specific modules or functions that are to be tested.

Provide assertions functions Assertions allow developers to compare an expected outcome with the actual outcome, when a specific input is given:

```
assert.equal( add(1,2), 3 )
// When inputs are 1 and 2
// the expected outcome of 'add' is 3
```

Generate, display, and watch test results This is the basic core functionality that is expected in order to know whether tests succeeded or not.

Generate and compare snapshots of component and data structures This ensures that changes from previous runs are intended.

Provide mocks, spies and stubs Mock objects can be used in place of actual software components. This allows a more controlled testing scenario and possibly faster test execution, e.g. in the case of accessing static example data instead of actually querying a database.
Stubs goes in a similar direction, but instead of replacing whole components, only select methods are replaced in order to force a certain behavior or object condition while any non-stubbed aspects of an object can still be subjected to tests.
Spies observe function calls, especially what causes specific functions to be called, where they are called and how often they are called.

Generate code coverage reports Allows to quantify how much of the code base is actually being tested.

Provide browser or browser-like environment with a control on their scenarios execution This is usually done by using browser automation like selenium[Sel] or headless browsers like PhantomJS[Pha].

All these functionalities can be provided by a set of additional tools, where some are able to address multiple points of that list, while others are more specialized on very few functions. Most additional tools need some kind of setup and need to be integrated well into the development process, which further illustrates the point of complexity.
The usefulness of these testing functionalities are dependent on the type of test to be performed. *Unit tests* may operate on mocked input data in order to assert whether a set of input will correctly map to an expected output. If tests are run across multiple units in form of *integration tests*, spies may deliver more insight whether all units are correctly calling each other. Asserts would be less meaningful at this points as the units themselves should already be tested and known to deliver the expected output on their own. *Functional tests*, which are black box tests, do not account for any specifics inside

the system, thus a browser or browser-like environment will be necessary in order to test scenarios similar to real-world user behavior.

2.3. Routing and Rendering

Traditional web sites used to operate on a document-basis, where the top-level domain would point to the document root and where subpages would follow a tree structure in the servers filesystem, containing many folders and more documents. In this case, an url like "*example.com/pages/about.html*" would have caused the browser to request the web service located at *example.com*, which would look for a folder named *pages* in its document root, and within that folder a document called *about.html*. With the advent of dynamic, server-side interpreters like PHP, ASP.NET, Ruby, Python and many more, this quickly changed. Instead of requesting different documents, a script was invoked with a HTTP GET parameter in order to specify the requested page. In such a system, the previous URL may be expressed with a new URL like this: *example.com/?page=about*. Instead of serving static files, a server-side application is invoked, with the key/value pair "*page=about*". The application then decides what content should be served and starts rendering the document just-in-time to be send back to the requesting browser. This approach is called *server-side rendering* (SSR) and comes with many advantages. Since the server forges the requested document, the provider of the web site will always be in control of what is rendered and how. Sensitive business logic does not need to be exposed to the client and can instead be processed on the server-side. As long as the server is able to keep up with incoming requests, the browser can start rendering the page instantly upon receiving the server response, since the HTML document is completely rendered at this point. Modern JavaScript

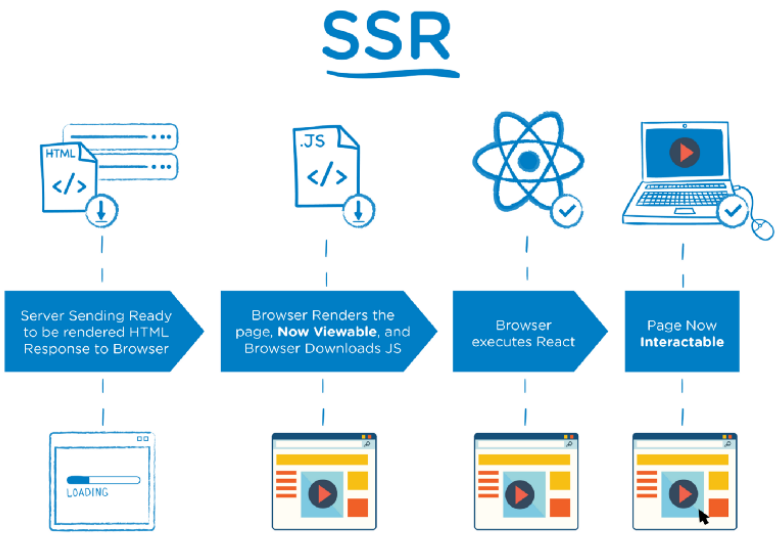

Figure 2.3.: Time until viewable/interactable on a site embedding a JavaScript library (React), using server-side rendering (see [Gri17])

applications on the other hand often use *client-side rendering* (CSR), where all routing happens inside the client. The advantage of CSR is an increased perceived (and often actual) performance, as the page does not reload from scratch every time a subpage is visited, but instead only the page contents are transmitted between client and server. This approach will, however, exchange network resources for processing power on the client side, since the browser is now responsible for rendering the page. With the current variety of browser-equipped devices, it can be hard to estimate how much UI processing can be offloaded to the client without slowing down a pages first impression (compare fig.2.3 and fig.2.4). Additionally, sensitive business logic still needs to be done server-side, requiring a more distributed architecture where client-side application logic and server-side application logic need to communicate with each other. In order

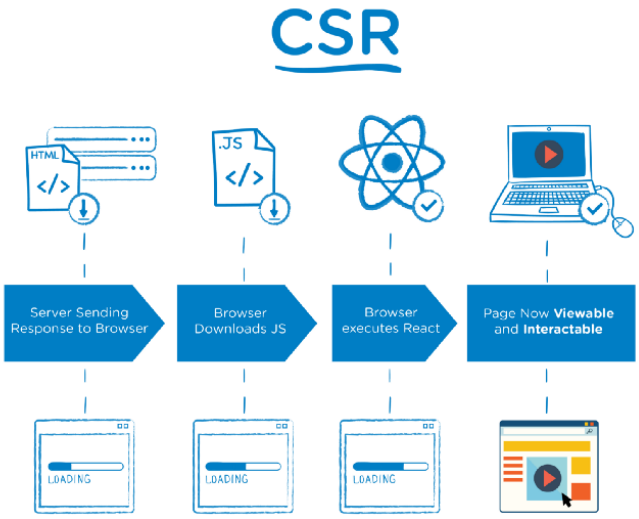

Figure 2.4.: Time until viewable/interactable on a site embedding a JavaScript library (React), using client-side rendering (see [Gri17])

to regain the ability to have distinct URLs for different pages, URL fragments are often used. The previous example URL could be expressed as "*example.com/#about*", using fragments. The client-side routing would then implement a *hashchange*-event to react when the URL fragment changes. Since the URL fragments stem from their original use of anchor links[1], most browsers will keep track of any fragment changes and automatically add them to the browser history. As a consequence a browsers back- and forward navigation methods will stay intact, despite not actually making network requests to different pages.
More sophisticated approaches use the history API of modern browsers. This allows a lot more freedom with the way URLs are formed. Instead of being forced to use specific URL fragments or conform to the HTTP GET parameter syntax, any URL can be pushed as a new state to the history API. Visiting a link could change the

[1] Anchor links allow to jump to a specific position on the current page, opposed to linking to a different page

URL to "*example.com/page/about*" without any folders or documents in the servers document root being named that way. This works due to only changing the visible URL in the browsers address bar, without actually sending a network request to the above mentioned URL. Additionally, the history API adds an entry to the browser history in order to jump back and forth between those states. One caveat with this approach is how URLs, manipulated by the history API, do not actually resolve on the server when actually following the link. Instead, the server has to be configured to redirect any requests, regardless of the specified URL, to the base URL, where a client-based routing mechanism attempts to resolve and render the requested URL.
In general, SSR-Rendered pages tend to have a longer *time till first byte* than their CSR counterparts, because of the initial processing done by the server. However, with SSR the contents can be rendered instantly upon receiving the network response, whereas CSR needs to finish loading additional JavaScript files in order to start rendering the page. There are also attempt to mix both, CSR- and SSR-rendered approaches with *isomorphic rendering* [Mar16]. This method uses SSR to pre-render the initial route in order to reach minimum *time till first byte*. However, any consecutive request is done via CSR to allow almost instant route navigation.
The upcoming chapter 3 will introduce new technologies, with some of them especially addressing the load times until a page becomes viewable and interactable.

3. Progressive Web Apps

This chapter will describe the core technologies to build *progressive web apps* (PWA), built upon the fundamentals presented in the previous chapter. Progressive web app is an umbrella term for a set of technologies and design patterns which aim to bring a *native-like* user experience to the browser by offering a quickly loading and continuously reactive UI with deeper integration into the underlying operating system compared to usual web applications.
The term *progressive web app* has first been coined by Google Developer Alex Russel and Designer Frances Berriman, which describe a class of applications with the following characteristics (points from [Rus15]):

Responsive Responsive UI elements ensure that information stays viewable on varying screen sizes and orientations.

Connectivity independent Critical resources can be pre-cached by a service worker (see section 3.1) in order to serve them from cache when no network connection is available.

App-like-interactions Use the app shell model (see section 3.2) to resemble interaction more similar to native apps.

Fresh While content may be fetched already up-to-date during the use of a progressive web app, the app shell can be updated by utilizing the service worker update process.

Safe Using HTTPS to ensure that requests are indeed sent to the intended recipient and responses have not been manipulated by anyone intercepting the connection (known as *man-in-the-middle* attack).

Discoverable Apps that use the web app manifest which contain essential meta data like title, icons, description and more (see section 3.5).

Re-engageable Allows access to re-engagement UIs of the OS. This usually boils down to push notifications, which may steer the user back to the progressive web app, even though the user may be engaged in another app right now.

Installable Allows to put an application shortcut to a home screen, a desktop, a launcher or start menu in order to integrate a progressive web app among other, native apps (see section 3.5), without relying on any specific app store.

Linkable Providing a URL allows to easily share an application with others. A progressive web app will be able to launch frictionless without any setup, once the URL is accessed.

This shows how the term *progressive web app* actually combines many different requirements or properties which actually depends on a set of technologies, with some of them being available for quite some time, rather than one specific technology. Using that term makes it easier to communicate those concepts and ideas and make them more marketable. From a software engineers perspective however, it's important to look behind the umbrella term and gain a better knowledge of the underlying technologies. That being said, progressive web apps are still evolving and new concepts and technologies may arise anytime, so while this work will attempt to elaborate on the core technologies, it cannot claim to paint a complete picture.

3.1. Service Workers

A *service worker* is a specification[Rus+17] for an additional, self-contained JavaScript file that may be installed during the execution of the regular JavaScript code of the current HTML document (see fig.3.1). Its main purpose is for web applications to exist beyond the browser tab. A successfully installed service worker can deliver cached contents when offline, send and receive push notifications or fetch data in the background, regardless of the web app currently being opened in a browser or not.
Service workers are based on web workers, which have been supported by every major browser since at least 2010. Web workers allow true concurrency for JavaScript applications, which are usually single threaded. This is mostly used to offload time-consuming processes without slowing the UI down (more about web workers in section 3.4). While a web application may use multiple web workers as needed, there can only be one service worker for a defined scope. Attempting to install another service worker will replace the active one. A service worker has multiple states that are described by the service worker lifecycle [Arc18]:

Installing This happens the first time a page that registers a service worker is called. On future requests, the service worker will usually be already installed from now on, so this only happens once.

Waiting When a service worker successfully installs, it will usually directly enter the *active* state, as long as there is currently no service worker active. However, if a previous service worker is active, the new service worker will enter the *waiting* state until the current service worker does not control any client anymore. The waiting phase can be skipped by calling *self.skipWaiting()*, which may be necessary when a faulty service worker needs to be replaced as soon as possible.

Active Service workers enter this state when they successfully registered upon the first visit, or activated on a consecutive visit. Only when this state is reached, service workers can receive *fetch* and *push* events.

One of the main benefits of service workers is the combination of the cache API with the ability to intercept *fetch*-events from the browser. These events are triggered by the

browser every time a resource is requested. When critical resources have been cached through the cache API, developers can either chose to instantly deliver them on fetch, or try to fetch from the network first and fall back on the cached resources if offline. Depending on the use case, developers can adopt different caching strategies[Ser]:

Network or cache The service worker will attempt to retrieve resources from the network if they load within a specified timeout, else resources are loaded from cache.

Cache only The service worker will always retrieve resources from the cache.

Cache and update The service worker will retrieve resources from the cache while also updating the cached entry for the next visit if possible.

Cache, update and refresh The service worker will retrieve resources from the cache and attempt to update the cached entry. If successful the cached content will be replaced by the current one.

Embedded fallback The service worker will attempt to retrieve resources from the network, if no network connection is available, resources are retrieved from the cache.

These caching strategies allow for fine-tuning the compromise between having a minimal load-time and maximum currentness of data.
In order to install a service worker, it needs to be registered (see fig.3.1). To speed up the initial page load, it is recommended to register the service worker once the page finished loading. Especially on low-end devices, the registration of a service worker during the initial rendering of the DOM can cause the application to become less responsive during that state [Pos18]. There are use cases where developers may want

```
1 if ('serviceWorker' in navigator) {
    window.addEventListener('load', function() {
3     navigator.serviceWorker.register('/sw.js');
    });
5 }
```

Figure 3.1.: Registration of the service worker (sw.js), once the site has finished loading

to register the service worker early on, in order to start caching immediately, however this may cause the application to load too many resources at the start which can possibly slow down the first visit. When all resources are prioritized to load at the beginning, nothing really is.

3.2. App Shell Model

The previous chapter touched on the topic of caching and different caching strategies. The *app shell* model is about decoupling the actual UI from the content and suggests the caching of all files necessary for rendering the UI, while the actual contents can still

be retrieved from the network if currentness is a concern, and eventually populate the UI (see fig.3.2). The goal is to instantly (low loading time) and reliably (regardless of network availability) load a web application similar to native applications.

When implementing the app shell model, a distinction has to be made about what to

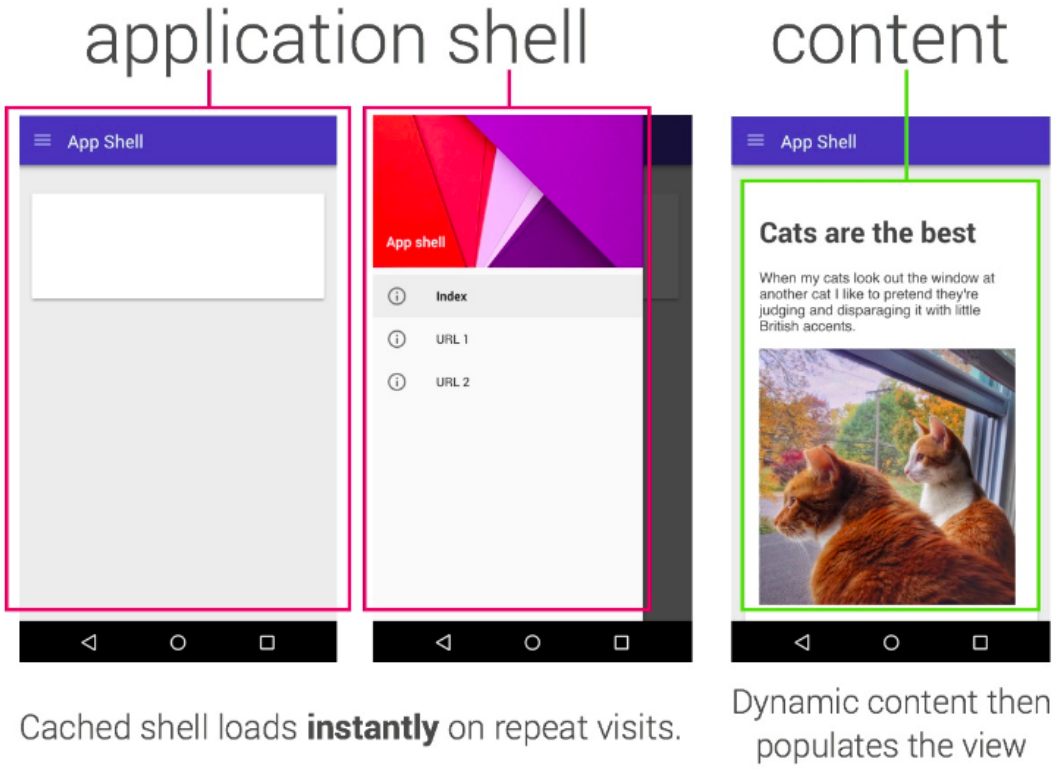

Figure 3.2.: Distinction between the app shell and content to load, source [Osm17a]

include in the initial UI. Even though the app shell is going to be cached for consecutive uses, it is important to reduce the amount of data needed to reduce the time for the *first meaningful paint* when a site is visited the first time. This will enable an increased perceived performance. The app shell should contain the following files (points from [Osm17a]):

- HTML and CSS for the *skeleton* of your user interface complete with navigation and content placeholders.
- An external JavaScript file (app.js) for handling navigation and UI logic as well as the code to display posts retrieved from the server and store them locally using a storage mechanism like IndexedDB.
- A web app manifest and service worker loader to enable off-line capabilities.

This approach adds a major benefit to *client-side rendered* web applications which, even though the *time till first byte* is very low as no preprocessing is needed, have a higher

load time for the *first meaningful paint* as the JavaScript responsible for displaying the site first have to be loaded and then processed in order to start building the HTML DOM. For maximum benefits, only resources *critical* for the initial load should be included in the *skeleton*. This topic will be explored more in-depth with the *PRPL-pattern*, presented in the next chapter.

3.3. PRPL-Pattern

This section describes the PRPL-pattern which stands for *push*, *render*, *pre-cache* and *lazy-load*.
Modern web applications tend to heavily focus on *client-side rendering* in order to increase the perceived (and often actual) performance when navigating between different *routes*. Purely *server-side rendered* pages reload from scratch when a different *route* is visited, causing a *white flash* which can significantly slow down the experience of navigating the application, especially compared to native applications. *Client-Side rendered* applications can display new *routes* quickly, as only the content has to be updated, however the downside is an increased initial load. This can be caused by having to deliver contents for all *routes* at once or having to include heavy-weight libraries in order to render the application.
The PRPL-Pattern, developed by Google, is an approach to organize a PWA in a way that the initial *route* loads as fast as possible while still gaining the general benefits of *client-side rendered* applications. PRPL stands for (taken from [Osm17b]):

Push critical resources for the initial URL *route*.

Render initial *route*.

Pre-cache remaining *routes*.

Lazy-load and create remaining *routes* on demand.

The goal is to achieve a minimum *time to interactive* on various devices, especially mobile device with limited processing/networking power.
This pattern, however, is still very new and still in the process of being tested. There do not seem to be definitive answers in how to achieve the four above points, however some suggestions can already be made. Pushing critical resources can be done in two ways. The first suggests using HTTP/2, which allows multiplexed downloads from the server[Osm17b]. If e.g. the browser requests an *index.html*-document, the server will understand which assets belong with it and automatically respond with the requested document, as well as any corresponding scripts, stylesheets and/or images. Before HTTP/2, a browser would only receive the requested document and would then parse it. Upon parsing, the browser will detect missing resources and will only then make further requests to the server which further delays the initial page rendering. With HTTP/2, by the time the browser parsed the document, additional resources may already finished downloading and thus can quickly be retrieved from the browser cache. However, since not all browsers and especially not all servers support HTTP/2, there is an alternative way to minimize the initial requests down to critical resources only.

This involves additional tools like module bundlers (more in chapter 4.2) which assist the development process by only including required assets on each page and combining shared scripts in an additional bundle so that it can be cached once for the whole application.
Rendering the initial route can be improved by sending down a *server-side rendered* page for a minimum time until *first meaningful paint*. Ideally, that initial page is pre-rendered to also size down on the *time till first byte*.
Further, likely to be visited, routes and their assets can then be preloaded. That way *client-side rendering* can be used on those routes without having to wait for the browser to download and process possibly heavy-weight JavaScript libraries as those have been preloaded while the user visited the initial route. One way to preload is using the *<link rel="preload">* HTML-Tag which will tell the browser to download a specified resource when idle. That way the initial downloading of critical resources will not interrupted and additional routes will be prepared with minimal effect on the browsing experience. Less important routes can be lazy-loaded on demand in order to minimize the transmitted data for users on an internet plan with limited traffic.

3.4. Web Workers

Similar to service workers, web workers are a specification[Hic15] of a self-contained JavaScript file, which allow the concurrent execution of JavaScript code outside the main UI process. This can be useful when e.g. slow calculations or filtering of large amounts of data needs to be done on the front-end. While there can only be one *service worker* for a site, there is no limitation (aside from technical limitations, due to available computing resources) on how many web workers are allowed to be active. Since web workers are detached from the UI process, it does not have access to the *document* and *window* global objects and thus cannot access the DOM at all. Having access to the DOM would defeat the point of web workers as tight coupling to the UI process could be reintroduced by the software engineer. This tight coupling would lead to possibly slowing the UI down again when taxing operations need to be performed, and taxing operations are the reason why one would want to offload a task to web workers to begin with. Instead, data can be exchanged between the UI process and web workers by using events. This type of interaction is comparable to message queues on other platforms where information is exchanged asynchronous.
It is important to distinguish between asynchronism and concurrency in JavaScript. While concurrency allows processes to run completely detached from the UI thread, initiating and receiving many asynchronous requests or responses will still affect it. As an example, AJAX requests are usually referred to as *non-blocking calls*, since the UI does not have to wait for an response to continue processing. However, if a larger response is to be received and processed, this can block the further execution of JavaScript to a noticeable extend. When this happens to the UI-Thread, it will negatively affect the perceived performance as the page will start to become unresponsive which will be most noticeable when scrolling the page or using inputs. As a consequence, even Web Workers will lose their benefits if large amount of data have to be passed between them and the UI Thread.

Some degree of unresponsiveness is often expected from web applications when compared to native applications. Web workers are one tool to address this and offer a much more *native-feel* in order to rise the acceptance of web applications for more complex tasks.

3.5. Operation System Integration

The previous chapters presented ways to improve the reliability and performance of web applications. What remained unchanged is the way web applications are accessed, which differs significantly from native applications. Native applications need to be downloaded and go through some kind of setup. This can be anything from a dialog based assistant to manually placing files in the file system. To simplify this process, most modern operating systems offer an application catalog (often called *store* or *market*) which present a list of installable applications, often curated by the OS manufacturer. Defining factors are an approachable catalog that assists in exploring possibly useful applications, as well as offering a one-click setup which does not require any input from the user. However, since not all applications are accepted into the respective stores, a manual setup still needs to be done on case-by-case basis. Once installed, applications can quickly launched by most operating systems and are handily available. Web applications do not go through a setup process by the operating system, instead they can be directly accessed by entering the URL into a browser. Not all users feel comfortable going through a setup process as they may not have the knowledge to do so, or do not trust the software to not do harm on their device or the data associated with it. In those regards, web applications are more accessible than native applications since their entry barrier is considerably lower. With the rise of mobile platforms, users became used to using stores, which grants a similarly lower entry barrier to native applications. Additionally, once installed, native applications integrate into the underlying operation system and can be launched easily. For web applications, even with consecutive use, the browser has to be launched first and the URL that points to the web application has to be made available again, either by retyping the URL, visiting a previously set bookmark or by searching the browser history if present. Once launched, the web application is still contained within a browser which does not necessarily comply with the design chosen for the application.
Web Manifest is an aiding technology that can help addressing the aforementioned points to some extent. The manifest allows to specify different properties that define the way an application looks and integrates by using a JSON-formatted document. Notable properties in regards to the points mentioned earlier are:

name Full name of an application.

shortname Shorter name to use where limited screen space is available (e.g. smartphone launchers).

display Specifies the degree to which the browser UI is visible. Options are *fullscreen* (the web application fills the screen completely, covering OS UI elements like taskbars, statusbars etc.), *standalone* (the web application fills most of the screen, the browser UI remains hidden), *minimal-ui* (most browser UI elements stay

hidden except navigational ones, varies by browser implementation) and *browser* (the web application will be displayed normally within the browser with all of the browser UI intact).

description Provides a short description about the application.

background_color Tints the initial splash screen that displays just before a web application loads.

theme_color Tints the browser window (if visible) and possibly other OS elements like appearance in task switcher (varies by OS).

icons Defines a set of icons for varying screen resolutions, which may be displayed in an application launcher or task switcher of the OS.

orientation Defines the preferred screen orientation of an application (e.g. portrait or landscape).

scope Defines the scope (in regards to the URL) where the application operates. Visiting a URL that points outside of the scope can then be launched in an external browser window instead of leaving the app.

Once created, the manifest file can be linked to an HTML-Document of an application by adding a *link*-tag to the document head.

```
1 <link rel="manifest" href="/manifest.json">
```

Most of the manifest is ignored until a user "installs" a web application. At the time of this writing there does not seem to be an accessible way to manually trigger that process. Instead browsers like chrome try to guess the degree to which a user engaged with a web application, if engagement is sufficient, an *app install banner* will be displayed which allows the user to install the application. Developers can not instantiate the banner manually (as a mean to prevent abuse, like popup ads in the past), they can however delay the display of that banner to an appropriate moment during software use. During some tests in the Android 6.0.1 OS, sticking the current website to the start screen would allow for most of the manifest to become active even without the *app install banner*. The only advantage installing through the banner would give was the addition of the web app icon to the software launcher, equal with native applications, and not to the start screen only, which happens when the user adds a web application to the Homescreen manually. Doing the latter will however still utilize other properties of the manifest, e.g. an application will launch fullscreen, with a splashscreen and separate to the browser application in android's task switcher view as long as the manifest says so. This shows that the utility and degree of integration a manifest offers, ultimately depends on both, the browser and the operating system manufacturer.
Additionally, Microsoft plans to crawl for progressive web applications using their search engine *Bing* in order to populate the Microsoft Store with them [Pfl+18]. This will make web applications further indistinguishable from native apps.
Service Workers also feature a Push API, which allows a web application to create notifications. Since the service worker remains active even without an open browser tab,

it is now possible for web applications to notify users much like native applications. This is utilized by the Twitter Lite PWA, which is able to display rich notifications on Android that come very close to native app notifications.

4. Tooling

Previous chapters have shown which concepts, ideas and technologies are used to build a modern web application. As more technologies are surfacing, the development process becomes more and more complex. Users are becoming more accustomed to a richer web experience which makes it challenging to deliver to all needs, especially as they become more and more implicit. It is not unusual to start a new web project by first setting up a complex tool chain - or - use even more tools to do the setup instead. This chapter will go into detail what types of tools are used and why it can be a benefit to have them as a part of the development process. The goal is to make the current JavaScript ecosystem more approachable as it is now. The specifics may change over time as the JavaScript ecosystem is moving at a very fast pace, which is why there is more focus on concepts as opposed to products.

4.1. Package Managers

Package Managers allow a separation between developed code and its dependencies. Not only are dependencies separated, but most importantly *managed* by the package manager. This brings multiple benefits to software developers. The actual code base becomes easy to share as it remains small, this is especially important when using version control software to keep track of changes or perform rollbacks if needed. Dependencies will not be checked into the version control system while changes of dependencies can still be tracked by checking in the project-specific configuration file that is read by the package manager. Another benefit is how the setup of a dependency is greatly simplified, since the package manager will automatically identify the correct source, perform the download and resolve new dependencies that the current one may bring. Finally, it is easy to manage the correct version of a dependency. That way legacy software can still be provided with outdated, but working, versions of a specific dependency when the updated version brings breaking changes. The usage of package managers is a popular practice on many platforms, as shown by tools like *maven* in Java, *composer* in PHP or *pip* in python.
In the current JavaScript ecosystem there does not seem to be any alternative to the NPM registry which hosts an constantly increasing amount of dependency packages. NPM originated from node.js, a platform to run JavaScript applications natively (e.g. on servers), and used to stand for *node package manager*. NPM, however, became increasingly popular as a package manager for webapp-specific dependencies (especially frontend-specific) and thus transcended the intended use of only providing packages for node.js-based applications. While there is only one notable package registry, there are different package managers to access that same registry. The first that comes to mind is *npm*, the original package manager included in node.js, which used to be the only one for some time. Another popular package manager is yarn, which also uses the

NPM registry but adds caching mechanism and other features to allow sped up use compared to npm. For this to work, both clients use a *package.json* configuration file that keeps track of dependencies. However, todays package managers bring another task to the table, as they are often also used as a *task runner*. Those tasks can also be defined within the *package.json* and allow have different tasks handy. As an example, there could be a simple *start* task that automatically boots up a development server which keeps track of file changes and reloads the page automatically, a *test* task that runs a linter to enforce code guidelines and tests to ensure code quality, or a *build* task that automatically bundles, transpiles (more on those in the upcoming chapters) and minify/uglify the codebase for maximum compatibility and minimum filesize. These tasks can be freely defined and often launch software that are also comfortably installable as an NPM package.
One benefit of package managers are maximal code *reuse*, this is often utilized by having dependencies that, again, depend on many other dependencies. Those subdependencies often bring in even more dependencies. Installing one dependency is often comparable to the tip of an iceberg where an unknown number of additional packages will be downloaded, which takes time and also brings some risks. There has been one incidents in the past where the NPM package *left-pad*, a simple module that allows to pad a given string with spaces or zeros, has been unpublished by the author, which broke a large amount of other NPM packages that depended on it[Sch16]. Notable ones include Babel, a very popular JavaScript transpiler that is crucial for many applications. This has been temporarily solved by another software author claiming the, now free to use again, *left-pad* package and uploading their version of a module that keeps the intended functionality intact. This can be very dangerous as it would have been possible for someone with malicious intent to claim that NPM package and inject harmful code in all projects that depend on it. This incident did bring a notable policy change however, as it is now harder to unpublish a package if this would cause other packages to break, while also replacing an unpublished package with a placeholder. That way, the name can only be reclaimed by going through the NPM support team which judges on a case-by-case basis whether there is a malicious intent.

4.2. Module Bundling

Years ago, websites used to offer very limited functionality and JavaScript code was limited to only a few lines of code, but as sites became more and more complex, those JavaScript files increased in size. At that point, having all of the code base in a single file makes it increasingly less maintainable. The obvious idea is splitting the code base into many files like it is the case at most other platforms. However, this can backfire quickly as the browser has to issue an HTTP GET request for every single file that the code is split into, which will drastically impact the initial loading performance. It becomes clear that the simple approach of merely using a browser and an editor for web development is just not appropriate anymore for modern web applications. The previous chapter touched on the topic of having a package manager to retrieve dependencies for a project, this is the point where one can capitalize on its second utility as a *task runner*. The task that comes to mind is a build step that bundles all

of the code base into a single file. To do that, a new tool has to be introduced: The *module bundler*.
Anytime the code base changes, one would run a build task that instructs the module bundler to apply multiple optimizations on the code base to obtain one or multiple bundled file. An increasingly popular module bundler is *webpack*, which seems to become the de facto standard at the time of this writing. It offers many functionalities beyond merely merging all JavaScript files together. Webpack works with ES6 modules and imports, so for any given HTML-File it does check which modules are actually important and builds a dependency tree of all important modules. It then uses a process called *tree shaking* to exclude any unused modules or module exports in order to exclusively bundle the needed code for any given page. Another useful functionality of webpack is the included development server. Once launched it will keep track of all source files of a project, if one of them changes it will quickly rebuild the project and reload the contents in the browser automatically. Webpack supports different *loaders* for different file types, which allows it to go far beyond merely bundling JavaScript files. These loaders allow webpack to also bundle CSS Stylesheets and other files that may be written in different programming or markup languages, which will then be transpiled into JavaScript and HTML, respectively (see fig.4.1). Transpiling is not only useful for

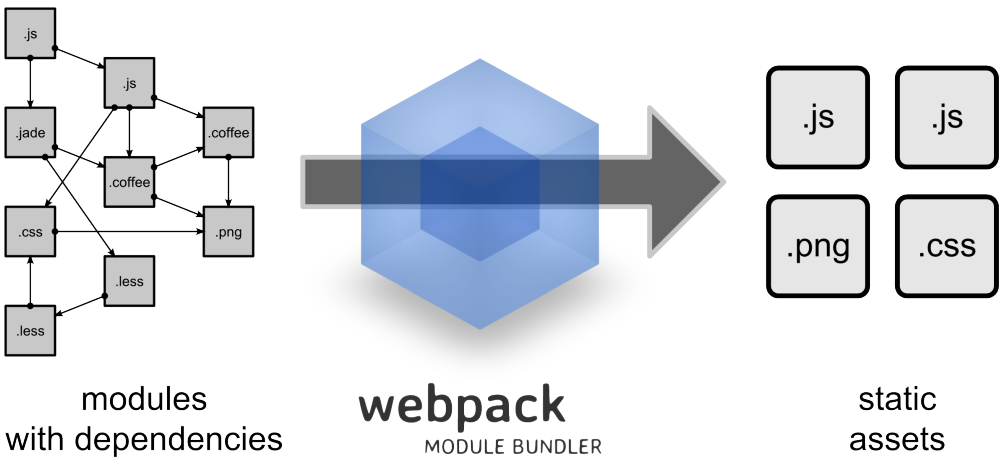

Figure 4.1.: Building assets with a module bundlers like webpack[web]

different languages, but also plain JavaScript files as will be touched on in the next chapter.

4.3. Transpiling

Transpiling is the process of converting code from one language into code from a different language while remaining a similar degree of abstraction [Fen12]. This is not to be confused with *compiling* which takes human-readable code and converts it into

machine-readable code, in the context of JavaScript, that would be the job of the browser's interpreter. There are multiple reasons why one would prefer to not use JavaScript in order to develop a web application. One may be that JavaScript is dynamically typed, which can be useful as developers do not have to convert between different types as often, but also introduces a whole class of possible errors. Some developers may prefer different styles than the typical C-style that JavaScript inherits, while others just need a more feature-rich language for their use case. However, transpiling does not limit itself to only JavaScript [Jan17]. There are also more feature-rich alternatives to CSS, like SASS or LESS which introduce variables and declarations for nested elements on top of many other features. Other Frameworks even introduce their own extension to known filetypes, e.g. *React* introduces JSX files [Mil15] that allow a mixture of JavaScript code and HTML to capitalize on the inherit tight coupling that view and view-logic already have [Hun13]. For plain text formatting (e.g. for articles and blogs), there seems to be an increasing trend of using *markdown* to style texts, a declarative language that is already widespread in wikis or README files, which is then transpiled into plain HTML.
However, not all developers are interested into increasing the complexity of the build process by introducing transpilers, especially if they may not be familiar with the mentioned other languages to begin with. There is still one compelling argument to be made to still use a transpiler: Browser compatibility. This has been an infamous issue in the past, where the inability of certain browsers to implement modern web technologies stunted progressing further in these technologies as a whole. Many useful technologies took a long time to be adopted, in order to account for those older browsers. Using a transpiler allows developers to write code in modern ECMAScript 6+ while the transpile process outputs ECMAScript 5 code that will run in most browsers. *Poly-fills*, to fill in missing features, will be automatically added by the transpiling process. This allows developers to adopt modern functionalities today, that would otherwise need to be implemented by hand which can cause errors and will increase the amount of time needed.
At this point of time, babel seems to be the de facto standard for any transpiling needs and can be extended with many different presets for different languages. Babel is used by many companies that have proven to push the state of web technologies further, like Facebook, Netflix, Mozilla, NPM, React and many more, to only name a few[Bab].

Part II.

Building Progressive Web Apps

This part of the work will attempt to guide through developing a progressive web application from scratch. A case study will be used to apply the concepts presented in the previous part, by building a new application. The first chapter *The Case Study*, will go into detail what the current system constitutes and what motivates rebuilding the application as a progressive web app. Additionally, organizational constrains and requirements, as well as general requirements for a PWA will be explored, in order to quantify if the resulting prototype is able to appropriately address the research questions outlined in the introduction.
The second chapter, *Implementing a Progressive Web App*, will document how to approach the development process and which decisions are to be made. Theoretical principles that have been mentioned in the first part of this work will be applied here.

5. The Case Study

The case study is a tool to gain the knowledge required to answer the research questions that have been stated in the introduction. For this, a closer look will be taken at an existing legacy system of which a subset of functionalities are to be reimplemented in the form of a progressive web app. While some requirements are highly specific to the underlying case study at hand, it does offer a platform to experiment with pwa-specific technologies and patterns. Requirements relevant to the implementation of a progressive web app will be highlighted in section 5.2.

5.1. Introducing the CRC806 Database

The Collaborative Research Centre 806 database (CRC806-Database, http://crc806db.uni-koeln.de) is a web database for scientists of the CRC806 to archive publications, datasets and other resources that constitute their work.
The CRC806 is funded by the DFG ("*Deutsche Forschungsgemeinschaft*", a german research alliance) and spans across different institutions and disciplines. Part of the cooperation are the universities of Cologne, Bonn, and Aachen, which work together to research across the fields Geosciences and Geography, Archaeology, and Antropology[Wil16, P.20].
Paleoenvironmental information is rarely available in reusable GIS (*Geographic Information Systems*) data formats, instead the data is represented in a variety of (structured/unstructured) formats and (digital/analog) media [Wil+17, P.40]. One of the main goals of the database lies within making existing information available in standardized GIS formats, while also collecting, storing, analyzing and publishing these data for other researchers to use [Wil+17, P.40].
The database gives scientists a permanent place where their work can be referred to, as well as the ability to acquire a DOI (*digital object identifier*) which grants a permanent link that can be used for immutable media like print media. If the actual URL were to change, this can be reflected by updating the meta data for an issued DOI to ensure that the DOI-URL will always remain valid.
Additionally, the database allows visitors to explore existing data, many of which are unrestricted. Upon uploading data, maintainers have to state to which degree their data can be used, this allows the system to display an appropriate license for any visitor that comes by. A license is a crucial part for others to know whether they are allowed to use the presented data right away, without having an additional barrier of contacting the author. The database encourages choosing an open license by applying a *5 Star Open Data*-rating[KH] to all datasets which will be publicly displayed.
DFG-funded projects introduce a time-constrain; Funding happens in phases, with each phase spanning across four years. Up to four years may be granted for a maximum total of 12 years funding[Wil16, P.20]. As the database is currently in the third and

final phase, further development of the database will eventually be suspended. As server technologies continue to change or grow, the database may become harder to host, which is largely based on the heavily distributed architecture (see fig. 5.1) of the current system which introduces many dependencies. Especially front-facing technolo-

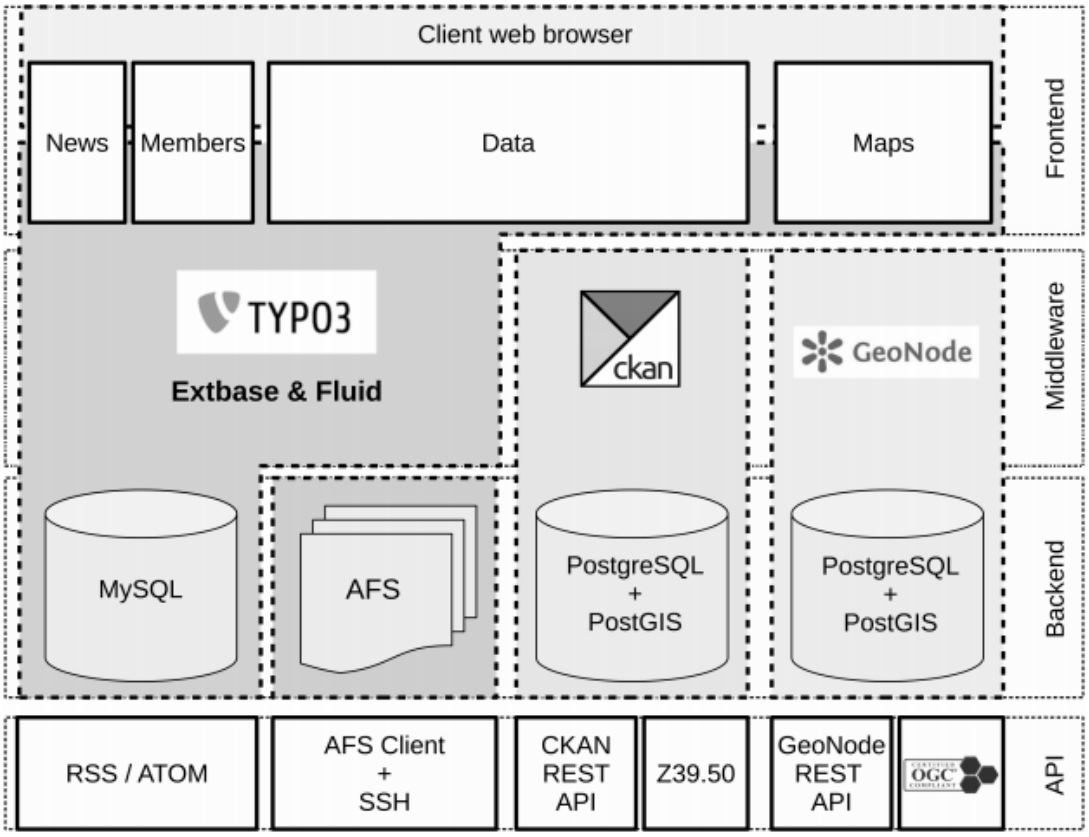

Figure 5.1.: Current architecture of the CRC806 legacy system[Wil+16]

gies like the underlying Typo3 Content Management System (CMS) may be subject to future findings of security holes and updating the CMS will likely include some breaking changes which need very specific knowledge to resolve. Those factors will likely endanger continued hosting of the database long after development stopped.
One suggested solution included caching the current application completely in form of static HTML/CSS files and host those for future access. Adding new data will not be possible with this approach, but the main concern of the database is the continuous hosting of the current data, to not compromise on the concept of offering a permanent place to present the uploaded data. This work attempts to iterate on the idea of transforming the current database to HTML/CSS files and build a progressive web application instead, which capitalizes on the increased focus on front-end technologies which may be demanding on the client, but less so on the server-side.
These and other factors create constrains and requirements that are further elaborated in the next chapter.

5.2. Requirements

Proper requirements are an important asset to build a system that will be able to actually fulfill its intended use. Requirements for the CRC806 Database are collected through three means:

- Adopting requirements of the current CRC806 Database website [REQ-CRC806]
- Organizational requirements of the underlying institution (University of Cologne) [REQ-ORG]
- Best practices for Progressive Web Apps [Goo17c] [REQ-PWA]

The collected requirements are then written out using the template provided by Chris Rupp et al [RSH09, P.215-245] in order to avoid common linguistic inaccuracies. Finally, the requirements are categorized into three of the categories provided by the KANO model (see fig. 5.2) in order to be able to prioritize requirements [Ver14]. To

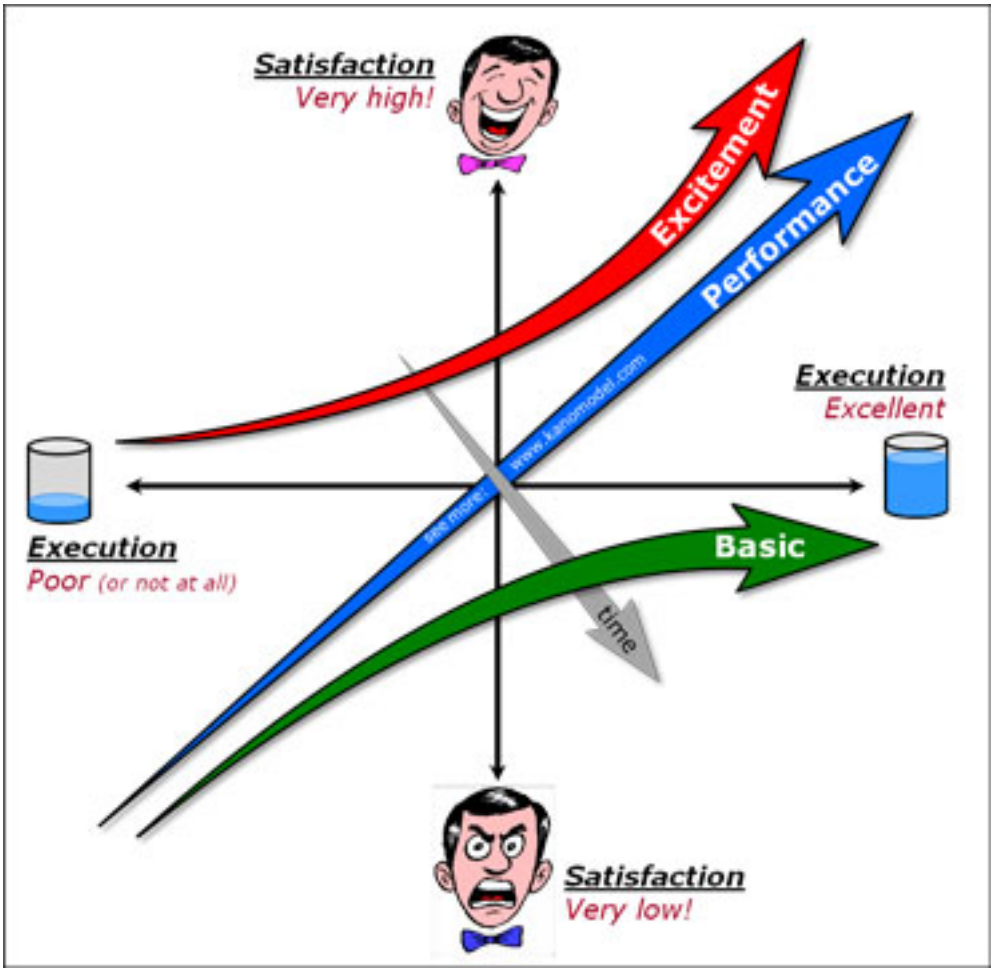

Figure 5.2.: How basic-, performance and excitement factors will influence satisfaction based on their execution. Source [Ver14]

avoid further inaccuracies, ambiguous terms are defined in a requirements-specific glossary and if there are synonyms, only the word in the first column is used consistently in the requirements definitions.

term	definition	synonyms
web app	A website with emphasis on user interaction and performance. Development focus lies heavily within front-end technologies. In this context, refers to the system that is to be developed.	-
website	A website contains multiple web pages of an institution, providing information within a browser. In this context, refers to the current CRC806-Database that is to be replaced at a later time by the upcoming web app.	-
mobile device	Any device with a working browser but restrictions regarding screen estate, input methods, network connectivity and processing power.	smartphone, tablet
home screen	A UI presented by a fully booted mobile OS where applications can be launched from.	start screen, launcher

5.2.1. Basic Factors

Basic factors are seen as mandatory and do to not directly improve satisfaction. Failing those requirements will, however, take a large toll on satisfaction. Basis factors often subconscious and as a result not always completely identified due to that nature.

Requirement	Description	PWA-Relevant
REQ001	The web app shall provide users with the ability to view datasets and publications with the respective metadata [REQ-CRC806]	No
REQ002	The web app shall provide users with the ability to search for datasets and publications using keywords [REQ-CRC806]	No
REQ003	The web app shall provide users with the ability to list and filter existing datasets and publications [REQ-CRC806]	No
REQ004	The web app will inherit the URLs from the current website in order to keep existing links to the site intact [REQ-CRC806]	No
REQ005	The web app shall be able to function with a web server (e.g. apache, NGINX) as the only dependency [REQ-ORG]	No

REQ006	The web app shall be able to display datasets and publication data without requiring to load JavaScript assets. [REQ-CRC806]	No
REQ007	The web app shall provide users with the ability to view research sites with their metadata [REQ-CRC806]	No
REQ008	The web app shall display the location of a research site on an interactive map [REQ-CRC806]	No
REQ009	The web app shall provide users with the ability to search for research sites using keywords [REQ-CRC806]	No
REQ010	The web app shall provide users with the ability to list and filter existing research sites [REQ-CRC806]	No
REQ011	The web app shall provide users with the ability to view map layers with their metadata [REQ-CRC806]	No
REQ012	The web app shall provide users with the ability to search for map layers using keywords [REQ-CRC806]	No
REQ013	The web app shall provide users with the ability to list and filter existing map layers [REQ-CRC806]	No
REQ014	The web app shall provide users with an interactive map to explore all existing research sites and datasets/publications that contain location data [REQ-CRC806]	No
REQ015	The web app shall be served over HTTPS [REQ-PWA]	Yes
REQ016	The web app shall be able to provide mobile users with a responsive design [REQ-PWA]	Yes
REQ017	When accessed offline, the web app shall be able to respond with a HTTP 200 status code, presenting some content [REQ-PWA]	Yes
REQ018	The web app shall provide users with the ability to add the web app to their home screen [REQ-PWA]	Yes
REQ019	The web app shall be able to become interactive under ten seconds on a simulated 3G network [REQ-PWA]	Yes
REQ020	The web app shall work in current versions of Chrome, Edge, Firefox and Safari [REQ-PWA]	Yes
REQ021	The web app shall be able to provide page transitions in order to increase the perceived performance [REQ-PWA]	Yes
REQ022	The web app shall be able to provide a unique URL for each individual page [REQ-PWA]	Yes

5.2.2. Performance Factors

Performance factors are conscious requirements which actively remove dissatisfaction and have potential to create satisfaction to some extent.

Requirement	Description	PWA-Relevant

REQ023	When used on a mobile device, the web app should be able to stay functional and completely usable [REQ-CRC806]	Yes
REQ024	The documentation will provide administrators with a workflow to migrate data from the existing website to the web app [REQ-ORG]	No
REQ025	The web app should embed schema.org metadata in order to improve the appearance in search engines [REQ-PWA]	Yes
REQ026	The web app should use the history API instead of fragment identifiers [REQ-PWA]	Yes
REQ027	The web app shall be able to become interactive under five seconds on a simulated 3G network [REQ-PWA]	Yes
REQ028	The web app shall use a cache-first caching strategy [REQ-PWA]	Yes

5.2.3. Excitement Factors

Excitement factors are often unknown factors that are rarely expected but will directly increase satisfaction if fulfilled.

Requirement	Description	PWA-Relevant
REQ029	After the first visit, the web app should be able to provide the user with basic metadata for datasets, publications, research sites and map layers without requiring a working network connection [REQ-CRC806]	No
REQ030	The web app should present contents in a way that elements do not jump as the page loads [REQ-PWA]	Yes
REQ031	When the user goes back to a previous page containing a list, the web app should be able to restore the scroll position of that list [REQ-PWA]	Yes
REQ032	When an input is selected that opens an onscreen keyboard, the web app should ensure that the input will not be covered by the keyboard or another element [REQ-PWA]	Yes
REQ033	The web app shall inform the user when accessed offline [REQ-PWA]	Yes

5.3. Architectural Challenges

The current system uses heavily distributed architecture to store and retrieve data. All the data has to be collected at one point and serialized to function as the input for the static site generation. Accessing the data sources by themselves does not work as it does need further processing to resolve relations between those different sources or to simply refine the data before it can be displayed. Data could still be retrieved from

their respective sources and the processing could be reimplemented in an import script, but it is much more time-efficient to use routines that are already in place in the current system. Therefore an export interface has to be implemented in the current system to deliver appropriate sources to base on site generation for the new web application.
A hard requirement for the web application is the display of data without JavaScript (REQ006), which dictates the need of server-side rendered content in order to display data even before a JavaScript bundle is loaded or in case JavaScript execution is blocked. However, in order to successfully implement the *app shell model* (see chap.3.2 on page 15), for further navigation only the contents should be replaced instead of reloading the whole page. This demands generation of static pages with traditional links to other pages if JavaScript is disabled. With JavaScript enabled, these links should be intercepted and instead new content should be loaded via JavaScript in order to achieve faster perceived and actual loading times. Perceived loading times become faster due to avoiding the white flash when the current page unloads and before the DOM of the new page is loaded. The actual load time speeds up too as only the actual content has to be retrieved opposed to having to also loading the shell on every consecutive page load. If pages are loaded dynamically through JavaScript, the History API[MDN18b] has to be utilized in order to restore browser forward/backward navigation as expected.
More challenges will likely surface during planning and implementation, starting in the next chapters, but those mentioned challenges already stand out after a rough look at the requirements listed previously.

6. Implementing a Progressive Web App

The first part of this work attempted to establish basic patterns and practices to build a modern web application, while this part started out with introducing the case study. This chapter will apply the knowledge from that first part to the case study in order to test which practices work well and which do not. In order to do so, this chapter will reiterate some points from the first part in order to concentrate more on the technical implementation side, opposed to more general factors that have already been mentioned in chapters 2 and 3.

6.1. Preparing the Environment

As web development becomes increasingly complex, it becomes harder to build everything from scratch. There is little reason to rebuild a functionality on the side when there are well-tested and long-standing libraries, build by authors which concerned themselves with the matter at hand on a deeper level. It can be concluded, that it is wise to use a package manager to capitalize on the existing JavaScript ecosystem. For this, Node.js needs to be installed first, which is available for all major operating systems. Node.js comes with the package manager *npm*, once that is installed different package managers could be installed via npm, but for now npm will suffice.
New projects can be started by calling the "*npm init*" CLI command inside a chosen directory. A quick assistant will ask for some meta information like project name and version, but will assume useful defaults if nothing is supplied (e.g. current folder name as the project name). As a result, a *package.json* file will be generated containing the provided information among others in the JSON format. From now on, every time a package is installed, it will be added as a dependency in that package.json file, while also downloading it in a folder called *node_ modules* inside the project folder. Exceptions to this behavior are packages that are installed globally, this is useful for tools that are not project-specific, which can then be invoked anywhere with just the name of the binary. Stating an absolute path is not necessary for globally installed packages.
From this point on, additional tools like module bundlers or transpilers can be installed, but instead a different approach has been used. Since static site generation is a heavy focus with the underlying case study, the project is initialized by using exactly such a generator. In this case *react-static*[Reac] has been used to generate a new project. To do that, *react-static* has been installed globally using "*npm install -g react-static*". In the next step, a new react-static project is generated by invoking "*react-static create*" which will start a quick assistant, similar in scope to the assistant that starts when a new npm project is initialized. Additionally, the assistant will ask if a template should be utilized, for this project, the *basic* template has been used.
Once this setup is finished, react-static will have generated a full project with a preconfigured module bundler and transpiler among other things. However before going

further into site generation with react-static, there needs to be a closer look to the frontend framework in use, namely *react*, and why it has been chosen for this project.

6.2. Choosing a Frontend Framework

Web applications in the past had an easy-to-follow program flow, a request is sent to the server, the server did some processing and answered with the resulting document. By passing HTTP POST/GET parameters in the request, the server may answer with different documents, but the application life cycle always started with the initial request and ended with a response sent. The downside of this approach is that anytime the UI needed to change, a new HTML document would need to be generated by the server to reflect that UI change. Especially with forms involved, this can negatively impact both, the users and the developers experience. Users are interrupted while doing their inputs in order to wait for the UI to react and developers have to strictly set all inputs with the user-provided data again in order to avoid data loss. Nowadays, those tasks are often handled client-side at the frontend in order to offer a better user experience and speed up the process for the user.
However, filling out forms is still only a basic task and modern web applications may need to offer much more demanding UI tasks. Web applications may offer custom elements, not covered by the HTML standard, to interact with. They may control audio or video playback or need to offer rich text editing and formatting tools. In fact, large part of this document have been edited in a web application that offers formatting tools and generates a continuous live preview during the writing process. The point is, web applications lost the initially simple program flow that ended once the page has loaded. They now have to react to many different types of events which introduces asynchronism to an originally synchronous flow. UI state has to be actively managed and this can grow into a complex tasks quickly. Frontend frameworks become a real asset in those scenarios, as they simplify the process by offering specific tools and forcing a certain style of arranging program code in order avoid typical problems that may occur in that domain.
There are many frameworks to chose from, which can make it hard to come to a decision. If the goal is to build an application that needs to be maintained long-term, maturity of a framework becomes a helpful criteria. The JavaScript ecosystem is a very fast moving place, if a specific frameworks stays popular for years, it is safe to assume that it successfully managed to handle most use cases that emerged within that time. Additionally, a mature frameworks usually went into many iterations, which may improved it at a finer level than younger frameworks. For this project, angular[Ang], react[Reaa] and vue.js[You] have been taken into closer consideration, which have proven to be among the three most popular frontend frameworks (see fig. 6.1).
Maturity is not the only important criteria though, Jens Neuhaus mentions additional factors[Neu17]:

- Are the frameworks likely to be around for a while?
- How extensive and helpful are their corresponding communities?

Figure 6.1.: The top 5 most popular frontend framworks for 2017 [bes17]

- How easy is it to find developers for each of the frameworks?
- What are the basic programming concepts of the frameworks?
- How easy is it to use the frameworks for small or large applications?
- What does the learning curve look like for each framework?
- What kind of performance can you expect from the frameworks?
- Where can you have a closer look under the hood?
- How can you start developing with the chosen framework?

Angular

Angular[Ang] is a UI framework developed by Google. While the first version of angular ran in vanilla JavaScript, the newer angular v2 forces developers to use TypeScript which offers features on top of JavaScript and thus needs to be transpiled down to JavaScript in order to execute in the browser. This among other changes, made the current angular in most parts incompatible to angular 1. TypeScript offers a strongly typed language with features still missing in ES6 like interfaces among others. However, with ES6 evolving at the pace as it does right now, it is hard to gauge the long term relevance of TypeScript and if it is worth locking on this dependency. While there does exist a migration guide from JavaScript to TypeScript, the reverse does not seem to be the case and transpiled code is not exactly well maintainable. One defining feature of

angular is two-way data-binding, which allows to either change a value in code which is instantly reflected in the frontend, or change a value at the frontend which will changes the value in code. This does introduce a whole class of possible errors for developers to make though, and adds a certain degree of complexity when reading code as it is not always clear how values are changed.

Vue.js

Vue.js[You] is not quite as mature as the other two Frameworks, but became increasingly popular (see fig.6.1). It is very similar to react as both prioritize performance and offer a component-based approach (detailed later). It is less opinionated compared to other frameworks as it allows for both, clean HTML-files with directives as attributes or HTML-in-JS with their own *.vue*-files. Directives are a way to allow some logic in a template declaration, e.g. if there is a list of items from a data source, a list item and what it contains only needs to be declared once and a looping directive will ensure that this declaration will be repeated for any amount of actual items. With a HTML-in-JS approach, developers are free to use native JavaScript functions like *Array.prototype.map()* to apply specific HTML to each element of a list of items and do not need to learn a framework-specific set of directives. A strength of vue.js are single-file-components that can contain a component's structure (HTML), logic (JS) and layout (CSS) at one place. This may seem like a *code-smell* as software engineers are used to separate those technologies, however react does make a very strong point for this approach which will be detailed in the next sub-chapter.

React

React[Reaa] is a frontend framework developed by Facebook which also uses it a lot in their own products, opposed to angular which Google seems to use sparingly[Cor17]. React was one of the first frameworks that titled angular's *two-way data-binding* as a bug and instead fell back to a more straight-forward one-way data-binding approach which many other frameworks like vue.js and Ember also adopted. Future versions of react changed many things under the hood but the api remained mostly the same, which caused very little trouble when upgrading[Neu17].
React does not allow for the usage of directives, instead native JavaScript has to be used. To simplify writing HTML declaration, react recommends the usage of JSX files, which allow HTML in JS (just like .vue files do for vue.js). It is important to note that JSX is not react-specific and can be used completely independent. Directives in HTML try to bring JavaScript into HTML, while react brings HTML into JavaScript. Neuhaus prefers the latter as JavaScript is much more powerful than HTML[Neu17]. By mixing HTML templates with view logic, react started to break a long-lasting best practice, which may sound like a bad idea at first. Software engineers often like to separate those two with *separation of concerns* in mind. Pete Hunt describes that term as "*Reduce coupling, increase cohesion*" in his talk about react[Hun13] and goes into detail how markup is tightly coupled *and* cohesive with display logic and should not be separated. He further describes a separation of HTML and JavaScript as a *seperation of technologies*, not *concerns*. React trusts software engineers to build many smaller

components to truly separate concerns instead of separating technologies. The tight coupling is especially prominent with template engines that offer *partials* (reusable template snippets) as the parent template often has to pass many parameters to the partial for it to function. Hunt also emphasizes the previous point that template directives are a less powerful re-implementation of existing JavaScript functions that need to be relearned for any specific framework at hand. ECMAScript on the other hand, ensures a consistent, framework-agnostic standardization of JavaScript functions. Additionally, since the markup is now part of the component, it becomes easier to write tests for.
React use multiple optimizations to increase application performance. The most notable one is the virtual DOM, a tree structure of plain JavaScript objects that contain information like the HTML tagname, HTML attributes and child objects, if any. This virtual DOM mirrors what the actual DOM consists of. When the application state changes, a quick check will be performed if any changes have to be performed on the DOM by checking the virtual DOM first. Instead of re-rendering the whole DOM, only the parts that change will be re-rendered. Additionally, if multiple DOM changes need to happen, react is able to apply all at once. This is important because changes on the real DOM will cause the browser to recalculate the whole page layout, which is a rather costly task. Applying changes as a batch, will minimize the number of times, this task is carried out, resulting in an increased performance. Another trick is applied in the way JavaScript events are bound to elements. E.g. if a table contains many rows with buttons, instead of binding a handler to each single button, react will instead bind only one click-event to the whole table and delegate the event to the respective button internally.

Conclusion

Ultimately, the decision depends highly on the requirements for the application that is to be developed. Out of the big three (angular, react, vue.js) all of them will be able to build rich interfaces in little time. Angular offers a more complete package, as it also takes care of other demands, not specific to the UI, like routing. However, that also makes angular more opinionated, as it covers a larger part of the architecture which may increase the hurdle of replacing specific parts with other implementations. This is also apparent with the hard requirement of using TypeScript and adopting a specific folder structure, whereas the other two frameworks are less restrictive in that regard. Performance-wise, all three will be quite comparable, with angular having a slightly higher memory footprint than the others (see fig. 6.2). The difference is usually not notable enough to impact the decision making though.

Vue and react both offer an optional way of using HTML in JS (via .vue or JSX files) which allows developers to use mostly standard-conform HTML (with few exceptions) which is easier to be understood, maintained and extended by designers that may be more familiar with HTML than the ever-evolving pool of JavaScript libraries. Vue also offers the usage of directives in HTML, similar to angular, to allow JavaScript in HTML, which is less powerful and also forces developers and designers to learn framework specific directives.
For the time being, react has the broadest ecosystem which offers multiple solutions

Duration in milliseconds ± standard deviation (Slowdown = Duration / Fastest)

Name	react-v16.1.0-keyed	angular-v5.0.0-keyed	vue-v2.5.3-keyed
create rows Duration for creating 1000 rows after the page loaded.	187.6 ± 4.3 (1.1)	185.7 ± 7.8 (1.1)	169.2 ± 3.6 (1.0)
replace all rows Duration for updating all 1000 rows of the table (with 5 warmup iterations).	165.2 ± 7.0 (1.0)	179.3 ± 6.5 (1.1)	181.8 ± 3.9 (1.0)
partial update Time to update the text of every 10th row (with 5 warmup iterations) for a table with 10k rows.	93.6 ± 5.6 (1.3)	73.5 ± 4.9 (1.0)	168.1 ± 7.4 (2.3)
select row Duration to highlight a row in response to a click on the row. (with 5 warmup iterations).	12.4 ± 4.1 (1.0)	7.6 ± 4.0 (1.0)	9.8 ± 2.5 (1.0)
swap rows Time to swap 2 rows on a 1K table. (with 5 warmup iterations).	19.6 ± 4.7 (1.0)	20.1 ± 4.2 (1.0)	21.8 ± 4.5 (1.1)
remove row Duration to remove a row. (with 5 warmup iterations).	51.5 ± 2.0 (1.1)	46.1 ± 2.6 (1.0)	52.5 ± 1.8 (1.1)
create many rows Duration to create 10,000 rows	2033.7 ± 32.0 (1.3)	1682.0 ± 53.1 (1.1)	1521.4 ± 55.7 (1.0)
append rows to large table Duration for adding 1000 rows on a table of 10,000 rows.	271.8 ± 9.9 (1.1)	257.6 ± 11.1 (1.0)	338.4 ± 10.3 (1.3)
clear rows Duration to clear the table filled with 10.000 rows.	224.4 ± 6.0 (1.0)	360.3 ± 16.4 (1.6)	240.9 ± 11.4 (1.1)
startup time Time for loading, parsing and starting up	49.4 ± 0.7 (1.0)	88.8 ± 2.9 (1.8)	48.4 ± 2.4 (1.0)
slowdown geometric mean	1.09	1.15	1.15

Memory allocation in MBs ± standard deviation

Name	react-v16.1.0-keyed	angular-v5.0.0-keyed	vue-v2.5.3-keyed
ready memory Memory usage after page load.	3.7 ± 0.1 (1.0)	6.7 ± 0.1 (1.9)	3.6 ± 0.1 (1.0)
run memory Memory usage after adding 1000 rows.	7.6 ± 0.0 (1.0)	10.5 ± 0.0 (1.5)	7.2 ± 0.0 (1.0)
update eatch 10th row for 1k rows (5 cycles) Memory usage after clicking update every 10th row 5 times	8.5 ± 0.0 (1.2)	10.6 ± 0.0 (1.5)	7.3 ± 0.0 (1.0)
replace 1k rows (5 cycles) Memory usage after clicking create 1000 rows 5 times	9.0 ± 0.0 (1.2)	10.8 ± 0.0 (1.5)	7.3 ± 0.0 (1.0)
creating/clearing 1k rows (5 cycles) Memory usage after creating and clearing 1000 rows 5 times	4.7 ± 0.0 (1.2)	7.1 ± 0.0 (1.9)	3.8 ± 0.0 (1.0)

Figure 6.2.: Performance and memory footprints of angular, react and vue.js [Kra]

for most use cases[Rai+]. Vue's ecosystem is, however, also becoming increasingly larger[Vue]. Angular's ecosystem seems to be the smallest, but that is to be expected as it comes with more features out of the box[Sta+]. All three frameworks have solutions to generate documents via server-side rendering, which is a requirement for the case study.

Finally, this work will use the popular (see fig.6.3) react framework, for its simpler

one-way data-flow, the more powerful approach of embedding HTML in JS (which vue also offers) and lastly, to capitalize on the large ecosystem and support available.

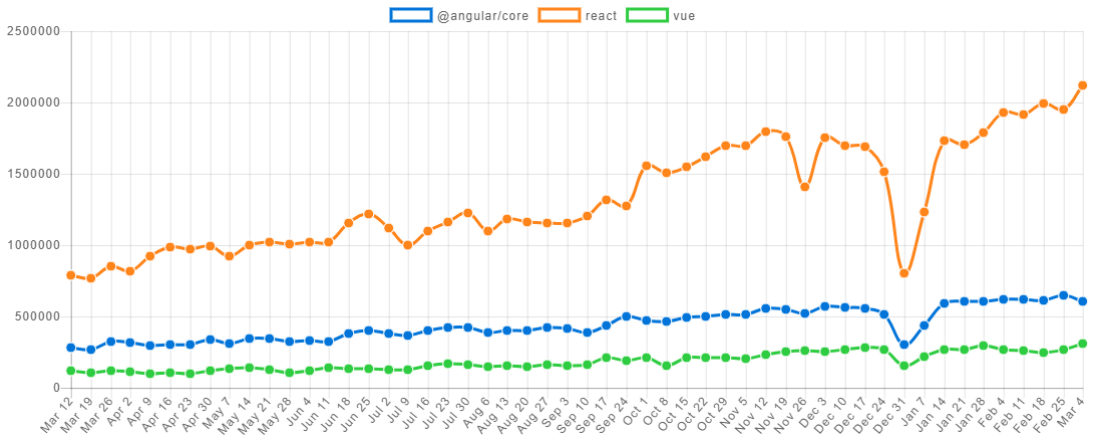

Figure 6.3.: Amount of NPM installs between angular(blue), react(orange) and vue(green) [Pot]

6.3. Static Site Generation

One hard requirement of the case study is the generation of static documents for minimal server-side dependencies to ensure continued hosting. Making data available for others to access is one of the most important traits any database should possess. This also applies to the CRC806 database, where data should stay available with as simple means as possible. While this approach does not work for all progressive web apps, it has advantages for any route where content rarely changes.
The advantages and disadvantages of server-side rendering and client-side rendering have been explored previously in chapter 2.3 (P. 10). A static site can potentially merge the advantages of both approaches while mostly removing the disadvantages. Static sites will be pre-rendered before the site is published, this minimizes the TTFB (*time till first byte*) as no processing needs to be done anymore when the site is accessed. Additionally, pre-rendering takes care of creating an initial DOM instead of dynamically rendering the DOM after the page and JavaScript bundle has been downloaded, processed and executed by the browser which lowers the time until a site becomes *viewable*. With this, a static site becomes viewable almost instantly, even while frontend frameworks may still be loading in the background. Once the framework is loaded, events can then be attached afterwards to the pre-rendered components to make the site *interactable* - React calls this process *hydration*. The *react-static* framework[Reac] illustrates this approach further (see fig.6.4). There are many static site generators available to chose from[Netb], but for this project, the generator should ideally support react out of the box to build on top of the previous frontend framework decision. Additionally, the static site generator should be JavaScript-based for two reasons: The existing JavaScript ecosystem (NPM) can be used to install/update the generator,

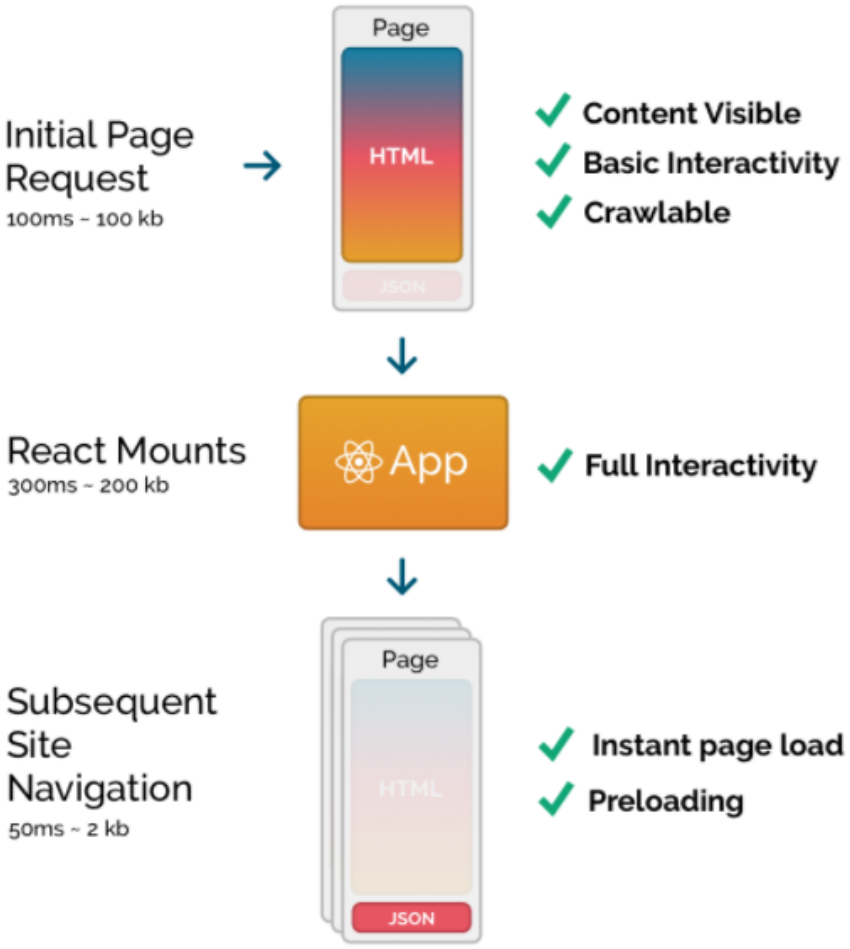

Figure 6.4.: Page is *viewable* upon the initial request, after react loads, the page becomes *interactable* [Rea18]

without introducing new environments that need additional setup. Also, less documented parts of the generator can be understood and edited, if necessary, using the same language (JavaScript) that is used throughout the project.
At this stage, there are only few frameworks available, most notable are Gatsby, React Static and Phenomic among very few less popular ones. Phenomic has been dismissed for now, as it is still in an alpha state at the time of this writing. Gatsby seemed to be an obvious choice as it went to many iterations and is used by many sites - among them even reactjs.org itself. One specific trait of Gatsby is its usage of GraphQL to query data from external sources, which is in turn used to populate pages with their respective contents. GraphQL is a querying language for APIs with one of the key feature being that specifically only the requested data will be delivered. This is very similar to *projections* in SQL, where only the requested fields are returned. With this, GraphQL presents intriguing benefits for load and processing time, but it also introduces an additional layer into the stack, where data has to be exported from the legacy

system, imported into a GraphQL endpoint which makes the data available, yet again, to be imported into the actual static site generator (Gatsby).
This overhead motivated to take a closer look into *React Static*, a site generator that has been build with these among other issues in mind[Lin17]. React static is less opinionated in when it comes to the way in which data is supplied. Any technology can be used to retrieve the data (e.g. REST, GraphQL, SQL, local files) during the initial build time, which will then be passed down to the underlying routes. Automatic data splitting ensures ideal bundling of this data. E.g. if specific data is used on multiple routes, an additional bundle will be created that is loaded on those routes, if data is used globally, it will be moved to the global bundle that is used by all routes. This module bundling is done with the help of *webpack* which other static site generators like Gatsby also rely on. All routes are defined in a configuration file (*static.config.js*). Generating routes dynamically from data is as easy as retrieving them with a technology of choice, iterating through them and applying a route configuration (this is usually done via *Array.prototype.map()*). And example of this can be seen in figure 6.5, where a static route (*/datasets*) is defined first, a page where a list of all datasets is displayed. Followed by that are many detail pages for each single dataset. Those routes (*/dataset/show/<datasetname>*) need to be dynamically generated by iterating through the available data. Each route needs to be supplied with one React component that determines how that route is rendered. These React components may be use more components as their children. When building the site for production, React Static will actually generate many folders that exactly match the provided paths in the *static.config.js*. For valid routes, an index.html will be placed in the folder which contains the pre-rendered DOM for that route as well as the bootstrapping JavaScript that will load React to enable client-side routing. This means that from that point on, only route specific data will be loaded and then replace the current route's content instead of reloading the whole page. This is comparable to the *App Shell model*, where only the contents are loaded via AJAX while the *skeleton* remains. If JavaScript is not enabled, the site will navigate to the other generated routes as normal, ditching the current route and loading the next route from scratch.
React static proves to be an appropriate tool for this case study, not only because it seems to work well for the most part, but also because it addresses many explicit and implicit requirements of a progressive web app. It is build with the PRPL pattern in mind (this is also true for Gatsby): Webpack uses code splitting and tree-shaking to ensure that only critical resources are loaded on a given route. All routes are pre-rendered to make them instantly viewable even before React loads. React Static uses its own *Link* component to create links to other pages, unless specified differently, data for these routes will be pre-cached after the current route has finished loading to speed up further navigation. Finally, pre-caching can be turned off for data-heavy routes, or alternatively, data can be lazy loaded via AJAX for specific routes. These points allow React Static to hit all the check boxes suggested by the PRPL pattern. The usage of webpack and its devserver also enables a positive developer experience as it features *Hot Reloading*. The webpack devserver will watch for any changes that may happen within the project files, if changes are detected, webpack will retranspile the affected files and its dependencies to then only reload the affected component on a page. This relieves the developer from having to refresh the page manually on each change, but

```
// importing data from file system,
// but any technology can be used for retrieval
import datasets from
    './src/tx_unikoelncrcdata_domain_model_dataset.json'

export default {
    getRoutes: async () => {
      return [
          {
              path: '/datasets',
              component: 'src/containers/Datasets',
              getData: () => ({
                  datasets: datasets,
              })
          },
          {
              path: '/dataset',
              children: datasets.map(dataset => ({
                  path: `/show/${dataset.name}`,
                  component: 'src/containers/Dataset',
                  getData: () => ({
                      dataset,
                  }),
              })),
          }
      ]
    }
}
```

Figure 6.5.: Simplified section of static.config.js, where data is passed down to routes

also from having to reinitialize the state the UI has been before.
There are still some caveats to have in mind when using static site generation with React. When developing for the front-end, libraries may be used that need to access the DOM or other browser-specific objects. These libraries are usually available via NPM, however if they are to be imported like usual (at the top of a file), this will cause errors during site generation. This is because React Static generates the routes server-side via node.js. At that stage, no browser exists and accessing browser specific objects like *document* or *window* will cause the build process to fail. There are multiple ways to solve this. One way could be stubbing the missing objects during the build process, but that may cause subsequent faults as the stubbed methods may not return the expected data. The most success has been had by using *require* instead of *import* and only using it in a scope that will be executed when run in a browser. Import only works on ES6 modules which, on their side, use export to expose certain functionalities (see chapter 1, P.XIV). Require, is a node.js-specific way to load modules which do not

quite follow the exact syntax of ES6 modules but have a similar approach. The require-specific module syntax is often used by pre-build JavaScript libraries that are meant to be included directly in a production site without further processing by a transpiler or similar. In the case of React Static, external libraries which access browser-specific objects need to be pre-built as the site generator will not be able to build them by itself. React Static speaks of *node-safe* code[Rea18], which describes code that is able to fully run in a headless (a.k.a. no browser involved) node.js environment. When using React components, there are two scopes that have proven to only be executed in a browser-environment: The *componentDidMount()*-hook and the *render()*-function. As the render function is executed every time the state significantly changes, it does not pose to be a appropriate place to import a library - a process that is only meant to happen initially. This is where *componentDidMount()* proves to be useful as it is only executed by React once the component did successfully load and before any render function (that may depend on a specific library being available) executes. It is important to have this issue of using external, browser-dependent libraries in mind as it is very possible to spend a lot time on building and polishing the look and feel of a site, using the provided devserver, only to realize near the end that the build process fails and deployment is not possible. It is recommended to test the build process every time a new library is introduced to make sure everything works as expected. Another issue to keep in mind is that React Static is still in heavily developed, which can result in breaking changes when updating this dependency. On rare occasions updates have introduced breaking bugs, but those have been resolved very quickly on most cases. Additionally it is always possible to downgrade to an earlier version from NPM if needed.

6.4. Importing the Data

The current CRC806 Database uses data from many different sources, this data is often intertwined which was one of the main causes of long load times as all the necessary data for a route has to collected upfront from different systems which all introduce their own latencies. The purpose of this chapter is finding a strategy to import all of the existing data in a useful way. However, first a closer look at the data at hand to get a better idea about their specific needs.

Publications and Datasets A publication or dataset describe the same entity in the underlying database which are merely *tagged* differently, it is even possible for this entity to be listed as both, a publication *and* a dataset. This is the primary data that has been provided by the users of the database and each entry presents a specific piece of work that scientists want to share with others. Publications and datasets contain many different types of values, from basic textual or numeric data to temporal and spatial data, as well as related data. Temporal data refers to either an specific event or interval in time, while spatial data describes either a point or a rectangular area of the world. Related data possibly contains other publications/datasets, but can also refer to Layers or Sites, which will be detailed in the next paragraphs. This data is mostly

hosted using a CKAN database[CKA], but due to technical limitations, relations to other data is additionally saved using a MariaDB[Mar], a MySQL-fork.
In order to export the data, an export interface has been implemented at the legacy system. This way we can reuse the mechanism that join the data from their respective sources and refine them. Refining contains further processing like correctly formatting Authors according to bibtex standard or building a citation string with consistent rules across all publications and datasets. Reusing the mechanics in place seems to be the best way to retrieve results that are consistent with the current data of the legacy system. A remaining challenge will be dataset resources as those vary in accessibility. A possible strategy could contain using the post-build configuration of React Static, which allows to automatically do additional tasks if building was successful. Such a task could decide which resources are meant to be publicly available using the provided data and then only copy those to a static folder.

Layers Layers present *georeferenced raster data* that is rendered on an interactive world map (see fig.6.6). In order to correctly retrieve imagery for each zoom level, a WMS (*Web Map Service*)[OGC] is queried via REST. The WMS standard offers an optional *getLegendGraphics* function which allows the web application to display a legend in the bottom right corner if available. Examples of raster data may include elevation maps and climate data among others. The WMS endpoint is provided by

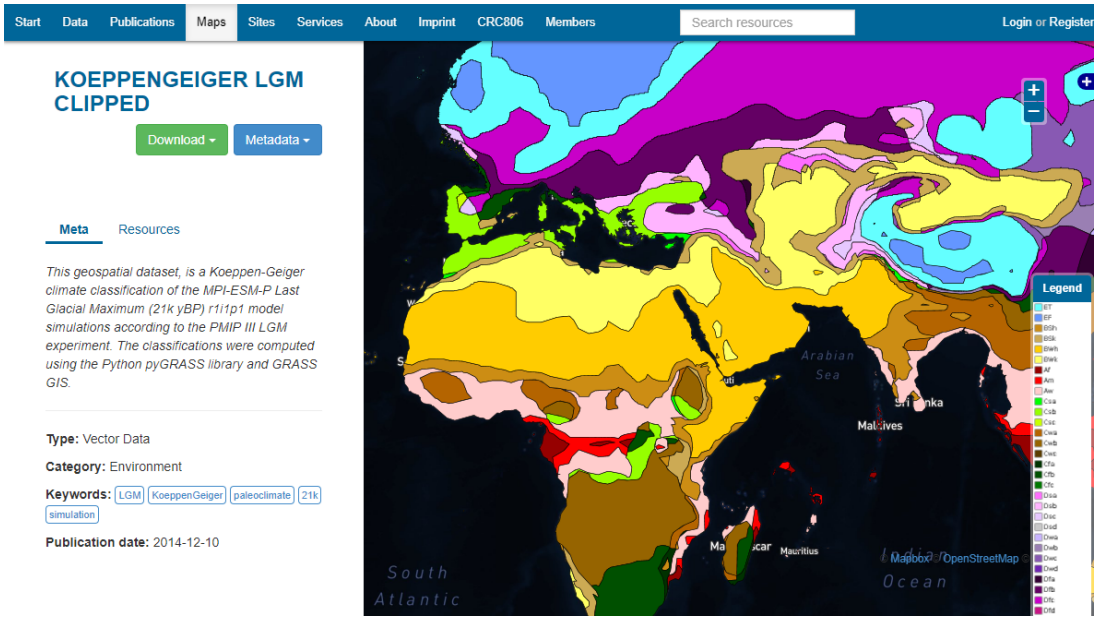

Figure 6.6.: Detail view of a layer in the current web application

a GeoNode[Geo] instance that is currently hosted by the university of cologne. Since GeoNode needs to be queried interactively, data can not simply be exported as JSON. One could attempt to fetch all generated imagery for all zoom layers which could then be reused for the generated web application, but this brings more challenges. The

frontend library that is used for map visualization (*Leaflet*[Aga+]) expects WMS (*Web Map Service*), TMS (*tile map service*) endpoints or similar, which it will then query autonomously. If such an endpoint fails to exists, heavy customizations need to be done in order to reuse manually exported data instead. As a consequence, *Layers* will not be available in the new web application, as relieving the data from server-side dependencies will leave the scope of this work.

Sites Sites pose a collection of research sites where publications, datasets or Layers may originate from. Sites contain mostly meta data that describes name, purpose and images of a specific research site. Additionally a spatial point is provided in order to show the location of the research site on an interactive map (see fig.6.7).
As research sites are completely contained within a MariaDB instance and no further joining of different sources is required, sites can be dumped directly from MariaDB without requiring a specific export interface to be implemented on the legacy system.

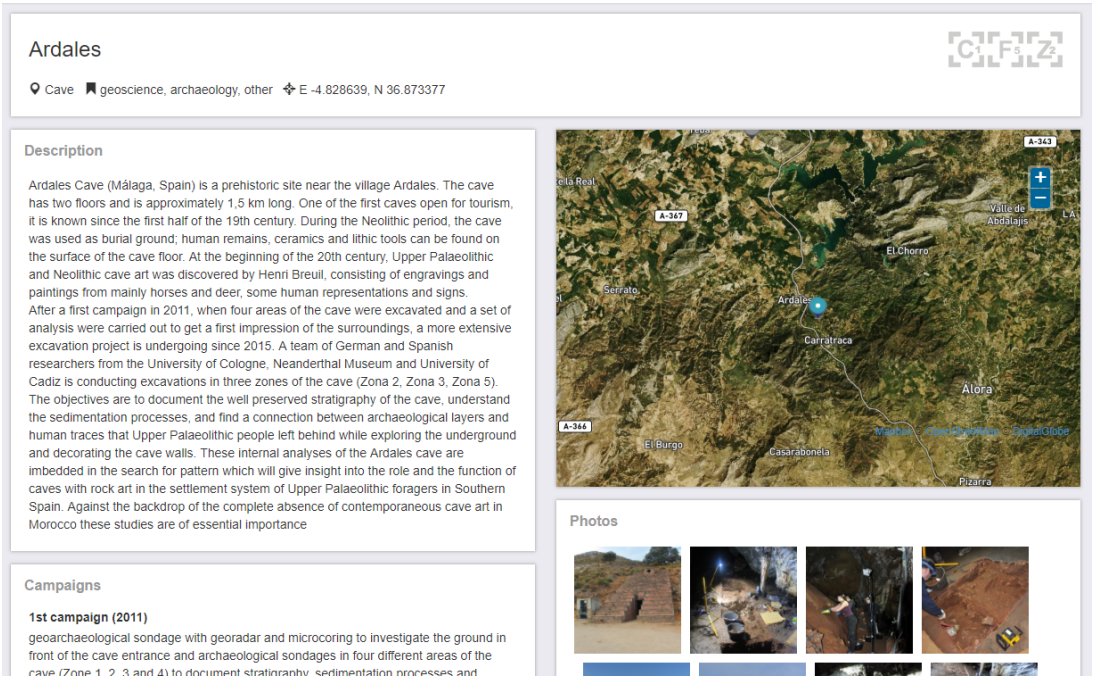

Figure 6.7.: Detail view of a specific research sites in the current web application

Most strategies come down to finding the point where all required data is joined and complete, followed by exporting them. When talking about exporting, the format used is JSON as this can be imported directly into JavaScript projects without any additional libraries that require slow parsing steps. Instead *JSON.parse()* can be used which is native to JavaScript and thus executes faster than manual implementations. The legacy system is PHP-based which also has fully functional, native support for the JSON format and allows easy exporting of arrays and objects using the *json_ encode()*-function. Finally, PHPMyAdmin - an administration backend for MySQL or MariaDB databases - also allows directly exporting SQL tables using the JSON format (see fig.6.8).

This is very similar to the JAMstack architecture, which describes itself as a "Modern web development architecture based on client-side JavaScript, reusable APIs, and prebuilt Markup"[jam]. JAMstack is a technology-agnostic architecture, that requires three key-factors[jam]:

(J)avaScript Any dynamic programming during the request/response cycle is handled by JavaScript, running entirely on the client. This could be any frontend framework, library, or even vanilla JavaScript.

(A)PIs All server-side processes or database actions are abstracted into reusable APIs, accessed over HTTP with JavaScript. These can be custom-built or leverage third-party services.

(M)arkup Templated markup should be prebuilt at deploy time, usually using a site generator for content sites, or a build tool for web apps.

As JAMStack is technology-agnostic, it is not specific to progressive web apps but can be applied to any kind of web application. Differences from JAMStack to the presented approach apply to APIs. Since server-side dependencies need to be removed completely, there will not be any API that can be queried directly. However, with the combined exports mentioned, multiple JSON documents will be generated that mimic possible API responses from various endpoints. As this data will be static and does not need to change while the application is deployed, the data will directly feed into the generation of the prebuilt documents instead of retrieving the data from scratch on each page load. This will ensure that contents are available instantly upon loading the HTML document, even before JavaScript files are downloaded and processed.

6.5. Making a Responsive Layout

One major benefit of web applications in general is the ability to address virtually any device with a web browser. In order to capitalize on that opportunity, developers need to take care of varying screen sizes, input methods and network availability. While the issue of network availability is mostly handled with the usage of caching strategies (see the next chapter), a responsive layout should keep the first two in mind. Originally, the first bare-bones, plain-HTML websites have all been responsive as no layout rules (Cascading Style Sheets) have been in place that limited how far content elements span. However, with no or minimal CSS rules in place, there is barely any conscious design to address usability concerns or corporate identity.

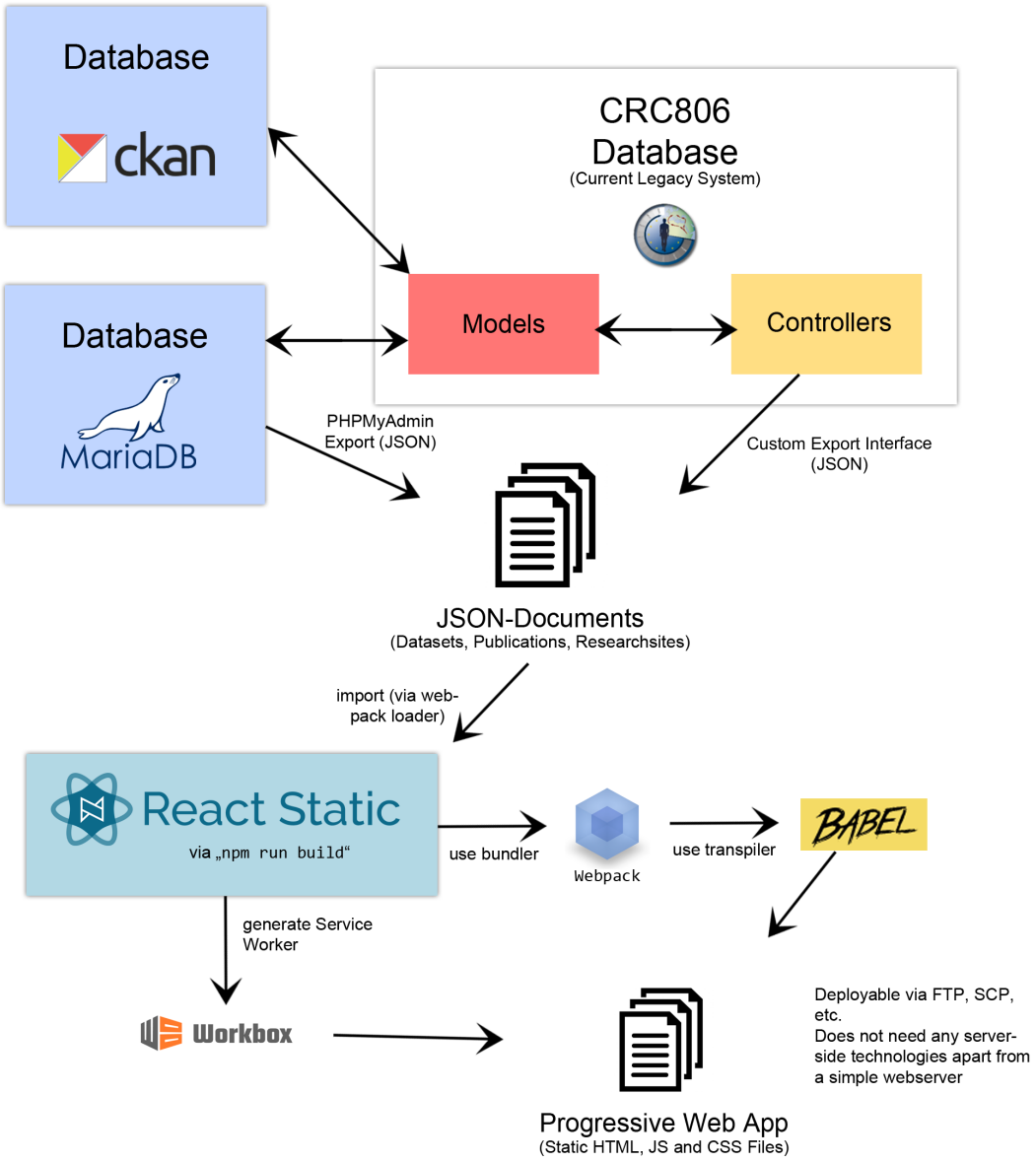

Figure 6.8.: Overview for the technologies used and how they interact with each other in this case study example. Images belong to [CKA], [Mar], [Reac], [web], [Bab] and [Con] respectively.

In order to accommodate varying screen sizes, modern web applications often use *responsive grids* where columns may break into rows in order to ensure readability of content elements on smaller screens (see fig.6.9). This is done by using *media queries*, which allow to specify CSS rules for certain *media types* and *media features*[W3Sa]. However, in most cases the media type is omitted (defaults to *all*) and for the purpose

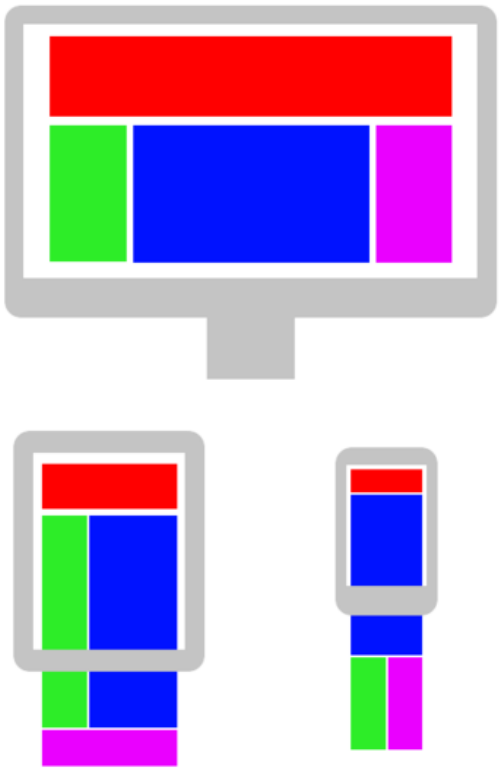

Figure 6.9.: A responsive grid that automatically falls back on varying screen sizes. Source: [Wik18]

of responsive grids, only the media features *min-width* and *max-width* are usually taken into consideration. For touch-screens, it is important that the pointer-event *hover* can not be used for important UI components like e.g. a navigation with child elements that only becomes visible on hover. For those cases, alternatives are needed, like opening a sub-navigation via click (touching an element briefly will issue a click-event).
For the case-study, the CSS library *UI-Kit*[UIk] has been used as it comes with a grid system among many other UI components. Reusing those components makes sure that the web application has a consistent look for the most part. There are many other UI frameworks to chose from, and while there are objective criteria like bundle size to ensure quick load times, most of it comes down to taste. Bootstrap as an example is very popular, but unless the components receive custom theming, there is the danger that different bootstrap sites may look similar. This can happen with many UI Frameworks, however some others may only focus on very basic aspects that are less obvious to observe. Example are frameworks like skeleton[Ske] or simplegrid[Col+] which contain only very basic styling combined with a responsive grid system.
Alternatively, since the CSS flexbox became mature enough to be used in production[Iri18], it is absolutely viable to not use a CSS library to begin with, as flexbox

combined with media queries allow for an easy and flexible way to build a custom grid system.

6.6. Implementing Caching Strategies

Caching strategies are a way to deal with unreliable or missing network connections. This is not to be confused with the usual browser cache as it will usually only cache certain assets, not the root document that enables the initial display of a web site to begin with. As discussed in chapter 3.1 (P.14), service workers are the only way for a web application to exist outside the browser tab. More importantly, the service worker is able to cache files, intercept any *fetch*-event from the browser and respond with data from that same cache instead.
To chose a proper caching strategy, an observation needs to be made which assets or routes need to be up to date at all times, which assets merely need to be updated eventually, and which assets are unlikely to ever change. The case study describes a read-only database system that is meant to archive the data that is currently in the legacy system for an undetermined time. Once the switch is made, it is very unlikely for the data to ever change again, which is why a more aggressive caching approach can be chosen. The service worker chapter mentions the strategies "*network or cache*", "*cache only*", "*cache and update*", "*cache, update and refresh*" and "*embedded fallback*". With currentness of the data being less of a concern, "*cache only*" and "*cache and update*" are appropriate candidates for a caching strategy. Both will , once the service worker did cache successfully, offer almost instant loading of resources, which is one major gain of using caching strategies to begin with. However, in order to be safe, if changes do happen, the strategy "*cache and update*" is preferred for this case study. Response times will still be near instant, but if a resource does become outdated, it will be up to date again on the next visit.

Implementing a caching strategy from scratch can prove to be a big challenge, while an incorrect implementation may also cause a state where the web application is unable to update itself anymore until the service worker is unregistered[Pop17]. Unregistering a service worker is not a simple task for average users as it often involves opening the developer tools of the browser in use which deviates strongly from the activities involved when normally using the web. To avoid those issues, it is recommended to use additional tools to implement caching strategies. At this point in time *workbox* seems to be the only available tool for this task[Con]. Workbox allows to assign a caching strategy on multiple resources at once using a wildcard syntax. It is also possible to use multiple strategies for different resources at once. There is a *workbox-cli* to automatically generate a basic service worker as a basis to start working with. Alternatively, workbox can be integrated into webpack in order to automatically generate a service worker with the bundles in mind that are generated by webpack. Unfortunately any attempts to use workbox this way have been unsuccessful. For this implementation, the workbox-cli has been used instead. By extending the "*build*" run script of npm, a new service worker is generated on each new build. This is necessary as webpack outputs assets that contain hashes in their names, which change on each build. By supplying

a configuration file, workbox-cli will automatically discover all assets in the webpack output folder and pre-cache these files. An initial configuration file can be generated by calling "*workbox-cli wizard*", which will start a short wizard with helpful defaults. Pre-caching will attempt to retrieve resources while network load is low, in order to not slow down load times of critical resources for the current route. The way pre-caching works in workbox ensures that new revisions of an asset will automatically be picked up and cached, so if any specific file were to change, generating a new service worker will automatically take care of updating the cache[Wor]. There have been issues where a new service worker would not activate and instead remain in *waiting* status indefinitely. This has been resolved by setting the *skipWaiting* variable inside the workbox configuration file.

Part III.

Evaluation

7. Evaluation of Progressive Web Applications

This chapter will attempt to evaluate different aspects of progressive web applications. First, a closer look will be taken on the requirements stated in chapter 5.2, in order to see which requirements were easy or challenging to fulfill and which requirements could not be fulfilled at all. Second, an assessment about current practices will be made, factoring in the experiences that have been made throughout the implementation phase of the case study. Followed by a short analysis on web applications in general in order to be able to judge their viability compared to native applications. The goal of this chapter is to analyze and evaluate on the aspects that have been outlined up to this part to prepare for the final conclusion in chapter 8.

7.1. Requirements Coverage

This chapter will revisit the requirements identified in chapter 5.2 in order to gauge which practices have been helpful in achieving those requirements, and which requirements remained challenging. For this, basic, performance and excitement factors will be checked to determine the exact coverage of the requirements in each category. Once the coverage has been identified, requirements that have been proven to be notably easy or challenging to implement using the practices mentioned throughout this work, will be elaborated.

7.1.1. Basic Factors

Requirement	Description	PWA-Relevant	Covered
REQ001	The web app shall provide users with the ability to view datasets and publications with the respective metadata	No	Yes
REQ002	The web app shall provide users with the ability to search for datasets and publications	No	Yes
REQ003	The web app shall provide users with the ability to list and filter existing datasets and publications	No	Yes
REQ004	The web app will inherit the URLs from the current website in order to keep existing links to the site intact	No	Yes

REQ005	The web app shall be able to function with a web server (e.g. apache, NGINX) as the only dependency	No	Yes
REQ006	The web app shall be able to display datasets and publication data without requiring to load JavaScript assets	No	Yes
REQ007	The web app shall provide users with the ability to view research sites with their metadata	No	Yes
REQ008	The web app shall display the location of a research site on an interactive map	No	Yes
REQ009	The web app shall provide users with the ability to search for research sites using keywords	No	No
REQ010	The web app shall provide users with the ability to list and filter existing research sites	No	No
REQ011	The web app shall provide users with the ability to view map layers with their metadata	No	No
REQ012	The web app shall provide users with the ability to search for map layers using keywords	No	No
REQ013	The web app shall provide users with the ability to list and filter existing map layers	No	No
REQ014	The web app shall provide users with an interactive map to explore all existing research sites and datasets/publications that contain location data	No	Yes
REQ015	The web app shall be served over HTTPS	Yes	No
REQ016	The web app shall be able to provide mobile users with a responsive design	Yes	Yes
REQ017	When accessed offline, the web app shall be able to respond with a HTTP 200 status code, presenting some content	Yes	Yes
REQ018	The web app shall provide users with the ability to add the web app to their home screen	Yes	Yes
REQ019	The web app shall be able to become interactive under ten seconds on a simulated 3G network	Yes	Yes
REQ020	The web app shall work in current versions of Chrome, Edge, Firefox and Safari	Yes	Yes

REQ021	The web app shall be able to provide page transitions in order to increase perceived performance	Yes	No
REQ022	The web app shall be able to provide a unique URL for each individual page	Yes	Yes

The total coverage of basic factors is 68.2% (15/22).
The total coverage of PWA-relevant basic factors is 75% (6/8).

The challenging part about making all previous data available (REQ001) lies mostly within implementing an export endpoint within the legacy system. With the data available, it is rather easy to generate all the required routes, using react-static. By generating folders for all possible routes, react-static ensures that there will be a static *index.html* waiting behind any route, without any server-side routing required (REQ005). With disabled JavaScript, client-side routing through react will not work, but since all links point to static documents, routes will still resolve (REQ006). The performance gain of replacing contents via AJAX will be lost though, as instead, the whole page will reload. The global search (REQ002) has been implemented using a Web Worker so that searching through large amounts of data will not reflect on the UI thread. However with the amount of data in this case study, there have not been any noticeable performance gains, as both the legacy search as well as the new search perform very fast. A reoccurring use case across the web application is the need to display location data on a map (REQ007, REQ008, REQ011). This can either be a point or a rectangular area. To avoid rewriting functionality over and over, a react component has been developed to display a map via leaflet[Aga+]. This component has been reused to display the spatial information for datasets or displaying the location of a research site. Additionally, there is an advanced use case for displaying multiple points/areas on a large map (REQ014). Goal of that map is to explore all existing data with spatial data attached. When multiple points/areas are in close proximity, they shall be clustered as one point with a visual distinction to regular points. To implement this cluster-map, a new react component has been created that extends the existing map component. This came with a big caveat, as react does not encourage inheritance between components, instead, react recommends composition over inheritance[Reab]. As a consequence, clustering needed to be supported by the core map component, while the clustermap component merely pre-setted the required properties, while also adding a few more HTML elements to display information about a selected cluster. Inheritance would not work well either way, as the whole component structure is defined in the render-method of a component. Any component inheriting from a parent component would either have to completely accept the structure or override the render method and re-implement the structure from scratch. This creates the possible danger of react components having to grow very large, just to support multiple functionalities, whereas a parent component merely presets those components to fulfill a more specific role.
Support for listing/viewing map layers (REQ011, REQ012, REQ013) had been dropped completely as there is no feasible way to simulate an geonode endpoint without server-side dependencies.

Basic support for mobile devices (REQ016) has been easy to achieve by using a grid system chapter (see 6.5 for more details).
By only bundling required assets via webpack, the web application loads within eight seconds (REQ019) from scratch (emptied cache) over a simulated 3G network (simulated via chrome devtools).

7.1.2. Performance Factors

Requirement	Description	PWA-Relevant	Covered
REQ023	When used on a mobile device, the web app should be able to stay functional and completely usable	Yes	Yes
REQ024	The documentation will provide administrators with a workflow to migrate from the existing website to the web app	No	No
REQ025	The web app should embed schema.org metadata in order to improve the appearance in search engines	Yes	No
REQ026	The web app should use the history API instead of fragment identifiers	Yes	Yes
REQ027	The web app shall be able to become interactive under five seconds on a simulated 3G network	Yes	No
REQ028	The web app shall use a *cache and update* caching-strategy	Yes	Yes

The total coverage of performance factors is 50% (3/6).
The total coverage of PWA-relevant performance factors is 60% (3/5).

The web app uses mostly relative sizes and avoids absolute positioning for the most part. When absolute positioning is used (e.g. for the search) the affected element pans upon the whole screen, which works well for desktop as well as mobile screens (REQ023). Scrollbars are added automatically through the CSS *overflow* property to ensure that contents are reachable even if they do not fit the screen bounds.
React-Static automatically utilizes the history API to simulate static URL paths (REQ026) without any manual implementation steps needed.
On first load, the web app repeatedly takes around eight seconds to load the initial resources on a simulated 3G network, with the global JavaScript bundle being the worst offender as it takes seven seconds on average. Upon further inspecting the bundle, it becomes apparent that react-static embeds meta data of all datasets into the main bundle, which makes up most of its file size. It is not clear why react-static promotes that data to the main bundle, as the data is only supplied to certain routes. This may prove to become a problem for sites with larger amounts of data, more investigation is

needed to come to a full conclusion. However, for now this fails to fulfill the five-second mark requested in REQ027.

7.1.3. Excitement Factors

Requirement	Description	PWA-Relevant	Covered
REQ029	After the first visit, the web app should be able to provide the user with basic metadata for datasets, publications, research sites and map layers without requiring a working network connection	No	Yes
REQ030	The web app should present contents in a way that elements do not jump as the page loads	Yes	Yes
REQ031	When the user goes back to a previous page containing a list, the web app should be able to restore the scroll position of that list	Yes	No
REQ032	When an input is selected that opens an onscreen keyboard, the web app should ensure that the input will not be covered b the keyboard or another element	Yes	Yes
REQ033	The web app shall inform the user when accessed offline	Yes	Yes

The total coverage of excitement factors is 80% (4/5).
The total coverage of PWA-relevant excitement factors is 75% (3/4).

With the whole webapp being prerendered, DOM contents are available as soon as the basic HTML document is loaded, which completely avoids the issue of lazy-loaded content jumping into appearance (REQ030). Additionally, most images and icons are used as a background image to HTML block elements with fixed dimensions. That way the browser will reserve the required space for an image in the document flow even before the image is loaded, because the dimensions are already known.

The total coverage of all requirements is 66.6% (22/33).
The total coverage of all PWA-relevant requirements is 70.6% (12/17).

Many of the case-study specific requirements have been dropped because of either technical limitations (REQ011, REQ012, REQ013) or time constraints. Some of the PWA-specific requirements that have not been fulfilled were due to not having a complete understanding about some specifics of the frameworks in use. This mostly affected missing page transitions (REQ021), a too large bundle to achieve initial load times under five seconds on a 3G network (REQ027) and not restoring the scroll position from

the previous page properly (REQ031). Pursuing these issues has been discontinued due to time constrains as they do not impact core PWA principles in a severe way.

7.2. Assessment of current Practices

This section will look at different practices that have been applied during the development of the prototype for the case study. These practices mostly stem from the Google Checklist for PWAs[Goo17c], JAMStack *best practices*[jam] and own observations made during the development process, as well as from looking into other web development projects.

Using a Secure Connection HTTPS offers a transport encryption between the browser and a web server and grants three major gains: *Confidentiality*, *data integrity* and *authenticity*. *Confidentiality* becomes important when sensitive information is transmitted over the network that must not be interpretable by anyone that may intercepts the network connection (also known as *man-in-the-middle*) [Isa18]. While the input of sensitive data is not a use case in the case study, the latter two points still offer benefits: *Data integrity* ensures that data has not been manipulated during the transmission from the webserver to the browser, while *authenticity* attempts to prove that the webserver that responds to a request is indeed the intended recipient/sender. The disadvantage is a slightly longer connection initialization, as at first an asymmetrical encryption will be used to exchange a key that is then used to symmetrically encrypt the actual HTTP responses. This combines the increased security of asymmetric encryption with the higher performance of symmetrical encryption. Additionally to the mentioned security reasons, certain technologies like *service workers* require an HTTPS connection while browsers start to indicate increasingly prominent warnings (see fig.7.1) to the user when visiting an unencrypted website.

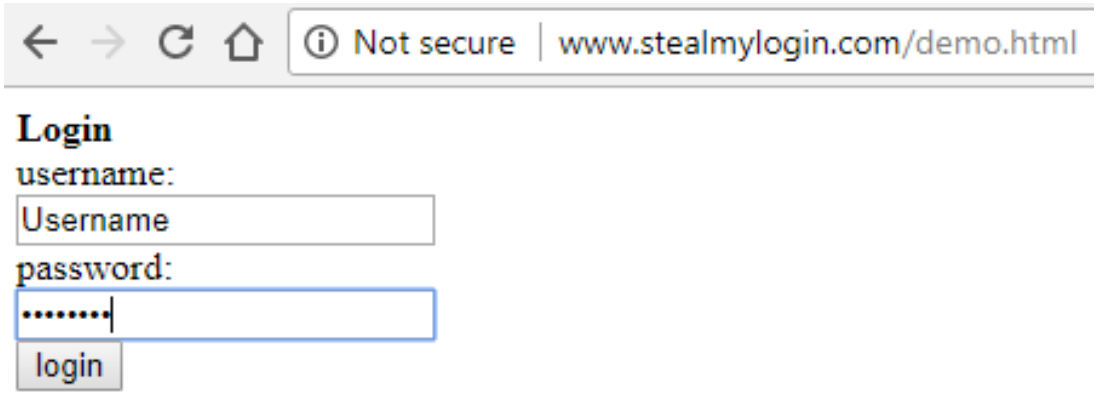

Figure 7.1.: Google Chrome browser showing a warning on a non-HTTPS page with a password input

Adopting a Responsive Design Mobile devices are making a huge part of todays web consumption and should definitely be accounted for. Statcounter reports mobile devices currently being globally the most popular device to browse the web with above 50%[sta18], however this does vary a lot with the type of industry that the website addresses, seen in the table below (data from [Eng17]).

Industry	% Mobile Traffic
Adult	74.90%
Food and Drink	65.00%
Beauty and Fitness	63.60%
People and Society	74.90%
Home and Garden	61.00%
Internet and Telecom	60.40%
Health	59.70%
Pets and Animals	59.50%
Sports	59.20%
Autos and Vehicles	58.30%
Business and Industry	57.40%
Shopping	56.90%
Books and Literature	54.60%
Reference	54.20%
Recreation and Hobbies	53.80%
Law and Government	52.80%
News and Media	50.70%
Travel	50.40%
Career and Education	47.60%
Art and Entertainment	45.00%
Science	42.90%
Computer and Electronics	42.60%
Games	40.40%
Finance	39.60%

Unless a web application is purely accessed on desktops (e.g. corporate intranet applications), adopting a responsive design is a good idea. Depending on the available resources, this can scale from merely utilizing of a responsive grid system to elaborate loading speed optimizations and specific UI components that capitalize on the availability of touch events and screen orientation.

Enabling an Offline Experience This allows web applications to match availability similar to native applications. While it is obvious that network dependent tasks cannot function offline, there should at least be some functionality or a short notice explaining that the user is currently offline. This can only work by utilizing the *service worker* and more exactly its caching API. However, the service worker life cycle is non-trivial and may not always function as expected. During the implementation phase for the case study, there have often been cases where a new service worker would not activate

or old files would keep getting served. While those issues can be solved, it is rather easy to fall in those traps for developers that do not have a good grasp on the service worker life cycle. This may lead to a flawed caching strategy staying around longer than intended or even cause a web application to wrongfully cache resources for an undetermined time while currentness may be a concern.

Ensuring Cross-Browser Compatibility This has been greatly simplified with the introduction of *transpiling*. Current transpilers replace modern ECMAScript with poly-fills if necessary to ensure compatibility with most browsers. Browserslist queries[Bro] allow the targeting of browsers in a linguistically clear way, e.g. "last 2 versions" for the last two major browser versions, or "$> 2\%$" for browser versions which usage is above two percent. This makes it very transparent for developers to see which browser will be supported after the transpiling process. Developers can use current ECMAScript functionalities now, without having to wait for all browsers to support those. However, this also means that the generated code may get bloated with poly-fills that could be avoided by adopting older ECMAScript feature sets. Additionally, debugging of transpiled code may prove to be harder unless additional source maps are provided which allow mapping from transpiled code to the more readable source code.

Emphasizing Content Semantics and Structures Web applications usually try to present their contents in a way that UI elements are easily visually recognizable. Beyond that visual presentation, applications need to be optimized to be read by other machines too. These are usually screen readers for users with a visual impairment or indexing services of search engines. Improving the readability for those instances can help in reaching more users in both cases. One basic approach lies within using semantically correct HTML5 markup. Before the advent of HTML5, *div* and *span* HTML elements have been used ambiguously anytime contents needed to be wrapped in either a block or inline element, which is not helped by the fact that CSS allows to overwrite whether an element is actually displayed as a block or inline element. HTML5 offers many new elements which often behave like a *div* while being more descriptive about their contents. This makes it easier for machines to guess with what kind of content is being dealt with, e.g. navigational contents can be wrapped in a *nav* element, articles in an *article* element, sidebars in an *aside* element to only name a few[W3Sb]. However, with rich web applications, not all HTML elements outline meaningful content, instead there will be many elements that may function as a toggle, a tab or a progressbar without additional text that contains semantic significance. ARIA[KGB18] solves this by adding additional attributes to describe elements further. Most importantly the *role* attribute, which declares when an HTML element is of functional purpose instead of holding content. Finally, structured data allows for easier indexing by search engines and can improve how easy a web application can be found by potential users. There are multiple ways to achieve this, namely JSON-LD, Microdata and RDFa, whereas google highly recommends JSON-LD[Goo17a]. Structured data works by describing meta information about the current page using the vocabulary of schema.org.
The idea of semantic markup does seem to clash a little with the usage of certain frontend frameworks. This is mostly the case with frameworks that aim to add respon-

siveness and specific UI components to the application. These frameworks cannot know any specifics about a web application, which is why they come with very generic CSS classes to be able to adopt many use cases. UI elements often have to be wrapped in additional container-elements which, while required by the UI framework, do not add any semantic meaning to the document (see fig.7.2).

```
<div class="container">
  <div class="row">
    <div class="col-sm">
      One of three columns
    </div>
    <div class="col-sm">
      One of three columns
    </div>
    <div class="col-sm">
      One of three columns
    </div>
  </div>
</div>
```

Figure 7.2.: Minimum markup required for a three-column grid in bootstrap [Boo]

Adopt a Service-Based Architecture Server-side rendered applications often present tight coupling between the frontend and backend. As a consequence, it can be very hard to replace specific technologies without having to reimplement large parts of a monolithic architecture. Supporting architectures like this over a long period can create situations where many resources need to be spend, managing and updating dependencies that may break certain functionalities. Alternatively, frontend and backend can be developed as independent projects that communicate with each other using APIs[jam]. A currently evolving trend is the advent of *headless content management systems*[Neta] which offer a UI for editors and content managers to create content that is then exposed via APIs to a web application. This approach has multiple gains: Data is decoupled from the application and while it can be managed conveniently in one place, the data can be accessed from multiple applications like web applications, mobile apps or as a B2B interface for business partners. The other benefit is how services can be easily replaced as long as the API remains the same. Having the option to adopt to new technologies quickly can be an asset to stay competitive as a business.

Adopt Pre-Rendering when appropriate Server-side rendering as well as client-side rendering can introduce certain delays before displaying meaningful content to the user. In the case of server-side rendered pages, this would be the processing time where the server may perform a set of operations until a response, containing the final HTML document, can be made. Client-side rendered pages have to retrieve, compile and execute JavaScript code before information can be displayed. It is important to

identify if currentness of a certain page is a concern or not. Pre-Rendered pages allow for reduced load times compared to both aforementioned approaches as no processing needs to be done anymore on server-side, while static content is already served and viewable by the time the document has been loaded by the browser. Only dynamic elements need to wait for JavaScript to execute in order to function. The downside of this approach is, how pre-rendered pages need to be manually rerendered and deployed whenever changes need to occur. To mitigate this, this may be automated by an headless CMS or a CI/CD pipeline. There may be cases where content needs to change the instant a user made input within a web application and where waiting for a pre-rendering process to generate and deploy changes is not appropriate. This is a case where the app shell model is more suitable. Pre-rendering can still be applied to the app shell, which rarely changes, while contents are dynamically fetched via JavaScript. To trick load time perception[Cam17], empty placeholders can be displayed, which are then later replaced with the actual contents, once those finished loading. While pre-rendering may use any form of internal data to generate pages, app shell style applications require some sort of service-oriented architecture in order to query endpoints to retrieve contents from. In general, if the app shell model is adopted, the lines between the app shell and actual contents may be blurred as a pre-rendered page may contain both already. If pre-caching is used (through service worker) it is not possible anymore to only pre-cache the app shell without the content.

Creating a Native-Like User Experience While initial load time may be one important factor for a good user experience, it is also important to retain a certain degree of responsiveness throughout the application life cycle. The goal is to reach a similar user experience as native applications in regards of performance and reliability, in order for web applications to become a viable option. While asynchronous tasks allow for non-blocking calls, it should not be used to transmit large chunks of data at once as this will still slow down the UI thread. Processing-intensive tasks should be moved to a worker-type (WebWorker or ServiceWorker) script in order to achieve true concurrency. If specific routes need large amounts of data, this data may be pre-loaded in preparation (e.g. using *<link rel="preload">*) on less data-heavy routes. The ability to select text should be disabled on interactive elements, as especially on mobile this can cause a disruptive behavior where consecutively pressing a button may select the button text and open a dialogue to copy the selected text. Images should use fixed dimensions so that the browser will reserve the required space in the document flow even before the image finished loading. Without fixed dimensions, a situation may arise where users try to read a text that is suddenly pushed out of view by an image that just finished loading. If exact image dimensions cannot be predicted, the image can alternatively used as a background to a fixed dimension container, with the CSS property *background-size* set to "*cover*". This will stretch the image to fill the container while keeping the correct aspect ratio, at the cost of possibly only displaying part of the image.

Improving Developing Experience In order to keep up with the demands of modern web applications, it becomes necessary to also address the needs of developers. There

are an overwhelming amount of tools, frameworks and libraries that aim to assist developers. The usage of package managers allow all developers involved in a project to have consistent dependencies, without having to add those dependencies into their version control (e.g. git, svn), which would cause unnecessarily large *commits* whenever a dependency is updated, added or removed. Transpilers give developers the freedom of choosing more feature-rich alternatives to HTML, JS and CSS while also ensuring that the resulting files are compatible with specific browser versions. Tools like the webpack devserver feature *hot module reloading* in order to create a faster feedback loop whenever changes to the code are made. The feedback loop can also be improved by adopting a *continuous integration* pipeline, where any code commits are automatically tested by a pre-defined process and warnings are send to the dev team if a test fails to succeed. This allows developers to receive warnings while they may still be in the mindset of that specific change, opposed to a much later date when the project may actually be deployed. In fact, many packages in the NPM registry feature a github page that shows the current build status (success or fail), test coverage or whether their dependencies are up to date.
Adding a large amount of tools does not come without a cost. Eric Clemmons talks about JavaScript fatigue[Cle15] where he describes how many decisions are needed to be made in order to set up an initial project, long before even a single line of code can be written. Dan Abramov, one of the React core authors, acknowledges this issue and suggests tool authors to find useful presets in order to minimize the cognitive load caused by those decisions[Abr17]. He also describes how authors of JavaScript tools are "*[...] a gatekeeper to the largest programming community in the world.*". Additionally to the complex tooling, many frameworks expect a very up-to-date understanding of ECMAScript and its ecosystem in order to build rich applications. This may require training current employees or hiring additional ones.

7.3. Costs

Browser Limitations Almost no operation system functionalities are exposed to a web application. A web application can not see tasks running on the system, and it has no "real" access to the filesystem. The HTML5 filesystem API[MDN18a] allows to reserve some space within the local file system to store information but there is no way to access actual files in the filesystem. This makes applications like media players or file managers mostly impossible. The only way around this, is by having the user manually supply files using the e.g. the HTML file input or creating a pane where files can be dropped on. However, this is still a read-only process and has to be initiated by the user every time the application is accessed. Additionally, Dan Dascalescu lists further limitations that can not currently be done by *progressive web apps* [Das18]:

- Accessing local contacts, calendar entries or browser bookmarks
- Setting alarms
- Telephony features, like sending/receiving SMS, calls or voice mails
- Access to specific hardware sensors

- System access, which contains aforementioned things like task management, but also modifying system settings or accessing log files.
- Becoming the default app for custom URL schemes, protocols or file types

Many of these limitations could be seen as a benefit as removing them would expose the devices in use to a variety of possible malware and other misuse. While this is also true for many native apps, this issue is more severe on web applications as they can be accessed by accident when browsing the web, without any setup or user confirmation required.

JavaScript Fatigue As mentioned earlier, the JavaScript ecosystem is heavily expanding which makes it increasingly harder to confidently make decisions about specific tools, frameworks or libraries at hand. Configuration of tools can take an significant amount of resources before the development process can even start[Cle15][Abr17].

Non-Trivial Service Worker Life Cycle It is easy to use a Service Worker wrongly, which can cause large problems like web applications unable to update themselves. Service Workers need to be used with caution[Pop17].

7.4. Benefits

Shareable and Linkable Web applications are very easy to share, as no large files need to be sent and validated, but instead a simple URL suffices[Rus15]. The concept of sharing URLs is well-established for many users, even more so on mobile operating systems where a *share*-functionality is embedded in many applications.

Device-Agnostic As a modern browser is the only hard-requirement, web applications run on a very broad amount of devices without requiring multiple code-bases to address multiple platforms. This makes web applications more resource-efficient to develop compared to native applications.

Instant Deployment No installation step is required when opening a web application. This trivializes deployment of new versions, as assets will be updated automatically, depending on the chose caching strategy. Deployment is especially easy compared to apps that reside in stores, where any update may need to be approved by the store provider to ensure that its policies are respected.

Approachable Web applications will automatically run, the moment they are accessed. Specific setup and configuration is not necessary, which is a benefit if a user is not comfortable or privileged (e.g. due to company policies) to set-up new software.

Low Storage Requirements As the browser acts as the primary platform for progressive web apps to run on, most heavy-lifting is already done there and does not need to be implemented by the software engineer. This results in much smaller file sizes

compared to native applications. Twitter demonstrates this by comparing their new progressive web app (600KB in size) with their native android application (23.5MB in size) [Goo17d].

8. Conclusion

8.1. Viability of Web Applications

While new technologies may be exciting to some, it will be hard to establish them in a corporate setting unless revenue can be generated. This section will check how how some businesses managed to adopt progressive web apps by presenting tangible numbers.
In 2017, twitter deployed a *progressive web app* to replace their current mobile web app [Goo17d]. The amount of interaction increased dramatically (75% more *tweets*, 65% more pages per session) and uses only a fraction of the storage (600KB) compared to their current native mobile app (23.5MB for the android app).
The indian cab aggregator Ola also reports success with their *progressive web app* version of their app [Goo17b]. While they do still offer both, the PWA and native apps, they report up to 68% increase in traffic. This is most noticable in less developed cities (Ola reports it as "Tier 3 cities"), where cellular network connection can be unreliable and where many devices have low processing power, memory and storage. In those areas, Ola reports a 30% higher conversation rate[1] on their new PWA compared to their native app. This is likely due to only using 200KB of data compared to their 60MB android app and 100MB iOS app. Repeat visits of the PWA take as little as 10KB to load. It is important to note that most success stories stem from the Google Developers portal and it is hard to find data that is not either directly from Google or Google-hosted events.
Formidable gives some insight on their work for the new *Starbucks* PWA which is 99.84% smaller than the existing iOS app (only 233KB compared to 148MB), while also being faster[For]. However, they do describe reaching this type of "native-like" feel as challenging. They do not report any data of actual usage or conversion rates though.
During finalization of this work, Microsoft has started to publish progressive web apps in their store which have been automatically indexed by the *bing* search engine. Microsoft is treating PWAs as "first-class citizens" [Pfl+18], making them mostly indistinguishable from native applications in windows 10.
It can be concluded that progressive web apps seem to be an attractive alternative for native mobile apps which require an installation step and use more data and storage, while offering a very similar experience. There is not much data about using PWAs as a substitute for native desktop applications, other than windows 10 including them in their store.

[1] Conversion rate describes the amount of users that become a customer

8.2. *Good practices* for Progressive Web Applications

This section will list a recommendation of *good practices* to build progressive web applications. The practices are a direct result of the evaluation process in chapter 7.2 (P.59), and are split into the three categories below.

Organizational Concerns

Train or hire developers to handle the current ECMAScript standard ECMAScript evolved a lot in the recent years and many modern framework and tools expect software engineers to have up-to-date knowledge.

Pick a filename for your service worker and stick with it This ensures that an installed Service Worker is properly replaced. If removal of a service worker is necessary a no-op service worker can be used. A no-op (short for *no operation*) service worker will offer no functionality other than replacing the current one which effectively disables any precaching and networking strategies that may have been set in place by the previous one.

Decouple front-end from back-end This makes it easier to replace specific technologies or reuse specific services for multiple, specialized front-ends (see [jam]).

Functional Concerns

Use HTTPS encryption This offers *confidentiality*, *data-integrity* and *authenticity* to the user which is also highlighted by browser vendors and search engines.

Use a transpiler to ensure browser compatibility Modern transpilers like Babel ensure that modern ECMAScript features will be replaced with polyfills in order to support older Browsers.

Use a cache-first network-strategy whenever possible This will enable browsers to instantly retrieve cached assets without the usual network delay on any consecutive visit[Cam16].

Usability Concerns

Use a responsive grid layout to adapt to varying screen sizes Important content elements need to be properly readable on a wide spectrum of heterogeneous devices.

Pre-cache critical assets to reduce load times Assets that are guaranteed to be required during use of the application (e.g. app shell), should be pre-cached by a service worker to allow almost instant load times.

Use a module bundler in order to push minimal assets on a given route This will further reduce load times as bundle sizes shrink. This is also one of the PRPL-principles.

Use semantic HTML markup and JSON-LD This will make it easier for search engines to index a PWA, which - as a result - will make it easier for potential users to find the application on the internet.

Use ARIA-roles for ambiguous HTML elements ARIA-roles will help assisting technologies like screen readers to tell functional elements apart from content elements.

Pre-render static pages Whenever a page does not contain any dynamic information, consider pre-rendering the page in order to make a site viewable even before any UI framework loads and executes. Pre-rendering can also be used when adopting *isomorphic rendering*[Mar16].

Use the app shell model for dynamic contents The app shell will only need to be loaded once and remains throughout the app navigation. The app shell can also be pre-cached for almost instant loading on consecutive visits.

Offload slow processing to web workers JavaScript may be event-based, but it is still single-threaded. Due to that nature, processing-intensive tasks can affect UI performance noticeably. Web workers are the only way to create true concurrency without affecting the UI thread.

Preload assets that are likely to be required for upcoming routes Assets that are likely to be required upon further navigation may be pre-loaded on the current route, using *<script rel="preload">*. The browser will start downloading the assets as soon as all current network requests are finished. Pre-loading is also one of the PRPL-principles.

Disable text selection on interactive elements like buttons or other non-content Interactive elements may be toggled in rapid succession. On touchscreen devices, this will often cause a text-selection tool to prompt the user, which is likely to be unintended and disruptive.

Use fixed-dimension placeholders for images Images usually have higher load-times than text. Without fixed dimensions, an image that has finished loading will retroactively push contents around, which can interrupt the user that already started to read the text[Cam16].

Use page-transitions to increase *perceived performance* This gives users feedback that their navigation request is processing. A *skeleton-page* should be displayed which is then replaced by actual contents, once the navigation request has been processed [Cam17][Goo17c].

Scroll positions should be preserved when going from a detail-view to a list-view This allows users to continue exploring the previous list with little friction [Cam17][Goo17c].

Offer a share-button if the PWA is able to be launched fullscreen If the PWA manifest allows for fullscreen (or other view modes that hide the address bar), an alternative way to share the current URL with others is recommended[Cam16].

Avoid "hamburger menus" for essential routes Hamburger menus hide possible navigation options from the user and require at least two presses to navigate to any of the main routes[Abr14]. Consider using a bottom navigation[Mat] instead.

8.3. Final Conclusion and Outlook

Progressive web apps present a viable approach to cross-platform needs, as software engineers can completely focus on web APIs without having to address the vast quantity of challenges that emerge when multiple, platform-specific, native applications have to be maintained at once[CY99]. However, not all cross-platform concerns vanish, the UI needs to adopt to varying screen sized and orientations (and ideally even screen readers), which, to get it right, can be a whole discipline in itself. Additionally, web applications are not able to access data on the local file system (unless the user manually provides them each time the application is used), as well as any os-level APIs (e.g. controlling other applications or launching processes), which limits the amount of use cases that can be solved with progressive web apps. There are other solutions like electron[Ele], which expose some of these os-level APIs, but these applications only run within their own instance of a modified chromium browser and need to be downloaded similar to a native app, which is the exact opposite approach (bringing web technologies to native apps), opposed to PWAs (bringing native look and feel to the web). If those concerns do not matter for the application at hand, progressive web apps seem to keep up just fine with their native variants[Goo17d] and even surpass them in some cases[Goo17b], while also using far less storage on the device[For].
Developing a progressive web app requires deep understanding of todays web development with a heavy focus on user interaction. This is also highlighted by the sheer amount of *good practices* that address usability concerns, as listed in the previous section. Progressive web apps grew beyond the simple *input-process-output* model and often adopt complex UIs, where managing state and delivering effective user experience become the main challenges. Additionally, the amount of technologies and patterns that need to be understood keeps growing, which can present a major hurdle for businesses that have to keep adapting, as well as beginner developers that try to enter this domain.
Once those hurdles are overcome, progressive web apps offer the most accessible, frictionless and shareable type of application that is available right now.

Bibliography

[Abr14] Luis Abreu. *Why and How to Avoid Hamburger Menus*. May 14, 2014. URL: https://lmjabreu.com/post/why-and-how-to-avoid-hamburger-menus/ (visited on 04/06/2018).

[Abr17] Dan Abramov. *The melting pot of JavaScript*. 2017. URL: https://increment.com/development/the-melting-pot-of-javascript/ (visited on 04/03/2018).

[Aga+] Vladimir Agafonkin et al. *Leaflet - a JavaScript library for interactive maps*. URL: http://leafletjs.com/ (visited on 04/12/2018).

[Ana16] Andrew Anampiu. *The four principles of OOP*. May 18, 2016. URL: https://anampiu.github.io/blog/OOP-principles/ (visited on 04/11/2018).

[Ang] Angular. *Angular*. Google. URL: https://angular.io/ (visited on 04/12/2018).

[Arc18] Jake Archibald. *The Service Worker Lifecycle*. Jan. 3, 2018. URL: https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle (visited on 01/09/2018).

[Bab] Babel. *Babel - The compiler for writing next generation JavaScript*. URL: https://babeljs.io/ (visited on 04/12/2018).

[bes17] bestof.js.org. *2017 JavaScript Rising Stars*. 2017. URL: https://risingstars.js.org/2017/en/#section-framework (visited on 03/05/2018).

[Boo] Bootstrap. *Grid system*. URL: https://getbootstrap.com/docs/4.0/layout/grid/ (visited on 04/12/2018).

[Bro] Browserslist Contributors. *Browserslist*. Evil Martians. URL: https://github.com/browserslist/browserslist.

[Cam16] Owen Campbell-Moore. *Designing Great UIs for Progressive Web Apps*. May 22, 2016. URL: https://medium.com/@owencm/designing-great-uis-for-progressive-web-apps-dd38c1d20f7 (visited on 04/05/2018).

[Cam17] Owen Campbell-Moore. *Creating UX that "Just Feels Right" with Progressive Web Apps*. Google I/O 2017. May 18, 2017. URL: https://www.youtube.com/watch?v=mmq-KVeO-uU (visited on 03/29/2018).

[CKA] CKAN Association. *About CKAN*. URL: https://ckan.org/about/ (visited on 04/12/2018).

[Cle15] Eric Clemmons. *Javascript Fatigue*. Dec. 27, 2015. URL: https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4 (visited on 04/03/2018).

[Col+] Zach Cole et al. *Simple Grid | Lightweight CSS grid for web development*. URL: http://simplegrid.io/ (visited on 04/12/2018).

[Con] Workbox Contributors. *Workbox | Google Developers*. Google Developers. URL: https://developers.google.com/web/tools/workbox/ (visited on 04/12/2018).

[Cor17] Chris Cordle. *Why Angular 2/4 Is Too Little, Too Late*. June 29, 2017. URL: https://medium.com/@chriscordle/why-angular-2-4-is-too-little-too-late-ea86d7fa0bae (visited on 04/12/2018).

[Cro01] Douglas Crockford. *Private Members in JavaScript*. 2001. URL: https://crockford.com/javascript/private.html (visited on 11/28/2017).

[CY99] Michael A. Cusumano and David B. Yoffie. "What Netscape Learned from Cross-platform Software Development". In: *Commun. ACM* 42.10 (Oct. 1999), pp. 72–78. ISSN: 0001-0782. DOI: 10.1145/317665.317678. URL: http://doi.acm.org/10.1145/317665.317678.

[Dai17] Brandon (@aweary) Dail. *you can push into Array.prototype and totally mess up empty arrays*. Nov. 10, 2017. URL: https://twitter.com/aweary/status/928848521012195328 (visited on 04/11/2018).

[Das18] Dan Dascalescu. *Why "Progressive Web Apps vs. native" is the wrong question to ask*. Chrome Developers. Feb. 2018. URL: https://medium.com/dev-channel/why-progressive-web-apps-vs-native-is-the-wrong-question-to-ask-fb8555addcbb (visited on 04/12/2018).

[Ecm17] Ecma International. *ECMAScript® 2017 Language Specification (ECMA-262, 8th edition, June 2017)*. June 2017. URL: https://www.ecma-international.org/ecma-262/8.0/index.html (visited on 04/11/2018).

[Ele] Electron Contributors. *Electron | Build cross platform desktop apps with JavaScript, HTML, and CSS*. URL: https://electronjs.org/ (visited on 04/12/2018).

[Eng17] Eric Enge. *Mobile vs Desktop Usage: Mobile Grows But Desktop Still a Big Player*. Apr. 5, 2017. URL: https://www.stonetemple.com/mobile-vs-desktop-usage-mobile-grows-but-desktop-still-a-big-player/ (visited on 03/27/2018).

[Fen12] Steve Fenton. *Compiling vs Transpiling*. Nov. 18, 2012. URL: https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/ (visited on 04/12/2018).

[For] Formidable. *Starbucks - Formidable Case Study*. URL: https://formidable.com/work/starbucks-progressive-web-app/ (visited on 04/12/2018).

[Geo] GeoNode Contributors. *GeoNode*. URL: http://geonode.org/ (visited on 04/12/2018).

[Goo17a] Google Developers. *Introduction to Structured Data*. Google Developers. Sept. 13, 2017. URL: https://developers.google.com/search/docs/guides/intro-structured-data (visited on 04/12/2018).

[Goo17b] Google Developers. *Ola drives mobility for a billion Indians with Progressive Web App*. May 17, 2017. URL: https://developers.google.com/web/showcase/2017/ola (visited on 11/20/2017).

[Goo17c] Google Developers. *Progressive Web App Checklist*. Nov. 14, 2017. URL: https://developers.google.com/web/progressive-web-apps/checklist (visited on 02/28/2018).

[Goo17d] Google Developers. *Twitter Lite PWA Significantly Increases Engagement and Reduces Data Usage*. May 17, 2017. URL: https://developers.google.com/web/showcase/2017/twitter (visited on 11/20/2017).

[Gri17] Alex Grigoryan. *The Benefits of Server Side Rendering Over Client Side Rendering*. Apr. 17, 2017. URL: https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8 (visited on 01/09/2018).

[Hic15] Ian Hickson. *Web Workers*. W3C. Sept. 24, 2015. URL: https://www.w3.org/TR/workers/ (visited on 04/12/2018).

[Hun13] Pete Hunt. *React: Rethinking best practices*. JSConf EU 2013. Oct. 30, 2013. URL: https://www.youtube.com/watch?v=x7cQ3mrcKaY (visited on 03/05/2018).

[Iri18] Paul Irish. *Flexbox layout isn't slow*. Google Developers. Jan. 3, 2018. URL: https://developers.google.com/web/updates/2013/10/Flexbox-layout-isn-t-slow (visited on 03/13/2018).

[Isa18] Ayo Isaiah. *What every Web Developer should know about HTTPS Part 1 - The value of HTTPS*. Jan. 28, 2018. URL: https://freshman.tech/the-value-of-https/ (visited on 04/12/2018).

[jam] jamstack.org. *JAMstack | JavaScript, APIs, and Markup*. URL: https://jamstack.org/ (visited on 03/12/2018).

[Jan17] Peter Jang. *Modern JavaScript Explained For Dinosaurs*. Oct. 18, 2017. URL: https://medium.com/@peterxjang/modern-javascript-explained-for-dinosaurs-f695e9747b70 (visited on 04/09/2018).

[Job13] William Jobe. "Native Apps vs. Mobile Web Apps". In: *International Journal of Interactive Mobile Technologies (iJIM)* 4 (2013).

[KGB18] Meggin Kearney, Dave Gash, and Alice Boxhall. *Introduction to ARIA*. Google Developers. Jan. 3, 2018. URL: https://developers.google.com/web/fundamentals/accessibility/semantics-aria/ (visited on 04/12/2018).

[KH] James G. Kim and Michael Hausenblas. *5-star Open Data*. URL: http://5stardata.info/en/ (visited on 04/12/2018).

[Kra] Stefan Krause. *Results for js web frameworks benchmark – round 7*. URL: http://www.stefankrause.net/js-frameworks-benchmark7/table.html (visited on 04/12/2018).

[Lin17] Tanner Linsley. *Introducing React-Static - A progressive stat-site framework for React!* Oct. 5, 2017. URL: https://medium.com/@tannerlinsley/%EF%B8%8F-introducing-react-static-a-progressive-static-site-framework-for-react-3470d2a51ebc (visited on 03/07/2018).

[Mar] MariaDB Foundation. *About MariaDB*. URL: https://mariadb.org/about/ (visited on 04/12/2018).

[Mar16] Azat Mardan. *Why Everyone is Talking About Isomorphic / Universal JavaScript and Why it Matters*. Capital One. Mar. 21, 2016. URL: https://medium.com/capital-one-developers/why-everyone-is-talking-about-isomorphic-universal-javascript-and-why-it-matters-38c07c87905 (visited on 04/12/2018).

[Mat] Material Design Contributors. *Bottom Navigation*. Google. URL: https://material.io/guidelines/components/bottom-navigation.html (visited on 04/12/2018).

[MDN17] MDN web docs. *Object.prototype.__proto__*. Nov. 28, 2017. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto (visited on 04/11/2018).

[MDN18a] MDN web docs. *FileSystem*. mozilla. Jan. 15, 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/FileSystem (visited on 04/12/2018).

[MDN18b] MDN web docs. *Manipulating the browser history*. Feb. 19, 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/History_API (visited on 04/12/2018).

[Mil15] Jason (@_developit) Miller. *WTF is JSX*. July 7, 2015. URL: https://jasonformat.com/wtf-is-jsx/ (visited on 04/12/2018).

[Neta] Netlify. *headlessCMS | A List of Content Management Systems for JAM-stack Sites*. Netlify. URL: https://headlesscms.org/ (visited on 04/12/2018).

[Netb] Netlify. *Top Open-Source Static Site Generators - StaticGen*. Netlify. URL: https://www.staticgen.com/ (visited on 04/12/2018).

[Neu17] Jens Neuhaus. *Angular vs. React vs. Vue: A 2017 comparison*. Aug. 28, 2017. URL: https://medium.com/unicorn-supplies/angular-vs-react-vs-vue-a-2017-comparison-c5c52d620176 (visited on 03/05/2018).

[Nor07] Dan North. *What's in a Story?* Feb. 11, 2007. URL: https://dannorth.net/whats-in-a-story/ (visited on 01/09/2018).

[OGC] OGC. *Web Map Service*. URL: http://www.opengeospatial.org/standards/wms (visited on 04/12/2018).

[Osm17a] Addy Osmani. *The App Shell Model*. Sept. 26, 2017. URL: https://developers.google.com/web/fundamentals/architecture/app-shell (visited on 11/16/2017).

[Osm17b] Addy Osmani. *The PRPL Pattern*. Sept. 26, 2017. URL: https://developers.google.com/web/fundamentals/performance/prpl-pattern/ (visited on 11/16/2017).

[Pfl+18] Kyle Pflug, Kirupa Chinnathambi, Aaron Gustafson, and Iqbal Shahid. *Welcoming Progressive Web Apps to Microsoft Edge and Windows 10*. Microsoft. Feb. 6, 2018. URL: https://blogs.windows.com/msedgedev/2018/02/06/welcoming-progressive-web-apps-edge-windows-10/ (visited on 04/12/2018).

[Pha] PhantomJS. *PhantomJS*. URL: http://phantomjs.org/ (visited on 04/11/2018).

[Pop17] Alexander Pope. *ServiceWorkers Outbreak: index-sw-9a4c43b4b47781ca619eaaf5ac1db.js*. JSConf EU 2017. May 16, 2017. URL: https://www.youtube.com/watch?v=CPP9ew4Co0M (visited on 03/13/2018).

[Pos18] Jeff Posnick. *Service Worker Registration*. Jan. 3, 2018. URL: https://developers.google.com/web/fundamentals/primers/service-workers/registration (visited on 01/09/2018).

[Pot] John Potter. *npm trends: Compare NPM package downloads*. URL: http://www.npmtrends.com/@angular/core-vs-react-vs-vue (visited on 04/12/2018).

[Rai+] Nick Raienko et al. *Awesome React*. URL: https://github.com/enaqx/awesome-react (visited on 04/12/2018).

[Reaa] React. *React - A JavaScript library for building user interfaces*. Facebook Inc. URL: https://reactjs.org/ (visited on 04/12/2018).

[Reab] React Contributors. *Composition vs Inheritance*. Facebook Inc. URL: https://reactjs.org/docs/composition-vs-inheritance.html (visited on 04/12/2018).

[Reac] React-Static. *react-static*. URL: https://react-static.js.org/ (visited on 04/12/2018).

[Rea18] React-Static. *Core Concepts*. Mar. 23, 2018. URL: https://github.com/nozzle/react-static/blob/master/docs/concepts.md (visited on 03/27/2018).

[RSH09] Chris Rupp, Matthias Simon, and Florian Hocker. "Requirements Engineering und Management". In: *HMD Praxis der Wirtschaftsinformatik* 46.3 (June 2009), pp. 94–103. ISSN: 2198-2775. DOI: 10.1007/BF03340367. URL: https://doi.org/10.1007/BF03340367.

[Rub18] Daniel Rubino. *First Windows 10 Progressive Web Apps (PWA) published by Microsoft hit the Store*. Apr. 7, 2018. URL: https://www.windowscentral.com/g00/first-batch-windows-10-progressive-web-apps-here?i10c.encReferrer=&i10c.ua=1&i10c.dv=14 (visited on 04/12/2018).

[Rus+17] Alex Russel, Jungkee Song, Jake Archibald, and Marijn Kruisselbrink. *Service Workers 1*. W3C. Nov. 2, 2017. URL: https://www.w3.org/TR/service-workers-1/ (visited on 04/12/2018).

[Rus15] Alex Russell. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. June 15, 2015. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/ (visited on 01/09/2018).

[Sch16] Isaac Z. Schlueter. *kik, left-pad, and npm*. Mar. 23, 2016. URL: http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm (visited on 02/23/2018).

[Sel] Selenium HQ. *Selenium - Web Browser Automation*. URL: https://www.seleniumhq.org/ (visited on 04/11/2018).

[Ser] ServiceWorker Cookbook. *Caching Strategies*. Mozilla. URL: https://serviceworke.rs/caching-strategies.html (visited on 04/12/2018).

[Shi16] Michael Shilman. *Testing Frameworks*. 2016. URL: http://2016.stateofjs.com/2016/testing/ (visited on 12/12/2017).

[Ske] Skeleton. *Skeleton: Responsive CSS Boilerplate*. URL: http://getskeleton.com/ (visited on 04/12/2018).

[Smu12] P. Smutný. "Mobile development tools and cross-platform solutions". In: *Proceedings of the 13th International Carpathian Control Conference (ICCC)*. May 2012, pp. 653–656. DOI: 10.1109/CarpathianCC.2012.6228727.

[Sta+] Patrick Stapleton et al. *Awesome Angular*. URL: https://github.com/gdi2290/awesome-angular (visited on 04/12/2018).

[sta18] statcounter. *Desktop vs Mobile vs Tablet Market Share Worldwide*. Feb. 2018. URL: http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet (visited on 03/27/2018).

[UIk] UIkit. *UIkit*. URL: https://getuikit.com (visited on 04/12/2018).

[Ver14] Dave Verduyn. *About the Kano Model*. Mar. 17, 2014. URL: https://www.kanomodel.com/about-the-kano-model/ (visited on 04/12/2018).

[Vue] Vue.js. *Awesome Vue.js*. URL: https://github.com/vuejs/awesome-vue (visited on 04/12/2018).

[W3Sa] W3Schools. *CSS @media Rule*. W3C. URL: https://www.w3schools.com/cssref/css3_pr_mediaquery.asp (visited on 04/12/2018).

[W3Sb] W3Schools. *HTML5 Semantic Elements*. W3C. URL: https://www.w3schools.com/html/html5_semantic_elements.asp (visited on 04/12/2018).

[Wal14] Philip Walton. *Implementing Private and Protected Members in JavaScript*. Apr. 9, 2014. URL: https://philipwalton.com/articles/implementing-private-and-protected-members-in-javascript/ (visited on 12/12/2017).

[web] webpack. *webpack module bundler*. URL: https://webpack.github.io/ (visited on 04/12/2018).

[Wik18] Wikipedia Contributors. *Responsive Web design*. Mar. 27, 2018. URL: https://en.wikipedia.org/w/index.php?title=Responsive_web_design&oldid=832660943 (visited on 04/12/2018).

[Wil+16] Christian Willmes, Yasa Yener, Anton Gilgenberg, and Georg Bareth. "CRC806-Database: Integrating Typo3 with GeoNode and CKAN". In: vol. 96. Kölner Geographische Arbeiten. Kölner Geographische Arbeiten. 2016. DOI: 10.5880/TR32DB.KGA96.17.

[Wil+17] Christian Willmes, Daniel Becker, Jan Verheul, Yasa Yener, Mirijam Zickel, Andreas Bolten, Olaf Bubenzer, and Georg Bareth. "PaleoMaps: SDI for open paleoenvironmental GIS data". In: *International Journal of Spatial Data Infrastructures Research* 12 (2017), pp. 39–61. DOI: 10.2902/1725-0463.2017.12.art3. URL: http://ijsdir.jrc.ec.europa.eu/index.php/ijsdir/article/view/431.

[Wil16] Christian Willmes. *CRC806-Database: A Semantic E-Science Infrastructure for an Interdisciplinary Research Centre*. Universität zu Köln, 2016. URL: http://kups.ub.uni-koeln.de/7381/.

[Wor] Workbox Contributors. *Workbox Precaching*. Google Developers. URL: https://developers.google.com/web/tools/workbox/modules/workbox-precaching (visited on 04/12/2018).

[Yen17] Yasa (@Kisaro) Yener. *Thanks, #javascript. That wasn't quite what I expected.* June 15, 2017. URL: https://twitter.com/Kisaro/status/864169660987658240 (visited on 04/11/2018).

[You] Evan You. *Vue.js*. URL: https://vuejs.org/ (visited on 04/12/2018).

[Zai17] Vitali Zaidman. *An Overview of JavaScript Testing in 2017*. Apr. 19, 2017. URL: https://medium.com/powtoon-engineering/a-complete-guide-to-testing-javascript-in-2017-a217b4cd5a2a (visited on 01/08/2018).

List of Figures

Appendix

1. ECMAScript 2015+

The contents of this and the following sections in this chapter will dive deeply into the specifics of JavaScript and thus, ECMAScript. While basic knowledge of ECMAScript is expected in order to benefit most from these technical sections, there are some newer aspects that have risen with the advent of ECMAScript 2015+. Some of these language additions will be referenced multiple time in this work and may be new even to the target audience. This section will touch on the most important ones to ensure a better grasp throughout this work.

Declaring Variables and Constants

For the longest time, variables have been declared using the *var* keyword. This does have some side effects though. In JavaScript, variables are *function-scoped* instead of *block-scoped*. So if a variable is defined e.g. inside a loop, this variable will be available everywhere in the parent function (see fig..1). In modern JavaScript, the keyword *let*

```
var t = function() {
  for(let i = 0; i < 5; i++) {
    var x = "Test"
  }
  return x
}
t() // returns "Test"
```

Figure .1.: Accessing a variable from outside the block it has been defined

can be used instead of *var*. Not only is *let* properly block-scoped, it also comes with another useful trait. A common use case in JavaScript applications is iterating through a list of UI elements, like buttons, and apply an eventlistener to them, to be able to react when a click occurs on those buttons (see fig..2). In that example, using *let* ensures that each button click will alert with the number that the index variable i had at the time the eventlistener callback has been created. On a page with 10 buttons, the first button will alert "I am button #0", while the last button will alert "I am button #9". If *var* had been used instead of *let*, all eventhandler callbacks would alert "I am button #9" regardless of which button is pressed, because *var* is function scoped it is merely referenced by those callbacks instead of copied and by the time a button is clicked, the loop has already finished and the index variable has been incremented to the maximum count. To conclude, unless function-scoped behavior is intended (it

```
const buttons = document.getElementsByTagName('button')
for(let i = 0; i < buttons.length; i++)
  buttons[i].addEventListener('click', function() {
    alert('I am button #' + i)
  })
```

Figure .2.: Using *let* ensures that the current value if i is copied to the event listener callbacks

almost never is), *let* should always be used over *var*.
Additionally, ES2015+ allows to define constants using the *const* keyword. Once a value is assigned, the value of a constant can not be changed. It is important to note here that if an object is declared as a constant, while it cannot be overwritten, the object does *not* become immutable. Any object declared as a constant can still completely change its state.
Finally, whenever a value has to be declared, in many cases *const* is a good choice, only when a value needs to be changed (and it really is a primitive value, not an object) later in the code, *let* should be used. This approach dismisses whole classes of issues that slow down productivity.

Declaring Functions Using the Arrow Notation

Functions are an essential fragment of any programming language. They allow to map a specific output to any given input. The arrow notation focuses on exactly that part by simplifying how functions are declared (see fig..3). On the left side of the arrow

```
const add = (a, b) => a + b
add(2, 3) // returns 5
```

Figure .3.: Arrow functions quickly map a specific output to any given input

are the inputs, if there is only one input, the parentheses can be omitted, on the right side of the arrow is the actual code that produces the output. If there is only one statement, the arrow function will do an *implicit return*. It will automatically return the result of that one statement. Alternatively, a code block can be used after the arrow using curly braces. In that case, there has to be an explicit return, using the *return* keyword as in usual functions. While arrow functions can be quicker to write, they also bring another important benefit. Since the *function*-keyword is not used, they do not create a *function scope*. This is important whenever the *this*-keyword is used, as it will always reference the current function scope. This can be useful when a callback function needs to access properties outside of the callback function. Previously, this needed less intuitive work-arounds to deal with, with arrow functions, this does become a lot easier (see fig..4).

```
class App {
    constructor() {
        const button = document.getElementById('somebutton')
        // 1. Deprecated work-around,
        // should absolutely be avoided!
        const that = this
        button.addEventListener(function() {
            that.handleClick()
        })

        // 2. Better solution, the need of bind(this)
        //  needs to be memorized though
        button.addEventListener(function() {
            this.handleClick()
        }.bind(this))

        // 3. Compact solution using arrow notation
        button.addEventListener(_ => this.handleClick())
    }

    handleClick() {
        // this needs to be accessed from a callback
    }
}
```

Figure .4.: Three ways to achieve the same goal of accessing *this*

Template Literals and Tagged Templates

Concatenating strings and variables can be straight forward in simple cases like this:

```
console.log("Hello, " + user.name + "!")
```

But it can become a lot less readable, when specific markup is involved, e.g.:

```
document.write("<a href=\"" + link.url + "\">" + link.label
   + "</a>")
```

This is where template literals can aid the developer. For this, strings are enclosed with the "‘" character while JavaScript expressions can be embedded everywhere in the string while enclosed in *${...}*. The above example, where HTML markup for a link is assembled, can be simplified as:

```
document.write(`<a href="${link.url}">${link.label}</a>`)
```

Additionally, these templates can be *tagged* in order to utilize custom parsing functions. As an example, Facebook's GraphQL uses tagged templates in order to automatically parse GraphQL queries:

```
import {graphql} from 'graphql'
const query = graphql`
  query LayoutQuery {
    site {
      siteMetadata {
        title
      }
    }
  }
`
```

The above template will automatically be passed as a parameter to the *graphql*-function that is imported at the top (more on imports on P.XIV).

Spread and Rest Operator

Spread and rest operators are an easy way to assign all elements of an array. This can be useful to define a function where separate parameters will be accumulated (rest operator):

```
const sum = (...numbers) => numbers.reduce((a, b) => a + b)
sum(1, 2) // returns 3
sum(1, 1, 1, 1, 1) // returns 5
```

They are also useful for the opposite case where an accumulated list of parameters should be passed separately onto a function (spread operator):

```
const getDistance = (x, y, x2, y2) =>
    Math.sqrt((x+x2)**2+(y+y2)**2)

const point1 = [3, 4]
const point2 = [7, 2]
// assign array values to all four parameters
getDistance(...point1, ...point2)
```

Another useful application for the spread operator is to achieve a *shallow copy* of an existing array:

```
const array1 = [1, 2, 3]
const array1copy = array1
array1copy[0] = 5
array1[0] // returns 5 since both constants reference the
   same array

const array2 = [1, 2, 3]
const array2copy = [...array2]
array2copy[0] = 5
array2[0] // returns 1 as expected
```

ES2015+ Modules

ES2015+ Modules allow to split codes into modules (not to be confused with the module pattern in chapter 3, P. XVI). Modules allow for code maintainability and code reuse across the same or different projects. Modules are pieces of code that exclusively disclose intended functionalities using the *export*-keyword. Other Modules can then reuse that functionality using the *import*-keyword. While modules can export multiple properties or functions, they can have only one default export. When properties are imported, they have to be imported using the name of the property, but can be aliased using the *as*-keyword. The default export can be imported using any name preferred. Additionally, non-default exports have to be imported using curly braces (see fig..5 for examples).

```
// In file: JSONClient.js
const version = 'v3'
const url = 'http://example.com/' + version + '/api.json'
const get = _ => {
    return await fetch(url)
                .then(response => response.json())
}
export {url: url, version: version}
export default get

// In file: index.js
import {url} from 'JSONClient.js'
import JSONClient from 'JSONClient.js'
console.log('Retrieving data from ' + url)
console.log(JSONClient())
```

Figure .5.: Exporting functionalities in form of modules, that an be reused

2. Examples for Prototypical Inheritance in JavaScript

This section goes into detail on how to use inheritance when building JavaScript applications. The goal is a better understanding how implementation is done and how certain pitfalls can be avoided.
Since prototypes are working objects, the constructor function is defined first. Only after that is done, the remaining properties and functions can be defined by using the *prototype*-property (see fig..6). This is the equivalent to creating an *Animal*-class with constructor, the property *name* and the method *getName()*. There is no protection against manipulating the prototype, which means later code could easily add new functions or change how existing functions work, which will affect all instances. Especially in the context of web applications, code can easily be manipulated by the client and is not to be trusted. If data integrity is a concern, critical operations will still need to be done on the server side, adding the requirement of having a working

```
const Animal = function(name) {
    this.name = name
}
Animal.prototype = {
    getName: function() {
        return this.name
    }
}
const dog = new Animal('Dog')
dog.getName() // returns 'Dog'
```

Figure .6.: Basic definition of a prototype and its instantiation

network connection at all times. However, in most use cases, the user would have no interest in breaking the application exclusively on his own end and as long as user-manipulated data has no way to feed back into the system that affects other users, the risk is negligible. It should be noted that since ECMAScript 2015, there is actually a *class*-Keyword to create classes in a more traditional way (see fig. .7). However this is only *syntactical sugar*, it masks away prototypical inheritance but still uses it and all the behavior that comes with it. To emphasize on the inherently prototypical behavior, the *class*-style syntax is mostly avoided in this section. If that caveat is kept in mind, it is a very efficient syntax, in regards to the amount of statements to write, and is more similar to the way inheritance is usually done in certain other languages. Inher-

```
class Animal {
    constructor(name) {
        this.name = name
    }
    getName() {
        return this.name
    }
}
const dog = new Animal('Dog')
dog.getName()
```

Figure .7.: ECMAScript 2015 style class definition, using the *class*-keyword

iting from a prototype is done by copying and extending the prototype of an existing class. The constructor will have to also call the constructor of the class the current class is inheriting from (see fig. .8). Using *.call* is comparable with the *super()*-method in Java, except the current instance always has to be passed using *this* as the first argument, before any potential argument of the parent method (or constructor in this case) is set. In this example, the *Animal()*-constructor only has the argument *name*, yet when invoked using the *.call()*-method, the current instance is passed before the actual *name*-argument. When using the *class*-keyword, inheritance can be done using

```
const Dog = function() {
    Animal.call(this, 'Dog')
}
Dog.prototype = Object.assign(
    Object.create(Animal.prototype),
    {
        sound: function() {
            return 'Woof!'
        }
    }
)
const dog = new Dog()
dog.sound() // returns 'Woof!'
dog.getName() // returns 'Dog'
```

Figure .8.: Prototypical inheritance of an existing class (Animal)

the *extends*-keyword.

```
class Dog extends Animal {...}
```

JavaScript interpreters resolve instance properties by utilizing the *prototype chain*. Each object instance has a *__proto__*-property containing everything defined in the prototype that instance is based off. When the browser interprets the *dog.sound()* call, it first checks whether *dog.__proto__* contains a function called *sound*. Since the *dog*-instance is based off of *Dog.prototype*, which does in fact contain a *sound*-function, the interpreter will know how this method works by merely entering the first level of the prototype chain. However, for the next line, *getName* is not to be found within the Dog-prototype. In this case, the interpreter will look one level deeper into *dog.__proto__.__proto__* which contains the prototype *Animal*, that *Dog* is based off of. Since animal does contain *getName*, the interpreter will know how to execute the method by going two levels into the prototype-chain. Ultimately, all object instances inherit from the general *Object*-prototype which ends all prototype chains. If no property-definition has been found by the time the interpreter arrived at the end of the prototype chain, an error will be thrown.

3. Module Pattern

The module pattern is an alternative way to structure code by mainly using JavaScript objects. The key idea is to define publicly visible instance properties within a plain JavaScript object, while private properties are defined outside of that object. This all happens within a self-executing, anonymous function, where only the object with its public properties is returned (see fig. .9). It is important to note that this is an older pattern that has been used long before ES2015+ modules existed and differs greatly

from those. ES2015+ modules allow to export anything useful, from constants, and functions to complete classes (refer to chapter 1 on P.XIV for more info). The module pattern on the other hand, is an alternative way to structure code opposed to using classes.

With that being said, back to how the module pattern works: A module named

```
const Person = (function() {
    // is only available within the module
    const think = function(message) {
        return message
    };

    return {
        // can be called from outside the module
        say: function(message) {
            console.log(think(message))
        }
    }
})()
Person.say("Hello") // logs "Hello"
Person.think("some secret") // will throw an error
```

Figure .9.: Definition of a simple module using a self-executing, anonymous function

Person is defined with the methods *think()* and *say()*. Since *say()* is contained within the object that is returned by the anonymous function, it will be publicly available as a property of the *Person* module. The *think()* method, on the other hand, is not returned and only available within the function scope. Since the function is, as mentioned, anonymous *and* self-executing, there is no way to address its contents anymore. However, the *say()* method being defined in that same function scope is still able to access *think()*, thus resembling a public/private encapsulation. It is important to note that the module is not instantiated using the *new* keyword and thus does not constitute an instance of a class. It is more comparable to *static* classes in other programming languages or the *Singleton*-Pattern, where everything operates on the same object instead of having multiple, independent instances.
Additionally, modules can be extended by passing it as a parameter to the self-executing function of a new module (see fig. .10). Some sources that mention extending modules will omit using *Object.assign()* and instead add new properties directly to the base module[1]. However, this will also change the base module, making the definition of an extended module redundant. Other authors even recommend assigning the base module to the *__proto__*-property of the extending module[2], which exploits the prototype-chain meant for actual inheritance, to implement the module pattern.
Object.assign() seems to be the most appropriate tool for this purpose, as it does exactly

[1] https://toddmotto.com/mastering-the-module-pattern/
[2] http://metaduck.com/08-module-pattern-inheritance.html

```
const Cook = (function(Person) {
    const secretIngredient = "Cinnamon"
    return Object.assign({}, Person, {
        cook: function(ingredients) {
            ingredients = ingredients || []
            ingredients.push(secretIngredient)
            console.log(ingredients.join(' and '))
        }
    })
})(Person)
// Logs "Oatmeal and Milk and Sugar and Cinnamon"
Cook.cook(["Oatmeal", "Milk", "Sugar"])
```

Figure .10.: The *Cook* module extends the *Person* module with further functionality

what is needed: Merging one or multiple object properties into one. It is important to note that *Object.assign()* does a *shallow copy* (opposed to a reference) of the supplied objects, which ensures that any modification on the extended module will not affect the module it is based off of.
While the module-pattern solves the missing encapsulation, it seems to miss out on the concept of independently working instances. One possible workaround is a combination of using prototypes with the module-pattern, where the prototype is defined within an anonymous function which returns the final prototype (see fig..11). The downside of

```
const Animal = (function() {
    var animalName
    const AnimalClass = function(name) {
        animalName = name
    }
    AnimalClass.prototype = {
        getName: function() {
            return animalName
        }
    }
    return AnimalClass
})()
const dog = new Animal('Dog')
dog.getName() // returns 'Dog'
dog.name // returns undefined
dog.animalName // returns undefined
```

Figure .11.: Combining a prototype definition with the module-pattern

this approach is the limited usefulness of the *this*-keyword, which is meant to allow addressing any property of an object. When the supposedly *private* properties are

defined outside the prototype (which they have to be, if they are to be made inaccessible from the outside), they can not be reached via *this*-keyword. This is also true in the opposite direction where the private methods and properties outside the prototype can not easily access properties within the prototype. This can be partly solved when moving further away from the module-pattern and defining private properties inside the constructor (see fig..12). With this approach, private properties still can not be accessed using *this*-keyword, but it will now work with the opposite direction, which means that private properties and methods will now be able to easily access all other properties and methods using the *this*-keyword. In real-world scenarios, where classes

```
const Animal =  function(name) {
    var animalName = name // inaccessible from outside
    this.getName = function() {
        return animalName
    }
}
Animal.prototype = {
    // reserved for public properties or
    // methods that purely access public properties
}
const dog = new Animal('Dog')
dog.getName() // returns 'Dog'
dog.name // returns undefined
dog.animalName // returns undefined
```

Figure .12.: Definition of private properties inside the constructor

can easily grow into lots of private properties/methods, this approach of defining those in the constructor will become increasingly less maintainable. The concept of having a prototype which contains all of an objects definition becomes simply less true while the code becomes harder to read. Additionally, while the concept behind *private/public* encapsulation has been implemented, there is still no equivalent to *protected* visibility, where methods and properties are only hidden from the outside but not from child classes. With the currently shown implementations, any objects that inherits from a base class with hidden properties will not be able to access those. Google engineer Philip Walton suggests very sophisticated solutions [Wal14] to the aforementioned challenges by implementing a private storage, for which the object instance can be passed to in order to approximate a similar syntax as using *this* for those private properties. However, Waltons illustration of a *protected*-visibility implementation introduce additional helper classes to do inheritance in order to distribute protected properties to child classes. This solution also introduces an additional dependency in the form of *node.js*.
With both, Walton and Crockford [Cro01], heavily advocating the use of proper encapsulation, there is a need to reflect upon the value that encapsulation grants to software engineers. One aspect is the ability of *information-hiding* in order to protect critical properties from being accessed/overwritten, which grants consistency and robustness

to an object. Another aspect is its self-documenting nature, as it clearly shows developers that need to use an object, which properties and methods are part of its API, and which properties and methods are only for internal use. Pseudo-private properties with a prefixed underscore will allow a comparable distinction, even though there is no compiler nor interpreter that will enforce correct usage. Taking a look back at the context of applications running inside the browser, it is important to understand that client-side code can *not* be trusted from a service providers side of view. JavaScript code running in the browser is completely hackable by design and to this date there are no protection- or validation-mechanisms. This begs the question if the added robustness of proper encapsulation is worth the added complexity of actually implementing it, given how the actual execution is outside of the developers control, and needs to be decided on a case-per-case basis.

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, April 13, 2018

Yasa Yener