Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Janne Vanhala

Implementing an Offline First Web Application

Master's Thesis
Espoo, November 2, 2017

Supervisor: Professor Petri Vuorimaa
Advisor: Timo Herttua, M.Sc. (Econ.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Janne Vanhala		
Title:	Implementing an Offline First Web Application		
Date:	November 2, 2017	**Pages:**	vii + 45
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Petri Vuorimaa		
Advisor:	Timo Herttua, M.Sc. (Econ.)		

The Internet usage of mobile and tablet devices combined overtook desktop for the first time in October 2016. Unlike desktop computers, which usually have physical access to the Internet, mobile devices' connectivity is not guaranteed even in a highly connected area such as a large city in a well-developed country. Dead zones can be encountered in many situations such as while commuting to work, standing in the wrong corner of a room, or attending an event with a large crowd. Despite this, web developers often build apps under the assumption that everyone has a fast and fixed broadband connection. Unfortunately, this leads to a horrible user experience for many people. A recent design paradigm, offline-first, aims to tackle this issue by focusing on the offline experience first.

The goal of this thesis is to investigate how a web application can be implemented with the offline-first mindset. The study consists of two parts: a literature study, which reviews the browser features that enable offline web applications, and a constructive study, in which a prototype web application using the offline-first approach is designed and implemented.

The prototype developed in this thesis presents solutions to problems such as accessing the application offline, persisting application data to the browser storage and synchronizing the local state to a remote server. However, it was revealed that there are a lot more aspects to consider in the scope of offline web applications. Challenges including conflict management and background synchronization were left to be solved in the future. Nevertheless, the prototype should still serve as a valuable reference to anyone who wants to build an offline-first web application.

Keywords:	offline-first, web application, single-page application, synchronization
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Master's Programme in Computer, Communication and Information Sciences

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Janne Vanhala		
Työn nimi:	Web-sovelluksen toteutus yhteydetön tila ensin		
Päiväys:	2. marraskuuta 2017	**Sivumäärä:**	vii + 45
Pääaine:	Computer Science	**Koodi:**	SCI3042
Valvoja:	Professori Petri Vuorimaa		
Ohjaaja:	Timo Herttua, KTM		

Mobiili- ja tablettilaitteet ohittivat Internetin käytössä työpöytälaitteet ensimmäistä kertaa lokakuussa 2016. Toisin kuin pöytätietokoneet, joilla on yleensä fyysinen pääsy Internetiin, mobiililaitteiden yhteyden laatu ei ole taattu edes alueilla, joilla on korkea matkaviestinverkon kuuluvuus, kuten kehittyneiden maiden suurissa kaupungeissa. Katvealueisiin voi törmätä monissa tilanteissa, kuten työmatkalla, väärässä huoneen nurkassa seisoessa tai suureen yleisötapahtumaan osallistuessa. Tästä huolimatta web-kehittäjät usein rakentavat sovelluksia sillä oletuksella, että kaikilla on nopea ja kiinteä laajakaistayhteys. Valitettavasti tämä johtaa kamalaan käyttökokemukseen monien ihmisten kohdalla. Yhteydetön tila ensin on tuore suunnittelumalli, joka pyrkii ratkaisemaan tämän ongelman keskittymällä ensin yhteydettömän tilan kokemukseen.

Tämän diplomityön tavoitteena on tutkia, kuinka web-sovelluksen voi toteuttaa yhteydetön tila ensin -ajattelutavalla. Tutkimus koostuu kahdesta osasta: kirjallisuustutkimuksessa käydään läpi yhteydettämän tilan web-sovellusten mahdollistamat selainominaisuudet; konstruktiivisessa tutkimuksessa suunnitellaan ja toteutetaan prototyyppi web-sovelluksesta käyttäen yhteydetön tila ensin -lähestymistapaa.

Tässä diplomityössä kehitetty prototyyppi esittää ratkaisut ongelmiin, kuten miten sovellukseen pääsee yhteydettömässä tilassa, miten sovelluksen tiedot voi säilöä selaimen tallennustilaan ja miten selaimen paikallisen tilan voi synkronoida etäpalvelimelle. Tutkimuksessa kuitenkin paljastui, että yhteydettömän tilan web-sovelluksiiin liittyy paljon enemmänkin seikkoja, joita tulee ottaa huomioon. Haasteet mukaan lukien konfliktien hallinta ja taustasynkronointi jäivät jatkotutkimuksissa ratkaistavaksi. Kaikesta huolimatta prototyyppi voi silti olla arvokas esimerkki kelle tahansa, joka haluaa rakentaa web-sovelluksen yhteydetön tila ensin.

Asiasanat:	yhteydetön tila ensin, web-sovellus, yhden sivun sovellus, synkronointi
Kieli:	Englanti

Acknowledgements

First, I would like to thank my supervisor Petri Vuorimaa and my advisor Timo Herttua for finding the time to guide me on this journey.

Furthermore, I would like to express my gratitude to all my colleagues at our company who made it possible for me to work full-time on my thesis. Special shout-outs go to Samuli Suortti and Pekka Pöyry for proof-reading my text, for the excellent peer support, and for all the suggestions. Also, big thanks to Dan Gebhardt for responding quickly to all my questions and issues about Orbit.

Last but not least, I want to thank you, Mirka. I could not have done this without your love, support, and encouragement.

Espoo, November 2, 2017

Janne Vanhala

Abbreviations and Acronyms

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CSS	Cascading Style Sheets
DUT	Device Under Test
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
ID	Identifier
IndexedDB	Indexed Database API
JSON	JavaScript Object Notation
SPA	Single-Page Application
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
W3C	World Wide Web Consortium
WebSQL	Web SQL Database
WHATWG	Web Hypertext Application Technology Working Group
WICG	Web Platform Incubator Community Group

Contents

Chapter 1

Introduction

We are living in a world where we are using more and more mobile devices to access the Internet. According to StatCounter, the Internet usage of mobile and tablet devices combined overtook desktop for the first time in October 2016 [1]. As of August 2017, mobile held a market share of 52.64 %, desktop 42.75 %, and tablet 4.62 % [2].

Unlike desktop computers, which are usually in a static location and have physical access to the Internet, mobile devices are often moving wherever their owners are going. Mobile broadband coverage is highly variable depending on the spot, which creates a connectivity challenge. Even in a highly connected area, such as a large city in a well-developed country, connectivity is not guaranteed. There can still be dead zones in many places. You might be commuting to work by bus. You might be standing in the wrong corner of a room. Or, you might be taking an elevator to the top floor. Besides dead zones, hugely crowded areas, such as music festivals, sporting events, and conferences, can also be a problem, since you are sharing the WiFi or 3G with thousands of other people. The situation is obviously worse, if you decide to have a vacation in the wilderness without mobile coverage, or if you happen to live in a third-world country with access to a GPRS connection only.

Web developers often assume that users are similar to them: using the latest hardware and software with the fastest Internet connection. Despite this, web developers build their apps on their modern desktop computers with a fixed broadband connection. Thus, from web developer's perspective, being offline or having a slow connection, might seem like an error. Unfortunately, this leads to a horrible user experience for many people. A new approach to web development called *offline-first* aims to tackle this issue by focusing on the offline experience first before adding online features. [3]

1.1 Motivation

In 2014, we launched a mobile web application at our company for conducting audits, inspections, and assessments. One of the key features of the auditing application is its offline support since audits are often carried out in conditions where there is poor network connectivity or no network at all.

As we knew the importance of offline from the start, we implemented the auditing application using the offline-first approach. We used a JavaScript framework called Orbit at its core for the offline capability. Orbit was quite immature at the time, which caused us invent our own solutions, hacks, and workarounds for many problems that arise from the offline requirement.

In 2017, a new version of Orbit was released. The new version is a complete rewrite compared to the old. The new version has a more elegant design and solves many of our problems with the old version. However, due to the vast differences in the Orbit versions and the large codebase of our auditing application, we deemed it is too difficult to attempt to upgrade Orbit directly in the auditing application. Instead, we decided that it is better to create a small proof-of-concept application from a clean slate with the offline-first approach and the latest Orbit version. This proof-of-concept application would then serve as a reference where we try to steer the auditing application towards to.

1.2 Research Goals

Based on the previous section, it was defined that the goal of this thesis is to find answers to the following questions:

- What features are there in browsers that enable offline web applications?
- How can these features and Orbit be used to implement a web application with an offline-first approach?

This thesis thus has two parts: a theoretical and a technical. The theoretical part is a literature study that aims to answer the first question. In the technical part, a simple prototype web application is implemented to answer the second question.

Since the implementation is a prototype application, the following things were decided to be left out of the scope of this thesis. Firstly, only modern browser environments on a mobile phone are considered. Supporting older browsers would cause too much work, and desktop environments are not that interesting from the offline perspective. Secondly, user experience is not

considered or evaluated it in any way in this thesis. Thirdly, authentication is not considered in this thesis even though it is required feature for many real-world applications.

When there is more than one device that makes modifications to the same resource offline, it is inevitable that this will result in conflicting versions of the resource. The handling and resolving of these conflicts are not addressed in any way in this thesis.

1.3 Structure of the Thesis

The structure of this thesis is the following. Chapter 2 gives an introduction to the design paradigms related to this thesis, such as single-page application, progressive enhancement, and offline-first. Chapter 3 provides an overview of the technologies available in web browsers that make offline web applications possible.

Chapter 4 defines the requirements for the prototype, presents its overall architecture, and introduces Orbit. Chapter 5 presents the implementation of the proof-of-concept web application using the offline-first approach. After that, Chapter 6 evaluates the implementation.

Finally, Chapter 7 discusses the shortcomings and future development areas of the implementation. Chapter 8 concludes the thesis.

Chapter 2

Design Paradigms

2.1 Single-page Application

Traditionally, web pages have been multi-page interfaces. When Tim Berners-Lee invented the web in 1990, most web pages consisted mainly of static text and images. It was possible to connect web pages to each other via hyperlinks. When a user followed a hyperlink to another page, the browser would refresh the entire interface with the contents of the new page. Later on, server-side scripting languages such as PHP started emerging. They allowed web servers to generate web pages dynamically based on user interaction. In 1996, browsers implemented JavaScript as a client-side scripting language, which gave web developers the option to embed logic that ran on the browser to the web pages. [4]

A paradigm shift started around 2005 when Asynchronous JavaScript And XML (AJAX) started gaining traction. AJAX enabled web pages to make asynchronous HyperText Transport Protocol (HTTP) requests to the web server, and update only one part of the web page. Asynchronous requests do not block the User Interface (UI), which keeps the web page more responsive. Another revolution started around 2006, when two groups, World Wide Web Consortium (W3C) and Web Hypertext Application Technology Working Group (WHATWG), began working together on a new HyperText Markup Language (HTML) standard, HTML 5. The development of HTML had stagnated since the release of HTML 4 in 1997. The HTML 5 standard added a lot of missing features to the web platform that allowed web developers to create richer and more complex web applications. [4]

The evolution of the web led to new techniques for building web applications. Single-Page Application (SPA) is a web application, which uses a single HTML page that is not refreshed during use. Instead, SPAs use

JavaScript heavily for all user interactions and rely on AJAX when they need to communicate with the server. [4] SPAs can be seen as a hybrid of a traditional multi-page web application and a native desktop application. Compared to a traditional web application a SPA provides an overall better user experience. The response times are faster, since the server interactions happen asynchronously in the background, and less data needs to be transferred. Compared to a native desktop application a SPA is more portable. It works on any operating system as long a modern web browser is available. It is also easier to distribute. The user just has to reload the page, and it works whereas native applications often require administrative privileges to install. [5]

2.2 Progressive Enhancement

Progressive enhancement is a web design philosophy that Steve Champeon introduced in 2003. The idea is that a web page should be built bottom-up starting from the most basic browser, the lowest common denominator, and then progressively adding more advanced features for more capable browsers. This philosophy is in contrast to *graceful degradation*, where a web page is built top-down starting from the most capable browsers and then making them work in less feature-rich browsers. [6]

2.3 Offline-First

As mentioned in Chapter 1, we are using mobile devices more and more to access the Internet instead of desktop computers. Moreover, when we are using mobile devices, there will inevitably be times when we are offline. Still, web developers often take network connectivity for granted. In offline-first paradigm, this assumption is reversed.

Offline-first is like progressive enhancement, but focusing on network connectivity. First, a web application that works entirely offline is built. Then, the experience is gradually enhanced with online features.

Offline-first is quite new design paradigm for web applications. To my best knowledge, the term *offline-first* was used for the first time in 2012 by Lambert in his article entitled "Offline First – A better HTML5 User Experience". In the article, he points out that offline is a feature and it is important to consider it already at the beginning of the development process. He also presents three guidelines for developing offline-first applications from the technical perspective. (1) Move the entire application logic from the

server to client-side. Use the server only as a data storage and communicate using JavaScript Object Notation (JSON) between the client and the server. (2) Create a client-side abstraction layer for interacting with the server-side Application Programming Interface (API). (3) Create a data layer that coordinates requesting data from the server and caching it in browser storage. [7]

Feyerke suggests similar approaches as Lambert for the technical aspects of offline-first. First, the application logic should run mostly in the browser. Secondly, for a full offline experience, the client should store the data in the browser and be able to sync it to the server. However, Feyerke sees the user experience aspect as a more challenging problem, where the industry has not yet developed proper design patterns. He presents various scenarios, where a change in connectivity state raises issues, such as the following:

- It is frustrating for the user if they lose access to data when they suddenly go offline. Minimally, the application should keep the data available in read-only mode. It is preferable, though, if it is still possible to make modifications while offline.

- Offline should not be treated as an error. If the user is attempting to create something offline, the application should save it locally and inform the user that it will be sent later instead of showing an error message. If the user is viewing a news feed, the application should show old cached data with a corresponding message instead of showing an empty view.

- It is inevitable that there will be conflicting versions of data if the application allows editing data simultaneously on multiple devices. The application should provide an easy-to-use user interface for resolving the conflicts.

- There can be challenges in what order to display chronological items such as chat messages. If the application shows the messages in the order they are transmitted, they are easy to notice. However, it is confusing, when a message is a reply to something much older. On the other hand, if the application shows the messages in the chronological order, a recently transmitted message is not always the newest. It might appear somewhere the user does not expect, and the user may not even notice it. [3]

Sauble presents in his book principles for good offline design from the user's perspective. The principles are as follows:

- Provide as much content as possible to the user even when they are offline.
- Let the user view and edit the content regardless of their connection state.
- Use clear and user-friendly wording in error messages.
- Let the user complete the tasks they started even if they suddenly go offline.
- Get input from the user when the application cannot resolve a conflict automatically.
- Cache the most relevant data to the user first.
- In an empty state, tell the user what they can do next.
- Preserve the state of the application over sessions.
- Scale the network usage based on the available bandwidth.
- Do not clear the cache unless the user requested explicitly. [8]

Some of the principles are not strictly related to good offline experience. Some are just good user experience, such as the principles regarding the error messages and the empty state. However, overall, the principles do support the scenarios Feyerke recognized.

Chapter 3

Offline Web Technologies

3.1 Web Workers

Web workers are scripts that are run in the background independently from the web page and the UI. They are intended for heavy scripts that are expected to have long running time, high start-up performance cost, or high memory usage. They are run in a separate thread from the web page's scripts, which means that the web page stays responsive even if the web worker performs heavy calculations. [9]

Web workers do not have access to the web page. Instead, the web page and the workers communicate with each other by passing messages. [9]

There are two types of web workers: dedicated workers and shared workers. Dedicated worker can only communicate with its creator. Use cases for dedicated workers include executing tasks that might otherwise block the UI such as

- processor-intensive calculations (for example, computing prime numbers), and
- input/output (for example, polling search query results from a server on user's request). [9]

Shared workers are similar to dedicated workers, but in addition to its creator, they are accessible also by other scripts on the same origin. Thus, they can be used for sharing state between multiple windows. [9]

3.2 Service Workers

A service worker is a special kind of event-driven web worker. Similarly to web workers, service workers also run in the background in a separate

thread from the web page. They have been designed for use cases such as, caching assets, handling push notifications, and managing background data synchronization. [10]

Service workers provide means to intercept network requests that originate from the web page it is associated to. This gives web developers full control how the requests are handled. In addition, service workers include a cache API to enable serving content for offline use. Thus, it is possible to return a response to a request straight from a cache instead of making the request over the network. [10]

As mentioned in Section 3.4, service workers are intended to replace application cache as the means for providing an offline experience. However, the browser support for service workers is not yet as extensive as for application cache. Currently, Chrome and Firefox are the only major browsers that have implemented service workers. In Edge and Safari, the feature is under development. [11]

3.3 Web Background Synchronization

Web applications often need to synchronize client data with the server. However, if the user decides to navigate to another web page or to close the browser, the synchronization is interrupted. The application cannot continue with the synchronization until the user opens the web application again.

Web Platform Incubator Community Group (WICG) is currently working on a new specification that allows web applications to synchronize data in the background. The specification extends service workers with a *sync* event. A web page can request for background synchronization. Then, if the page is running, as soon as there is network connectivity the browser will fire a sync event. Otherwise, the browser will fire the event at their earliest convience. Then, a service worker registered with the web page can listen for these sync events and react accordingly to them. [12] Chrome is currently the only browser to implement the specification [11].

3.4 Application Cache

Application cache is a mechanism that allows users to continue using web applications and pages even when their network connection is not available. The web developers can provide a manifest file listing all the files needed by the application to work offline. The browser will then cache these files and make them available for offline use. [9]

Listing 3.1 presents an example manifest file, `manifest.appcache`, of a simple web application. The web application consists of an HTML page, `app.html`, a Cascading Style Sheets (CSS) stylesheet, `app.css`, and a JavaScript file, `app.js`. The first line of the manifest file must be `CACHE MANIFEST`. Otherwise, the browser ignores the manifest. Then, the following lines list the three files the application needs cached for offline use.

```
CACHE MANIFEST
app.html
app.js
app.css
```

Listing 3.1: An example of an application cache manifest file.

The web developer must link the manifest to the web application for the browser to use it. This is done by adding a `manifest` attribute in the `<html>` element in the web application's HTML pages. The manifest attribute's value is a Uniform Resource Locator (URL) pointing to manifest file. Listing 3.2 presents how this is done.

```
<!DOCTYPE html>
<html manifest="manifest.appcache">
  <head>
    <meta charset="utf-8">
    <title>Application Cache Example</title>
    <script src="app.js"></script>
    <link rel="stylesheet" href="app.css"></script>
  </head>
  <body>
    ...
  </body>
</html>
```

Listing 3.2: An example of an HTML page that uses application cache.

However, Archibald [13] has listed nine situations, where application cache's behavior may surprise developers. For example, the browser always serves the cached version of the page, even when the user is online. After the page has finished rendering, the browser will check whether there are updates available. This also means that the user must refresh the page to see the new version of the content.

Another example is that the application cache does not update the files listed in the manifest unless the contents of the manifest have changed. There are some workarounds for this. One option is to add a version number or hash of the file contents to filenames. Then, each time the file is updated, it is served under a different URL, which requires a change to the contents of the manifest. This option works well for static assets such as JavaScript, stylesheets, and images. Another option is to add a comment to the manifest for a cached file and change that whenever the cached file is updated. This

option works for resources such as HTML pages that must always be served under the same URL. It is thus recommended to automate the manifest generation to a build tool rather than edit it manually. [13]

It is also not possible to have conditional downloads of resources. Consider that there is a responsive image, which has two different versions: a small and light for mobile devices, and a big and heavy for desktop devices. If the page uses application cache, both versions of the image must be listed in the manifest, and the browser downloads them both. Thus, it makes more sense to just use one version of the image for all devices. [13]

All major browser vendors support application cache [14]. On the other hand, the WHATWG has deprecated the feature in the HTML living standard. They are currently in the process of removing it completely from the standard. Instead, they recommend developers to use service workers. [9]

3.5 Client-Side Storage

This section gives an overview of the APIs available for web applications to store data on the client. First, the two main storage APIs — Web Storage and Indexed Database API (IndexedDB) — are covered. Then, a comparison regarding their features is presented.

There is also a third storage API: Web SQL Database (WebSQL). However, W3C has stopped working on the specification [15]. In addition, many major browsers, such as Internet Explorer, Edge, and Firefox, do not support it [14]. Thus, this thesis will not address it any further.

3.5.1 Web Storage

Web storage is an API part of the HTML specification. It allows web applications to store key-value pairs in client-side. Both keys and values are always strings. [9] Web applications can store more complex values as well, if they serialize the values to strings using JSON or some other serialization format.

Web storage includes two storage mechanisms, which both have similar APIs:

- `sessionStorage` keeps the stored key-value pairs for the duration of the browser session. Usually, the browser session ends when the browser is closed.
- `localStorage` has a more permanent lifetime than `sessionStorage`. It can maintain the stored key-value pairs even when browser is closed and opened again. [9]

Each origin have their unique instances of `sessionStorage` and `localStorage`. The HTML specification recommends that browsers should limit the available storage by origin. It suggests a limit of 5 MB, but the exact limit is up to the browser to decide. [9]

3.5.2 Indexed Database

IndexedDB is an API for storing structured data. It allows web applications to store records locally in the browser. Each record consists of a key and a value. The values can be of any type that is supported by the structured clone algorithm. This includes types such string, object, file, blob, array, and date. [16]

IndexedDB supports indexes, which makes it possible to do efficient searches over the records in the data store. IndexedDB also supports transactions, which are used to interact with the data in the database. [16]

3.5.3 Comparison

Table 3.1 summarizes the features of `sessionStorage`, `localStorage` and IndexedDB. Both web storage and IndexedDB have broad support in browsers. According to *Can I use... Support tables for HTML5, CSS3, etc.* web storage is supported by 94 % of the browsers based on their global market share. IndexedDB has a slightly worse support of 93 %. [14]

Feature	**sessionStorage**	**localStorage**	**IndexedDB**
Browser Support	94 %	94 %	93 %
Persistence	Session	Best-effort	Best-effort
Sync/Async	Sync	Sync	Async
Web Workers	No	No	Yes

Table 3.1: Summary of client-side storage features.

Persistence refers to how long the stored data stays in the browser. For `sessionStorage`, the browser stores the data for the duration of the session. Both `localStorage` and `IndexedDB` are best-effort in persistence, which means that the browser keeps the stored data over browser sessions, but the browser may decide to clear the data automatically [9, 16]. For example, both Firefox and Chrome start clearing the stored data, when running out of storage space [17, 18].

The web storage has synchronous API, and IndexedDB has asynchronous API [9, 16]. Thus, using web storage blocks the user interface of the web page while using IndexedDB does not. Also, a web worker script can access an IndexedDB data store, but not web storage [9, 16].

Table 3.2 summarizes the storage space limits of the three storage APIs. In Chrome, Firefox and Edge, the storage limit for both `localStorage` and `sessionStorage` is 10 MB per origin [19, 20]. Safari is the only one that uses the limit of 5 MB per origin [19] as recommended by the HTML specification.

Browser	**sessionStorage**	**localStorage**	**IndexedDB**
Chrome	10 MB	10 MB	up to ∼7 % of free space
Firefox	10 MB	10 MB	up to 10 % of free space
Safari	5 MB	5 MB	unknown
Edge	10 MB	10 MB	10 MB to 500 MB

Table 3.2: Browser limits for client-side storage.

For IndexedDB, both Chrome and Firefox use a dynamic storage quota based on the available free disk space [18, 17]. In Chrome, there is a shared pool of up to one-third of available disk space for all web apps. Of this shared pool, each web app can use up to 20 %. Thus, a single web app can use approximately up to 7 % of free space in Chrome [18]. In Firefox, there is a global limit of 50 % of free disk space for all websites. Of this global limit, each domain can use up to 20 %. Thus, a single domain can use up to 10 % of free space in Firefox. [17] In Edge, the storage quota is also dynamic but based on the volume size. Each domain can use up to 500 MB of space. [21] For Safari, I could not find a reliable source for IndexedDB storage limits.

Chapter 4

Design Overview

This chapter presents a design overview for the proof-of-concept web application. I chose to implement a simple task management application as the proof-of-concept. This chapter begins with a section that defines the requirements for the application. Then, the following sections describe the architecture of the application and give an overview of the used technologies.

4.1 Requirements

The following requirements were recognized for a simple task management application:

- **Requirement 1. List tasks**

  The user must be able to see their tasks so that they know what they need to do.

- **Requirement 2. Create a task**

  The user must be able to add tasks so that they remember to do them later.

- **Requirement 3. Rename a task**

  The user must be able to change the name of a task so that they can correct any mistakes they may have made.

- **Requirement 4. Complete a task**

  The user must be able to mark their tasks completed so that they can remain focused on their unfinished tasks.

- **Requirement 5. Delete a task**

  The user must be able to remove tasks so that they can clear tasks, which they do not need anymore, or which they accidentally added.

Based on the literature review on offline-first in Section 2.3, the following requirements were identified for an offline web application:

- **Requirement 6. View tasks offline**

  The user must be able to see their tasks offline so that they do not get frustrated and can know what they need to do next even when they suddenly lose the connection.

- **Requirement 7. Edit tasks offline**

  The user must be able to make modifications to their tasks offline so that they can remain productive regardless of their network connection.

- **Requirement 8. Preserve tasks**

  The application must preserve everything the user types into the application so that the user does not have to worry about accidentally losing their data if they close the application or refresh the page in the browser.

- **Requirement 9. Synchronization**

  The application must synchronize the tasks to all user's devices so that the user can plan their things-to-do using whatever device they have at hand.

- **Requirement 10. Optimistic UI**

  The application must feel fast regardless of the network quality so that the user does not get frustrated waiting even if they have a slow connection.

- **Requirement 11. Browser support**

  The application must work with the latest versions of Safari on iOS and Chrome for Android. As the application is developed using an offline-first mindset, I decided to support the browsers of the two largest mobile platforms. Moreover, as the application is a prototype, I decided to leave out support for older browsers in the scope of this thesis.

4.2 Architecture

Figure 4.1 presents an overview of the application architecture. The system consists of two components: a client and a server. Based on the offline-first guidelines in Section 2.3, I decided to implement the client as a SPA. I wrote the client in TypeScript language, which is a superset of JavaScript with static type-checking. The client uses Angular for its web framework and Orbit for its data layer. Understanding Orbit's concepts and terminology are crucial to understanding the implementation in the next chapter. Therefore, Section 4.3 describes Orbit in-depth later. I also used a tool called Angular CLI to speed up the development process and to provide a Webpack-powered build pipeline.

To synchronize the tasks to multiple devices, the server needs to act as a central storage for the tasks. The server provides the client an API for retrieving and storing tasks. Orbit includes built-in support for remote servers that implement the JSON API 1.0 specification. The JSON API specification defines a set of conventions and rules for building APIs that use JSON as their data serialization format [22]. Therefore, I decided to make the server adhere to the JSON API v1.0 specification.

I implemented the server in Python programming language. It uses Flask for its web framework, a Flask extension called Flask-REST-JSONAPI to build an API compliant with the JSON API v1.0 specification, SQLAlchemy for its object-relational mapper, and PostgreSQL for its database.

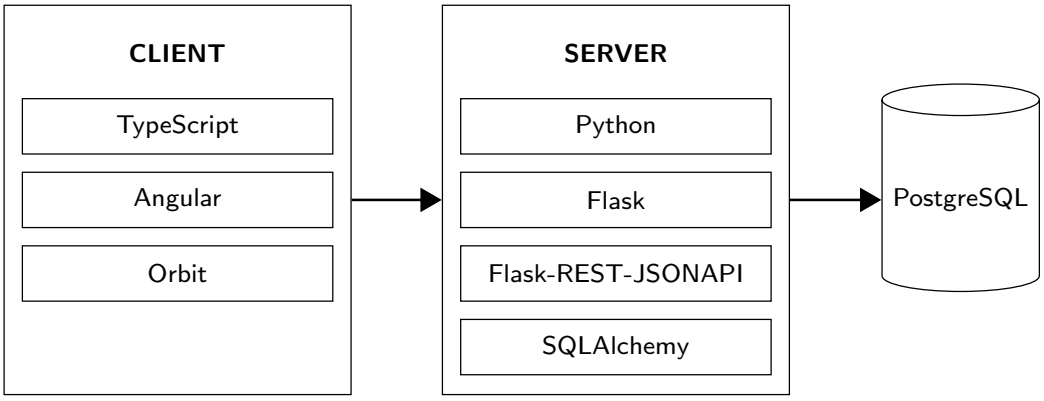

Figure 4.1: Overview of the application architecture.

4.3 Orbit

Orbit is an open source framework for "orchestrating access, transformation, and synchronization between data sources." [23] Orbit is developed and maintained by Dan Gebhardt from Cerebris Corporation. The framework is based on the following key concepts:

- A *record* represents a data instance. It has an identity, which consists of an id and a type. It may also contain attributes, which represent information about the record, and relationships, which are links to other records.
- A *schema* defines the models of the domain. Each model, in turn, represents a single record type and defines what attributes and relationships that the record has. For example, a model of an article might state that each article has attributes such as title and content and relationships such as author and comments.
- *Sources*, such as an in-memory store, a remote server, or an IndexedDB database, provide access to records. A source may offer various interfaces to the data depending on its type. For instance, some source may allow to query and update records, whereas another source may just notify of changes to records.
- *Queries* make it possible to retrieve records from sources based on specific criteria. Orbit provides tools for applying various refinements, such as filtering and sorting, to queries.
- A *transform* can be applied to a source to modify its contents atomically. A transform consists of a set of operations. Each operation represents a single change to a record or a relationship. For instance, adding a record, updating an attribute, and deleting a relationship, are examples of an operation.
- A *bucket* is a key-value store that the application can use for persisting its state. For example, sources use a bucket to store their internal task queues and transformation history.
- A *coordinator* observes a set of sources and applies coordination strategies to them. The use cases for coordination strategies include keeping the contents of sources synchronized, logging source events, and handling of errors.

Chapter 5

Implementation

5.1 Client

This section explains how the client component of the application was implemented. As the goal of this thesis was to build a web application using offline-first approach, this section describes the implementation from the perspective of the offline-first process. Sections 5.1.1 to 5.1.4 describe how the offline-only functionality of the application was implemented. Then, Sections 5.1.5 to 5.1.7 focus on how the offline-only application was enhanced with the online features.

5.1.1 User Interface Skeleton

The first step in implementing the application was to build a UI skeleton. UI is a mandatory part of any web application, but since the focus of this thesis is not in the UI, only the outlines of the UI implementation are covered here.

The UI uses Angular Material[1] library, which provides components that adhere to Google's Material Design[2] guidelines. Figure 5.1 presents a screenshot of the final UI, which consists of three components. First, there is the application header at the top of the screen, which just states the application name. Then, there is a text input for creating new tasks (Requirement 2). Finally, there is the task list, which shows each task as a task item (Requirement 1).

Each task item is composed of three parts. The checkbox is for displaying whether the task has been completed and for marking the task completed (Requirement 4). The label shows the task description and clicking it opens an inline edit for editing the description (Requirement 3). The button on the right

[1] `https://material.angular.io/`
[2] `https://material.io/`

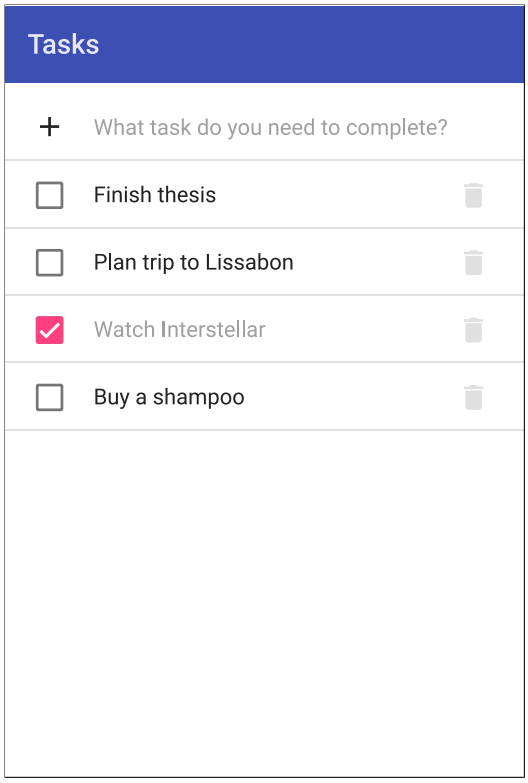

Figure 5.1: A screenshot of the application UI.

represented by a trash icon allows the user to delete the task (Requirement 5).

5.1.2 Storing and Querying Tasks

The UI skeleton was just using dummy data. In the next phase, a data store was introduced to the application to finish the implementation of Requirements 1 to 5, which were started in the previous section. The data store is an in-memory cache that allows the application to store tasks and query them. Thus, it also enabled the user to view and edit tasks offline (Requirements 6 and 7).

Listing 5.1 presents a code snippet of defining the data store and the schema associated with it. The schema defines one model, `task`, and its attributes. Each task has a title (for example, "Plan the trip to Lisbon"), a boolean flag indicating, whether the task has been completed or not, and a timestamp, when the task was created. The timestamp is assigned to a task, when it is created in the client. It is used in the task listing to keep the tasks

```
const schema = new Schema({
  models: {
    task: {
      attributes: {
        title: { type: 'string' },
        isCompleted: { type: 'boolean' },
        createdAt: { type: 'date' },
      },
    },
  },
});
const store = new Store({ schema });
```

Listing 5.1: Defining schema and store for the application.

in consistent order and to have the most recent tasks appear at the top of the task list.

Additionally, a live query module was introduced to the application. Live query is a query that returns a new result-set whenever the store's data changes. The task list uses the live query module to display the tasks in the store, which simplifies the process of keeping the UI updated. Figure 5.2 presents a high-level overview of how the live query works. The horizontal arrows in the figure represent data streams where time flows from left to right. The data stream at the top part symbolizes a stream of transform events, which in this case are t_1, t_2, and t_3. Each transform event, in turn, represents a change to the data store, such as an added task. The data stream at the bottom part illustrates a stream of query results (r_0, r_1, r_2, and r_3), which is the output of the live query. When invoking live query with query q, it first executes q immediately against the current state of the data store. The live query emits the results of this initial query as result-set r_0. Then, the live query starts observing on the transform stream. Then, whenever a transform event, such as t_1, happens, the live query executes q against the data store and then emits the corresponding result-set, such as r_1 in the case of t_1.

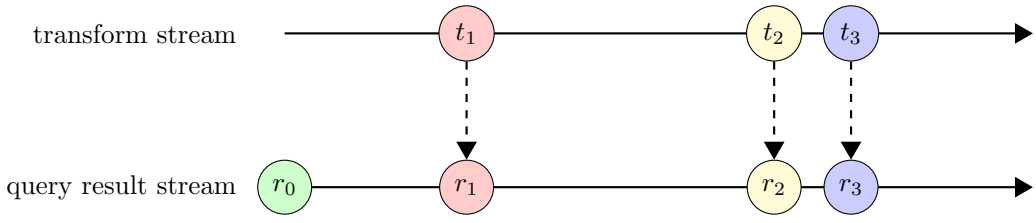

Figure 5.2: An overview of how a live query operates.

5.1.3 Persisting Data to Browser Storage

A task management application is of little avail if the application loses all data as soon as the user closes or refreshes the browser tab that has the application open. To prevent this data loss, the application needs to persist the data in the browser (Requirement 8). The data itself consists of two types: records and source-specific state. In this application's context, the records consist of just the tasks. Source-specific state includes things such as the pending transforms and queries.

As was pointed out in Section 3.5, there are two options for persisting data in the browser: `localStorage` and IndexedDB. Based on the comparison in Section 3.5.3, IndexedDB was chosen as it has several advantages over `localStorage`. First of all, it is asynchronous, so it does not block the UI. Secondly, it has larger storage limit, so the application can store more data in the client-side cache. Lastly, it works in a web worker script, which is useful if background synchronization is implemented later.

To persist the records, a new source called *backup source* that uses IndexedDB was introduced to the application. To synchronize the in-memory store with the backup source, a coordinator and a coordination strategy that observes the store for changes and reflects them to the backup source were added to the application. Correspondingly, when the application starts, the application restores the store contents from the backup source. To persist the source state, an IndexedDB bucket was created and assigned to both the store and the backup source.

5.1.4 Accessing Application Offline

There was one final issue before the application could work fully offline. If the user opened the application while offline, the browser greeted them with a "no internet connection" error. To solve this, there were two options: service worker and application cache.

Application cache has been deprecated in favor of service workers as mentioned in Section 3.4. Therefore, a service worker was added to the application at first. Since the application was using Angular CLI, enabling service worker support for the application was rather easy. There was only a need to flip one boolean flag, `serviceWorker`, in Angular CLI's configuration.

Unfortunately, Safari does not yet support service workers. Thus, the addition of service worker was not enough to satisfy all supported browsers (Requirement 11). Hence, application cache support was needed in the application as well. As mentioned in Section 3.4, it is recommended to automate the generation of the application cache manifest file. Therefore,

a Webpack plugin called `appcache-webpack-plugin`[3] was used to generate the manifest file during the build process. Then, a library called `appcache-nanny`[4] was used to link the manifest to the application. `appcache-nanny` works around many of the issues in application cache mentioned Section 3.4. Finally, before enabling application cache, the application checks that the browser does not support service workers so that it does not use application cache unnecessarily.

5.1.5 Synchronizing with the Server

As the application now worked fully offline, the implementation of the synchronization with the server (Requirement 9) could begin. Section 5.2 later in this chapter describes the server implementation in detail, but from this point forward this section assumes that there is a working server.

First, a new Orbit data source called *remote source* for communicating with a remote server was defined. Both the remote source and the server implement the JSON API specification as mentioned in Section 4.2. Then, new strategies to synchronize the remote source with the other sources were introduced. These strategies are explained using two scenarios: the store is queried and the store is updated.

Figure 5.3 presents a sequence diagram showing how these strategies coordinate the data sources in the first scenario: when the store is queried. The sequence is as follows: (1) The main loop of the application queries the store with query q. (2) The store triggers a *beforeQuery* event. (3) A strategy called *store-remote-query-optimistic* listens for the *beforeQuery* events in the store. (4) As the name implies, this strategy executes q optimistically on the remote source. (5) The remote source makes an HTTP request according to q to the server. (6) As q was executed optimistically on the remote source, the store does not wait for the results from the remote source. Instead, the store runs q on its cache and returns the results to the main loop. (7) The HTTP request finishes. The remote source parses it to a transform t and triggers a *transform* event. (8) Another strategy listens for the *transform* events in the remote source. (9) This strategy synchronizes all transforms on the remote source to the store. (10) The strategy described in Section 5.1.3 persists the changes in the store to the backup source.

Similarly, Figure 5.4 presents a sequence diagram showing how these strategies coordinate the data sources in the second scenario: when the store is updated. The sequence is as follows: (1) The main loop of the application

[3] `https://github.com/lettertwo/appcache-webpack-plugin`
[4] `https://github.com/gr2m/appcache-nanny`

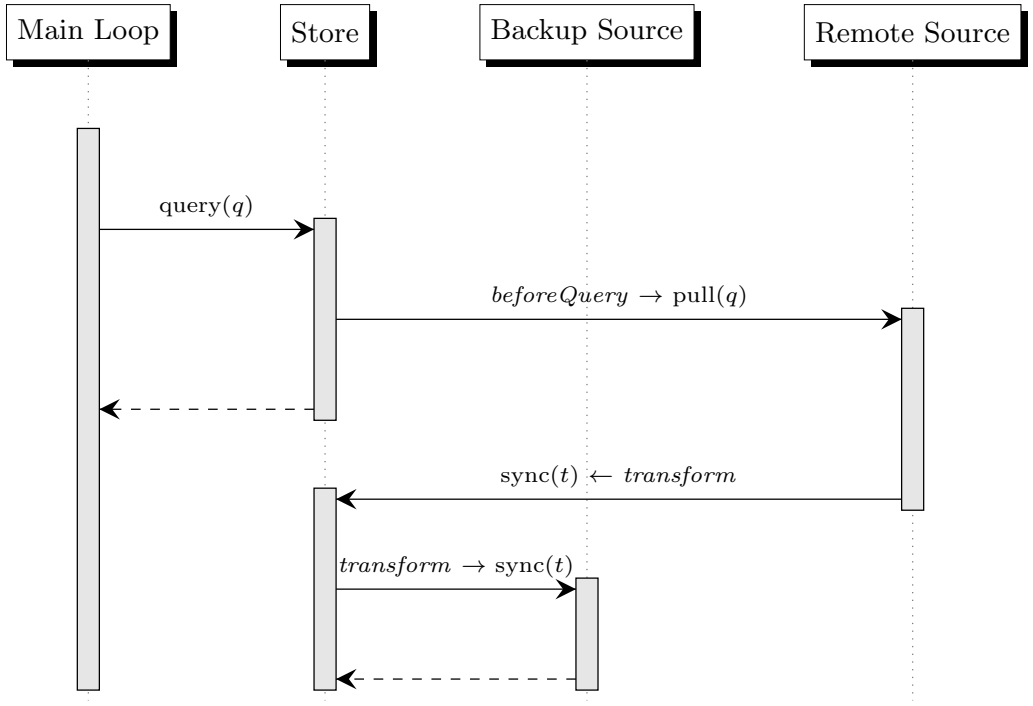

Figure 5.3: Sequence diagram of data source coordination when querying the store with query q.

updates the store with transform t. (2) The store triggers a *beforeUpdate* event. (3) A strategy listens for the *beforeUpdate* events in the store. (4) This strategy executes t optimistically on the remote source. (5) The remote source makes an HTTP request according to t to the server. (6) As t was executed optimistically on the remote source, the store does not wait for the results from the remote source. Instead, the store applies t on its cache and returns the results to the main loop. (7) The strategy described in Section 5.1.3 persists the changes in the store to the backup source. (8) The HTTP request finishes. If the server did additional updates besides t, the remote source composes another transform t' from them based on server's response and triggers a *transform* event. (9) From now on, the coordination process continues similarly as in the store query scenario described previously.

5.1.6 Handling Failures

Several things can go wrong when the remote source communicates with the server. Therefore, an introduction of two strategies for handling these errors

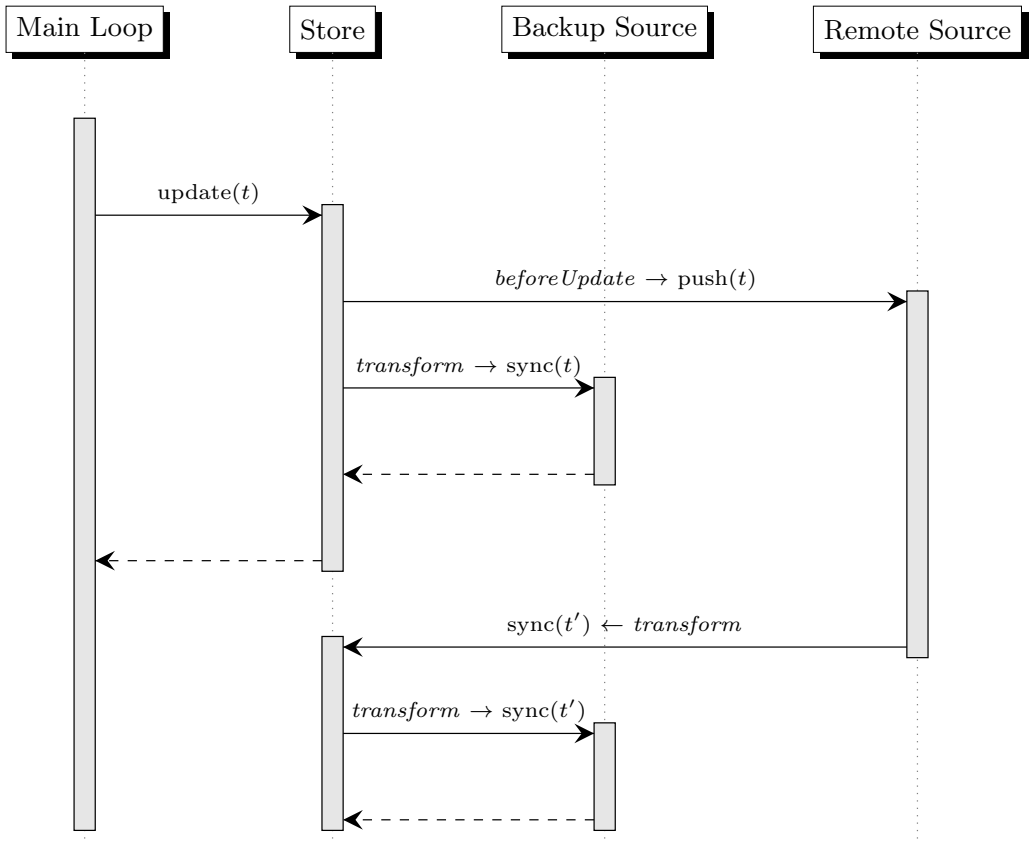

Figure 5.4: Sequence diagram of data source coordination when updating the store by applying transform t to it.

was needed. These strategies are called *remote-push-fail* and *remote-pull-fail*. As the names imply, the former handles failures when *pushing* to the remote source, and the latter handles failures when *pulling* from the remote source.

With respect to the *remote-push-fail* strategy, Figure 5.5 presents a flowchart of what the application does when a push of transform t to the remote source fails.

First of all, there are three main types of errors that can happen: network errors, server errors, and client errors. Network errors occur when the remote source is unable to get a response from the server, for example when the application is offline. Server errors happen when the server responds with HTTP status in the 500 to 599 range. They may indicate, for instance, that there was a programming error on the server that caused an unexpected exception (500 Internal Server Error), or that the server is temporarily

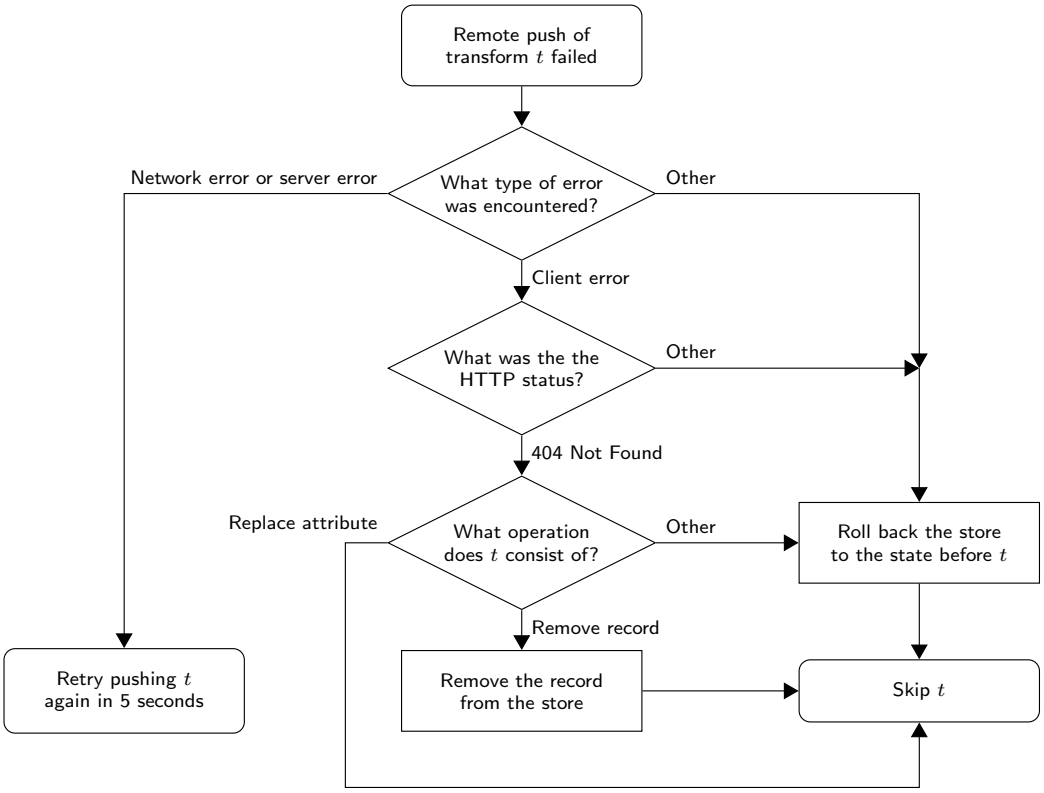

Figure 5.5: A flowchart demonstrating how the application handles failures when a change is pushed to the remote server.

unavailable (503 Service Unavailable). Client errors happen when the server responds with HTTP status in the 400 to 499 range. They indicate that there was a problem with the HTTP request.

If the error was either a network error or a server error, the application retries pushing the transform t again in five seconds. The reasoning for the retry is that the application assumes that both error types are temporary. Therefore, retrying them again later should eventually be successful. It is quite obvious that network errors are temporary: the application just has to wait until the Internet connection is available again. Server errors are trickier than network errors. The application assumes that time will fix all the server errors as well. On the other hand, one could also argue that not all server errors are temporary in nature.

Regarding client errors, there is a special handling for 404 Not Found status code. This error can happen in two scenarios: when trying to remove a task that does not exist on the server; or when trying to replace an attribute

of a task that does not exist on the server. In both scenarios, the reason why the task does not exist anymore on the server is that another client has deleted the task. In the former scenario, the application can just skip pushing the transform t to the remote source, since the server is already in the state it would be after applying t. In the latter scenario, it was decided that the deletion wins over the update of a task for simplicity's sake. Thus, the application removes the task in question in t from the store and then skips pushing the transform t to the remote source.

If any other error occurs, the application assumes that it cannot handle it properly. Therefore, it rolls back the store to the state before t, so that the remote source can continue processing other transforms.

Regarding the *remote-pull-fail* strategy, there is only one error with special handling: 404 Not Found on find record query. This HTTP status indicates that the record does not exist on the server, so the application removes the record in question from the store as well. On all other errors, the application just skips the query.

5.1.7 Evicting Deleted Tasks

There was one issue with the synchronization related to deleted tasks. If somebody removes a task with one device, the task remains in other devices' caches. The user must first try to edit or delete the task so that the remote fail strategies described in Section 5.1.6 come into play.

This problem is tackled with a new strategy. The application executes this strategy whenever the application pulls all tasks from the remote source. Then, it compares the tasks the server returned to the tasks in the store. The strategy marks each task that is in the store, but not in the server, as an eviction candidate. There are two reasons the task might exist in the store, but not in the server: either the task is new, and the application has not yet synchronized it to the server, or some other client deleted the task from the server. To find out if the reason is the latter, the strategy queues up a find record query to the store for each eviction candidate. If the server returns a 404 Not Found response for the eviction candidate, the client removes the task from the store.

This strategy works because of the following reasons: (1) The store is a source. (2) Sources use the same request queue for both queries and transforms. (3) Sources process the request queue serially. (4) Thus, it is guaranteed that the store executes the find record query after the possible transform that creates the task. (5) The *remote-pull-fail* strategy described in Section 5.1.6 removes the task from the store if the find record query does not find the task.

Parameter	Purpose
`page[number]`	A number indicating which page is returned.
`page[size]`	A number indicating how many tasks are returned per page.
`sort`	A comma-separated list of attributes the task collection is sorted by. Each attribute may be prefixed with a minus for descending order, otherwise the order is ascending.

Table 5.1: List of query parameters supported by the `GET /tasks` endpoint.

5.2 Server

This section provides an overview of the server implementation. As mentioned in Section 4.2, the server implements the JSON API v1.0 specification. This section is divided into two parts. The first part describes what resources the API consists of. Then, the second part describes the available API endpoints briefly.

5.2.1 Resources

The API consists of only a single resource, *tasks*, which corresponds to the client-side task model defined in Section 5.1.2. A task has three attributes. `title` is a string that describes the task (for example, "Book flights to Lisbon"). `is_completed` is a boolean flag that indicates whether the task has been done or not. `created_at` is an ISO 8601 formatted string representing the time when the task was created.

5.2.2 Endpoints

The server has API endpoints for fetching, creating, updating, and deleting tasks. The client can fetch a collection of tasks by making a `GET` request to the `/tasks` endpoint. The server responds to a successful request with a `200 OK` response. Listing 5.2 shows an example of such response. The server also supports query parameters for sorting and paginating the collection of tasks. Table 5.1 lists these parameters and explains their purposes. The client can also fetch an individual task by making a `GET` request to the task's URL. The task URLs follow the format `/tasks/:id`, where `:id` is the task's Identifier (ID).

The client can create a task by sending a `POST` request containing the task

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "type": "tasks",
      "id": "5d94fdfc-1a59-49b7-9f17-e58a9ba69885",
      "attributes": {
        "is_completed": false,
        "created_at": "2017-07-26T09:15:53.865000Z",
        "title": "Buy groceries"
      },
      "links": {
        "self": "/tasks/5d94fdfc-1a59-49b7-9f17-e58a9ba69885"
      }
    },
    {
      "type": "tasks",
      "id": "01f0c474-d8d3-464e-84f2-6efafca599b2",
      "attributes": {
        "is_completed": true,
        "created_at": "2017-07-26T09:15:24.589000Z",
        "title": "Book flights to Lisbon"
      },
      "links": {
        "self": "/tasks/01f0c474-d8d3-464e-84f2-6efafca599b2"
      }
    }
  ],
  "links": {
    "self": "/tasks"
  },
  "meta": {
    "count": 2
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```

Listing 5.2: An example response for a `GET` request to a collection of tasks.

resource in the request body. The `POST` request must contain at least the title for the new task. Other attributes are optional. It may also include a client-generated ID, which is specified using the `id` key and which must be a Universally Unique Identifier (UUID). If the server created the requested task successfully, the server returns a `201 Created` response.

The client can update a task by sending a `PATCH` request to the task's URL. The `PATCH` request may update any or all of the task's attributes. If the update is successful, the server responds with a `200 OK` response.

The client can delete a task by sending a `DELETE` request to the task's URL. If the deletion is successful, the server responds with a `204 No Content` response.

Chapter 6

Evaluation

This chapter determines if the implementation presented in Chapter 5 meets the requirements defined in Section 4.1 by using manual testing. First, Section 6.1 describes the test devices, which were used to perform the tests with. Then, Section 6.2 introduces the test cases the implementation was exercised against. Finally, Section 6.3 presents the test results.

6.1 Test Environment

The tests were performed against two mobile devices: one iPhone and one Android phone. These devices were chosen to confirm that the implementation works with the browsers listed in Requirement 11. Table 6.1 presents the details of each device's model, operating system, and browser. Each test device had an Internet connection via a wireless network. Also, before each test, the browser's cache was cleared to have a clean state.

Device	**Operating System**	**Browser**
Apple iPhone 6	iOS 10.3.3	Safari 10
LG Nexus 5X	Android 7.1.2	Chrome 60.0.3112.107

Table 6.1: Overview of the test devices

6.2 Test Execution

Manual testing was used to confirm that the functionality defined in Section 4.1 was implemented correctly. Each manual test case consists of a short title, a

description, preconditions that must be true before running the test, and the steps the tester needs to follow to see if the application works as expected.

Some tests require another device in addition to the Device Under Test (DUT). The tester uses the second device for verifying that the data has synchronized properly. Since there were only two test devices, when iPhone was the DUT, Nexus was used as the second device and vice versa.

Test Case 1: Opening the Application Offline

The purpose of this test is to verify that the application can be opened in offline, which is implicitly required by Requirements 6 and 7. First, the application is opened in online so that the browser has the chance to cache the application for offline use. Then, the device is put offline, and the page is reloaded.

Preconditions

None.

Steps

1. Open the application in the browser.
2. Verify the task list is displayed.
3. Turn on the airplane mode.
4. Refresh the page.
5. Verify the task list is displayed.

Test Case 2: Create a Task

This test case tests that creating a task works properly to validate Requirement 2. First, a task is created in the DUT. Then, the second device is used to verify the task was synchronized to the server as required by Requirement 9.

Steps

1. Open the application in the browser in the DUT.
2. Tap on the “What task do you need to complete?” text field.

3. Type the title for the task.
4. Press Return.
5. Verify the new task is displayed in the task list.
6. Open the application in the browser in the second device.
7. Verify the new task is displayed in the task list.

Test Case 3: Rename a Task

This test case tests that renaming a task works properly to verify Requirement 3. First, a task is renamed in the DUT. Then, the second device is used to verify the rename of the task was synchronized to the server as required by Requirement 9.

Preconditions

There is at least one task in the task list.

Steps

1. Open the application in the browser in DUT.
2. Pick a task from the task list.
3. Tap on the task.
4. Verify the title of task turns into a text field.
5. Type the new title for the task.
6. Press Return.
7. Verify the task is displayed in the task list with its new title.
8. Open the application in the browser in the second device.
9. Verify the task is displayed in the task list with its new title.

Test Case 4: Complete a Task

The goal of this test is to confirm that completing a task works properly to verify Requirement 4. First, a task is marked completed in the DUT. Then, the second device is used to verify the change was synchronized to the server as required by Requirement 9.

Preconditions

There is at least one uncompleted task in the task list.

Steps

1. Open the application in the browser in the DUT.
2. Pick a task from the task list.
3. Tap on the checkbox of the task.
4. Verify the checkbox is checked.
5. Open the application in the browser in the second device.
6. Verify the task's checkbox is checked.

Test Case 5: Delete a Task

This test verifies that deleting a task works as required by Requirement 5. First, the application is opened both in the DUT and the second device to make sure they have the tasks in their caches. Then, a task is deleted in DUT. Finally, the page is refreshed in the second device to confirm that the task deletion has synchronized properly to the server and that the deleted task is properly purged from the cache.

Preconditions

There is at least one task in the task list.

Steps

1. Open the application in the browser in the DUT.
2. Open the application in the browser in the second device.
3. Pick a task from the task list in the DUT.
4. Tap on the task's delete icon.
5. Verify the task is not displayed in the task list anymore.
6. Refresh the page in the second device.
7. Verify the task is not displayed in the task list anymore.

Test Case 6: Short Offline Period

The purpose of this test is to verify if the application retries synchronization when the device goes offline for a short while. First, the DUT is put into airplane mode to simulate offline state. Then, a task is created, and the airplane mode is disabled. Finally, the second device is used to verify that the task was synchronized properly to the server.

Preconditions

None.

Steps

1. Open the application in the browser in the DUT.
2. Turn on the airplane mode.
3. Create a task as in Test Case 2.
4. Turn off the airplane mode.
5. Open the application in the browser in the second device.
6. Verify the task is displayed in the task list.

Test Case 7: Long Offline Period

This test case is very similar to Test Case 6. The difference is that this test case simulates a scenario where the user opens the application, makes some changes while offline, and leaves. Then at some point later, the user opens the application again, this time online. The application should then synchronize the changes made earlier to the server.

Preconditions

None.

Steps

1. Open the application in the browser in the DUT.
2. Turn on the airplane mode.
3. Create a task as in Test Case 2.
4. Refresh the page.
5. Turn off the airplane mode.
6. Open the application in the browser in the second device.
7. Verify the task is displayed in the task list.

6.3 Test Results

Table 6.2 presents the results of running the manual test cases that the previous section described. Both devices passed all the test scenarios. Thus, based on the manual testing the implementation is working as intended.

However, it is worth noting that these tests are not able to reveal all possible problems related to error handling or network connectivity in the implementation. A more extensive automated test suite that simulates various network conditions could be developed in the future to accommodate this shortcoming.

Test Case	iPhone	Nexus
1	Pass	Pass
2	Pass	Pass
3	Pass	Pass
4	Pass	Pass
5	Pass	Pass
6	Pass	Pass
7	Pass	Pass

Table 6.2: Results of the manual test runs.

Chapter 7

Discussion

This work focused on investigating how it is possible to implement an offline-first web application using Orbit. Based on the testing in Chapter 6, the implementation was able to meet all the requirements set in Section 4.1. However, there are still many shortcomings and areas for improvement in the implementation, which this chapter discusses.

First of all, the prototype is a simple application that consists of only a single text-based resource, tasks. In the real world, applications are more complex. They have many more resources, and they can also consist of other data types than text, such as images, video, and audio. These data types are much more challenging from the perspective of an offline application since they require much more storage space, which is limited in the browser environment. Thus, regarding a real-world application, the application might have to prioritize what data is cached for offline use first and what happens if the application runs out of storage space. Due to the simplicity of the prototype, this thesis did not consider these issues.

Also, due to the application being a simple prototype, the implementation did not consider the handling of software updates. The browser caches the client-side code and static assets of the application. Therefore, unless the client has an explicit mechanism to force itself to update to the latest version, the application developers must take into account that there might be older clients in use for a long time after an update. Thus, the developers need to take extra care if they introduce backwards-incompatible changes to the API in the server so that the older client versions do not break. One possible solution to this is to implement API versioning. Another aspect regarding software updates is schema changes. Since the browser caches the application data, there must be some mechanism in the client code to perform schema upgrades to this cached data.

Regarding the synchronization, the application fetches updates from the

server only when the application starts, that is, when the user loads the page. Thus, if a user has the application open on one device, and another device updates something, the update does not appear in the first device until the user refreshes the page. There are also two other problems related to how the client fetches the tasks from the browser. Firstly, the client fetches all the tasks from the server even if it has cached some of them already in the browser. The client could significantly reduce the amount of data transferred by fetching only the changes since the last update. Secondly, the client fetches all the tasks in one request, which can add a lot of stress to both the server and the client concerning the response size. This is especially apparent if the network connection speed is slow. The client could reduce this stress by using pagination to split the single response to multiple responses.

Furthermore, if the user makes changes in the application while offline, the changes are not synchronized to the server until the user has the application open while online. This fact might be surprising to the user, especially since the application does not have any indicator for unsynchronized changes. The user might close the application while there are unsaved changes, and not open it again for weeks. The background synchronization feature presented in Section 3.3 could be a solution to this issue.

Regarding the live query explained in Section 5.1.2, I noticed that the implementation is not very efficient. The live query feature executes a query against the whole contents of the store whenever there is any change to the store contents. Thus, when the application fetches the tasks from the server, the live query executes the query for as many times as there are tasks. The more there are tasks, the more apparent this problem is. A more efficient solution could just compare if the change (for example, an addition of a new task) would match the given query, and then return the difference to the previous result set. Another approach could be to implement a buffering feature so that if there are many changes in a short period, the live query will execute the query only once after the last change.

As explained in Section 1.2, I decided to leave conflict handling out of the scope of this thesis. Thus, the application does only the minimum regarding conflict management by handling the common failure cases when synchronizing with the server as described in Section 5.1.6. If there are any conflicting changes, the last changeset to a task that is synchronized always overrides the previous changes. The only exception is if another client deleted the task in question in which case the deletion wins. There are various different algorithms for properly detecting and resolving conflicts, such as operational transformation [24] and differential synchronization [25]. This topic needs further research in the future to see what would be the most feasible conflict management strategy regarding this implementation.

There is a bug in the server, which happens if a client attempts to create a task with an ID that already exists. The server responds to such request with 500 Internal Server Error response, which is a bug in Flask-REST-JSONAPI. A JSON API compliant server would instead respond with 409 Conflict response. This kind of a situation can happen, if a client sends a request to create a task, the server receives and processes the request, but the response does not for some reason like a network problem arrive at the client. Then, when the client attempts the request again later, the request fails on the server, since the task already exists.

Finally, there is an issue in the implementation, which occurs when using the application in multiple windows or tabs in the same browser instance. The window where the user makes their last change overwrites the pending changes of other windows in the IndexedDB. Thus, if the user closes or reloads the window where the pending changes were overwritten before they the application has synchronized them to the server, the changes are lost. One possible solution might be that the browser windows would elect one of them to be a master, which would be responsible for managing the remote and backup sources. However, this subject needs further research in the future.

Chapter 8

Conclusions

Offline-first is a recent design philosophy for developing applications. The main idea of offline-first is that the application is first implemented with the assumption that there is no network connection available. After that, online features are gradually added to the application while maintaining the offline capabilities. The objective of this thesis was to research how such a web application could be implemented.

The first part of this thesis introduced the related design paradigms, including what is offline-first. It also reviewed the different browser technologies that make offline web applications possible, such as application cache, service workers, and client-side storage solutions. The second part of this thesis focused on how to implement an offline-first application in practice based on the findings of the first part.

The design and implementation of the prototype offline-first web application is one of the major contributions of this thesis. Thus, I decided to release the source code as open-source (see Appendix A). The suggestions on how to improve the prototype (see Chapter 7) are the second major contribution of this thesis.

During the literature review, I noticed that the web technologies that enable offline applications are not yet wildly supported in browsers. For example, implementing caching of the application code and assets for offline usage requires using two different technologies, service workers, and application cache, to support the majority of the web browsers. Another example is the background synchronization, which is only supported in Chrome at the moment.

During the implementation, I found out that there a vast amount of challenges and things to consider when building an offline web application. This thesis presented solutions for things such as caching application code and assets, caching application data, and synchronizing changes to a remote

server. However, many aspects were still left to be solved in the future such as conflict management, handling the limited storage space for application data caching, and background synchronization.

All in all, I believe the prototype developed in this thesis will serve as a valuable reference for our company or anyone else who wants to build an offline-first web application. I also believe that the prototype provides a solid playground for experimenting with solutions to offline related challenges not addressed in this thesis.

Bibliography

[1] StatCounter Global Stats. *Mobile and tablet internet usage exceeds desktop for first time worldwide*. Nov. 1, 2016. URL: http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide (visited on Sept. 18, 2017).

[2] StatCounter Global Stats. *Desktop vs Mobile vs Tablet Market Share Worldwide*. 2017. URL: http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#monthly-201608-201708 (visited on Sept. 18, 2017).

[3] Feyerke, A. "Designing Offline-First Web Apps". In: *A List Apart* 386 (Dec. 4, 2013). URL: https://alistapart.com/article/offline-first (visited on Sept. 20, 2017).

[4] Fink, G. and Flatow, I. *Pro Single Page Application Development. Using Backbone.js and ASP.NET*. Apress, 2014. 324 pp. ISBN: 978-1-4302-6674-7. DOI: 10.1007/978-1-4302-6674-7.

[5] Mikowski, M. S. and Powell, J. C. *Single Page Web Applications. JavaScript end-to-end*. With a forew. by G. D. Benson. Manning Publications, 2014. 432 pp. ISBN: 978-1-6172-9075-6.

[6] Champeon, S. *Progressive Enhancement and the Future of Web Design*. Mar. 21, 2003. URL: http://hesketh.com/publications/progressive_enhancement_and_the_future_of_web_design.html (visited on Sept. 14, 2017).

[7] Lambert, J. "Offline First – A better HTML5 User Experience". In: (Nov. 26, 2012). URL: http://www.joelambert.co.uk/article/offline-first-a-better-html5-user-experience/ (visited on Sept. 20, 2017).

[8] Sauble, D. *Offline First Web Development*. Packt Publishing, 2015. 316 pp. ISBN: 978-1-78588-457-3.

[9] van Kesteren, A. et al. *HTML Standard*. Living Standard. WHATWG. URL: https://html.spec.whatwg.org/multipage/ (visited on Sept. 22, 2017).

[10] Russell, A. et al. *Service Workers 1*. W3C Working Draft. W3C, Oct. 11, 2016. URL: https://www.w3.org/TR/2016/WD-service-workers-1-20161011/ (visited on Sept. 19, 2017).

[11] Archibald, J. *Is Service Worker Ready Yet?* URL: https://jakearchibald.github.io/isserviceworkerready/ (visited on Sept. 21, 2017).

[12] Karlin, J. and Kruisselbrink, M. *Web Background Synchonization*. Draft Community Group Report. WICG, Aug. 2, 2016. URL: https://wicg.github.io/BackgroundSync/spec/ (visited on Sept. 21, 2017).

[13] Archibald, J. "Application Cache is a Douchebag". In: *A List Apart* 350 (May 8, 2012). URL: https://alistapart.com/article/application-cache-is-a-douchebag (visited on Aug. 28, 2017).

[14] Deveria, A. *Can I use... Support tables for HTML5, CSS3, etc.* URL: https://caniuse.com/ (visited on Sept. 22, 2017).

[15] Hickson, I. *Web SQL Database*. Working Group Note. W3C, Nov. 18, 2010. URL: http://www.w3.org/TR/2010/NOTE-webdatabase-20101118/ (visited on Aug. 29, 2017).

[16] Mehta, N. et al. *Indexed Database API*. W3C Recommendation. W3C, Jan. 8, 2015. URL: https://www.w3.org/TR/IndexedDB/ (visited on Sept. 22, 2017).

[17] Huang, S. et al. "Browser storage limits and eviction criteria". In: *MDN Web Docs* (July 13, 2017). URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria (visited on Sept. 22, 2017).

[18] "Managing HTML5 Offline Storage". In: *Chrome Developers* (). URL: https://developer.chrome.com/apps/offline_storage (visited on Sept. 21, 2017).

[19] Kitamura, E. *Working with quota on mobile browsers*. A research report on browser storage. Research Report. Google, Jan. 28, 2014. URL: https://www.html5rocks.com/en/tutorials/offline/quota-research/ (visited on Sept. 21, 2017).

[20] Carey, A. "Web and Offline Storage". In: *Microsoft Edge documentation* (Feb. 8, 2017). URL: https://docs.microsoft.com/en-us/

microsoft-edge/dev-guide/storage/web-and-offline-storage (visited on Sept. 21, 2017).

[21] Carey, A. and Grzegorzewski, A. "IndexedDB". In: *Microsoft Edge documentation* (Feb. 8, 2017). URL: https://docs.microsoft.com/en-us/microsoft-edge/dev-guide/storage/IndexedDB (visited on Sept. 21, 2017).

[22] Klabnik, S. et al. *JSON API*. Specification. Version 1.0. May 29, 2015. URL: http://jsonapi.org/format/1.0/ (visited on Sept. 12, 2017).

[23] Gebhardt, D. *Orbit.js Guide*. Version 0.15. Cerebris Corporation. 2017. URL: http://orbitjs.com/v0.15/guide/ (visited on Aug. 2, 2017).

[24] Ellis, C. A. and Gibbs, S. J. "Concurrency Control in Groupware Systems". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*. Ed. by J. Clifford, B. G. Lindsay, and D. Maier. ACM Press, 1989, pp. 399–407. DOI: 10.1145/67544.66963. URL: http://doi.acm.org/10.1145/67544.66963.

[25] Fraser, N. "Differential synchronization". In: *Proceedings of the 2009 ACM Symposium on Document Engineering, Munich, Germany, September 16-18, 2009*. Ed. by U. M. Borghoff and B. Chidlovskii. ACM, 2009, pp. 13–20. ISBN: 978-1-60558-575-8. DOI: 10.1145/1600193.1600198. URL: http://doi.acm.org/10.1145/1600193.1600198.

Appendix A

Source Code

The prototype application implemented in this thesis is released as open-source under the MIT[1] license. The source code is available on GitHub in the following repositories:

- **Client:** `https://github.com/jpvanhal/tasks-frontend`
- **Server:** `https://github.com/jpvanhal/tasks-backend`

[1] `https://opensource.org/licenses/MIT`