Offline Functionality and Testing of Webapps

MASTER THESIS

Artur Wasinger

Submitted on 10 September 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Dr. Andreas Kaufmann
Prof. Dr. Dirk Riehle, M.B.A.

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 10 September 2025

License

Erlangen, 10 September 2025

Abstract

Some of the core characteristics of a progressive web app is the ability to work in the background and to provide offline functionality. QDAcity, a web application for qualitative data analysis, was designed as such a progressive web app, but due to architectural changes, this functionality was partially lost.
Ensuring reliable offline support and verifying its correctness in testing remains a challenge. In QDAcity additional layers like collaborative functionality increase this challenge.
For that reason, this thesis explores the possibilities to implement offline functionality with the help of a service worker for such a collaborative design.
Additionally, offline network emulation on Chromium-based browsers with the Selenium test framework is explored and implemented to especially support end-to-end testing in scenarios of network connection loss.

Contents

List of Figures

List of Tables

Acronyms

CI/CD Continuous Integration/Continuous Delivery

CAQDAS computer assisted qualitative data analysis software

DOM Document Object Model

E2E end-to-end

GAE Google App Engine

GCP Google Cloud Platform

PDF Portable Document Format

QDA qualitative data analysis

PWA progressive web app

SW service worker

UI user interface

CRDT conflict-free replicated data type

CES collaborative editing service

RDBMS SQL-based Relational Database Management System

OPFS Origin Private File System

GCS Google Cloud Storage

SEO search engine optimization

AT Acceptance test

CDP Chrome Devtools Protocol

1 Introduction

According to Global Market Insights, Progressive Web Apps (PWAs) represent a rapidly growing market. For instance, the market size is expected to increase from USD 9.4 billion to USD 40 billion by 2033 (Grand View Research, 2025).

This thesis explores the core concepts of progressive web app (PWA)s and applies them in the context of the QDAcity[1] software platform. QDAcity is designed as a PWA (Kaufmann, 2021). According to Mozilla, a PWA is defined by its ability to work in the background and provide offline functionality (Mozilla Developer Network, 2025a). However, due to changes to the system architecture in QDAcity, one essential feature —*offline functionality*— has been partially lost.

This thesis specifically investigates how this capability can be restored and maintained in future development for the QDAcity project.

The primary challenge addressed in this work is enabling QDAcity to operate effectively in offline scenarios. This challenge manifests in two essential use cases:

- Ensuring application operation during loss of internet connectivity.
- Ensuring data consistency across offline and online states.

A secondary goal is to ensure that the resulting implementation remains maintainable and aligned with modern engineering standards.

This thesis builds upon the previous work of Sebastian Knauer (Knauer, 2018), who initially implemented offline support in QDAcity. However, due to architectural changes, particularly the introduction of the collaborative editing service (CES), his approach became obsolete. CES adds complexity to the system by enabling real-time collaboration on domain objects like *documents* and *projects* via the `Yjs` library and exchanges the Realtime Collaboration Service (RTCS) on which Knauers implementation was based on.

Furthermore, the original end-to-end (E2E) testing strategy for offline functionality, which relied on shutting down the backend server to simulate offline use,

[1] https://qdacity.com/

was abandoned due to changes in the testing infrastructure.

The main goals of this thesis are to enable offline support for QDAcity, especially the CES, and to add testing infrastructure, that works also in the GitLab CI/CD pipeline for the QDAcity project.

1.1 Methodology

The development process in this thesis followed an agile and iterative approach. While the rough requirements were worked out at the start of the project, the exact technical design and implementation evolved based on continuous evaluation and feedback in an active development process.

Initially, the application was analyzed to identify components that lacked proper offline behavior and which part of the earlier implementation of Knauer still worked. Some foundational offline features, such as caching static assets, were already in place. However, for example, critical areas like HTTP request handling and data synchronization were not fully addressed anymore.

The first implementation for the offline functionality focused primarily on core application components, including the document editor and project management features. The guiding principle was to prioritize working solutions first, followed by iterative refinement.

Evaluation was integrated into the development cycle. After each implementation step, feedback was gathered and used to assess both functionality and maintainability. It was continuously integrated into the QDAcity project.

1.2 Thesis structure

Following this chapter the thesis shows browser technologies in chapter 2 that are needed to understand and follow the architecture and implementation. In chapter 3 the requirements are defined. Then following the architectural overview of the app (chapter 4) and in chapter 5 the design and implementation of the work. The evaluation of the approach and requirements is described in chapter 6 followed with a conclusion in chapter 7.

2 Browser Technologies

PWA rely on a set of browser technologies that enable offline availability, background processes, and data persistence of resources. In the context of QDAcity, these technologies form the foundation for ensuring that users can continue working in unstable or offline environments while maintaining data integrity and usability. Additionally, PWAs include features like add-to-homescreen installation and push notifications that make them feel more like native apps on mobile devices, but which are not relevant for QDAcity.

2.1 Service Worker

A service worker (SW)[1] is a type of Web Worker[2] that allows a browser to run JavaScript in a background thread, separate from the main execution thread of a web application.
Figure 2.1 shows an exemplary multi-process architecture of Chromium-based browsers. Every new window or tab is run as a separate render process. The SW is not part of the render process, it runs in a worker context and therefore has no Document Object Model (DOM) access.

The SW operates as a programmable proxy between the web application and the network. It enables advanced features such as offline functionality, intelligent caching strategies, background synchronization, and interception of network requests. As such, SWs are a key component in the architecture of modern PWAs.

Service Worker Messaging

SWs communicate with the main thread through an *event-driven messaging system* based on the `postMessage` API[3]. This mechanism enables bidirectional data

[1] https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
[2] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API
[3] postMessage API Documentation:

- https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker/postMessage
- https://developer.mozilla.org/en-US/docs/Web/API/Client/postMessage

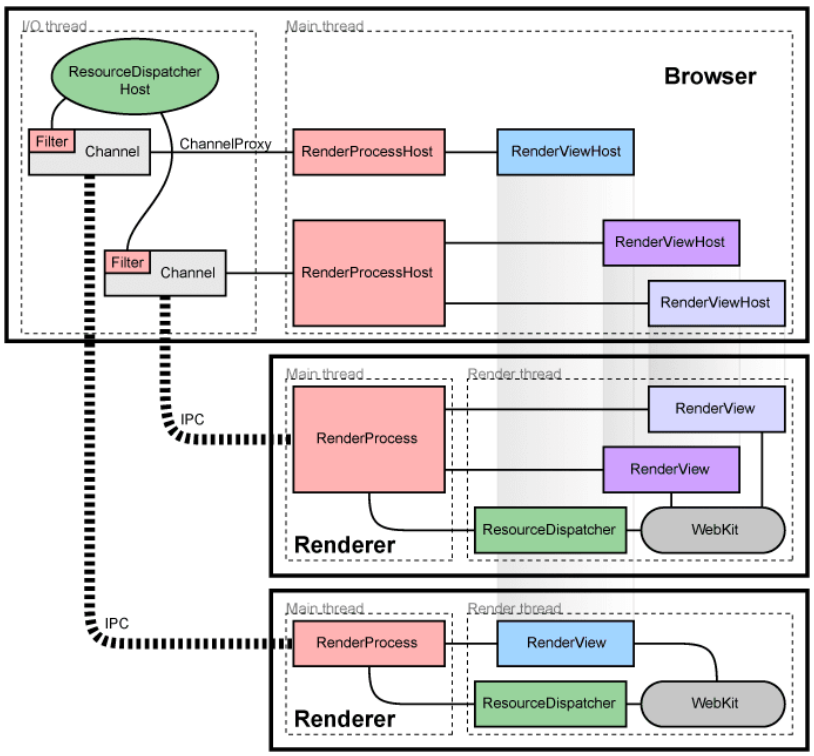

Figure 2.1: Multi-process architecture for Chromium-based browsers (Chromium Project, 2025)

exchange between the page context and the SW. Such messaging is essential for synchronizing application state, triggering background tasks, and informing the main application about network or cache-related events.

2.2 Caching & Persistence

Within browser development, there have been many established concepts of how to persist data. To store data client-side the development team of Mozilla explains some of the most common structures (MDN contributors, 2025):

Web Storage API The Web Storage API[4] provides storage and retrieval for smaller data items in form of key value pairs. It is most useful to store simple data. The Storage API uses two main mechanisms for that one is the `localStorage`, where data persists even when the browser is closed, and the other is the `sessionStorage`, where closing the browser tab will destroy all data in this storage.

[4]https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

IndexedDB API Another mechanism to store data on the client-side is the IndexedDB API[5]. This mechanisms provides the browser a complete database system, which enables the storage of complex data, e.g. complete sets of records or data types like audio or video files. An IndexedDB is a transactional database system, but compared to SQL-based Relational Database Management System (RDBMS) it uses `keys` as indexes for the stored objects, instead of fixed-column tables. The database consists of key-value pairs, where the value can be complex structured objects, not like the `localStorage` or `sessionStorage`.

Cache API This API[6] is best used to store HTTP Request / Response objects, e.g. storing website assets. These objects are cached in a long lived memory, which is browser dependent. Still, these objects don't expire unless deleted or until the hard limit on the amount of cache storage is reached. Some of the most used browser quotas are mentioned in table 2.1. In case of *best-effort storage*, data may evict when the storage is low and for *persistent storage*, data is not automatically evicted, even under storage pressure, and needs explicit requests.
For browsers the Chrome team notes, that even in the case for *best-effort storage*, data is rarely evicted automatically by the system (LePage & Web ev, 2020). For Webkit/Safari the quotas are based on the origin. For the Safari web browser, QDAcity and any cloud app each can use up to 60% of the total disk space, where Webkit also enforces an overall total of 80% for browser apps. Non-Webkit browsers for macOS/iOS impose their own quota policies.

HTTP Cookie Cookies are a mechanism for storing small amounts of client-side data. They are automatically transmitted with every HTTP request to the originating domain, which makes them commonly used to persist state and user information across page navigations. However, due to limited storage capacity, which is usually 4KB[7], these are not useful for offline data persistence and are therefore not considered further for this work.

Origin Private File System Origin Private File System (OPFS)[8] is another storage endpoint that makes it possible to access persistent storage. The File System API[9] allows to have a special kind of file that is private to the origin of the page and is not visible in the users regular file system.
This is a newer mechanism to support file read and write and is not sup-

[5] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
[6] https://developer.mozilla.org/en-US/docs/Web/API/Cache
[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies
[8] https://developer.mozilla.org/en-US/docs/Web/API/File_System_API/Origin_private_file_system
[9] https://developer.mozilla.org/en-US/docs/Web/API/File_System_API

ported by all browsers fully yet, this can be seen in appendix A.

Browser / Engine	Quota Policy (approx.)
Firefox	Best-effort storage: up to 10% of disk or max 10 GiB. Persistent storage: up to 50% or max 8 TiB.
Chromium (Chrome, Edge, etc.)	Up to ∼60% of total disk space per origin (best-effort and persistent storage)
Safari/WebKit	For browser apps up to 60% per origin and 80% overall Embedded in another app's WebView: 15% per origin. When user saved the site as web app, then same quota as the browser app.

Table 2.1: Approximate browser storage quotas relevant to the Cache API[10]

[10]https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_eviction_criteria

3 Requirements

This chapter introduces the requirements for this thesis. To define, structure and then test these, this thesis follows the template of FunktionsMASTER from the SOPHISTen (SOPHISTen, 2024).

The requirements details were developed continuously in an iterative approach. The refinement of these requirements was made and given based on the problems that were faced during development. Because this work uses and is based on the work of Sebastian Knauer (Knauer, 2018), there are many intersections with his defined requirements.

3.1 Functional Requirements

3.1.1 General

FRQ-1 As soon as the user is offline, the app shall display an indicator for being offline.

FRQ-2 As soon as the user is offline, the app shall give feedback when the user tries an offline feature, that is not supported in offline mode.

3.1.2 Offline Support

FRQ-3 The app shall not work offline until the user has logged in successfully.

FRQ-4 The app shall allow a user to enter and edit data of domain objects, that are configured for the offline functionality, without loss of input, when the network connection is lost.

FRQ-5 The app shall retry POST requests that are not meant for synchronization, like one-way logging, when the network connection is re-established. These should be configured and marked accordingly.

FRQ-6 The app shall handle GET requests when the network has lost connection by returning cached results if available otherwise, return an error

response.

FRQ-7 The app shall handle POST/PUT/DELETE HTTP requests when the network has lost connection, that need synchronization aftercare, by returning a simulated success response and storing the request data locally for synchronization.

FRQ-8 The app shall register a service worker to handle caching for site content and API responses.

3.1.3 Persistence

FRQ-9 The app shall store user data in local persistent storage, by using Cache and IndexedDB when offline.

FRQ-10 The app shall have a UI indication for domain objects when they are stored locally and awaiting synchronization.

3.1.4 Synchronization

FRQ-11 The app shall synchronize stored changes with the server as soon as the connection is re-established.

FRQ-12 The app shall resume synchronization after temporary network loss.

FRQ-13 The app shall resolve conflicts using a CRDT strategy when syncing offline data.

3.2 Non-Functional Requirements

The non-functional requirements are based on the PropertyMASTER template from (SOPHISTen, 2024). The following chapters declare the technological requirements and quality requirements, that are used to further define the framework of the development.

3.2.1 Technological Requirements

NFRQ-1 The implementation shall use TypeScript.

NFRQ-2 The app shall use a service worker to support offline functionality.

NFRQ-3 Data types, used for the collaboration functionality, shall be compatible with the Yjs Docs library for the offline functionality.

NFRQ-4 The testing environment shall use Selenium as the base for End-to-End testing.

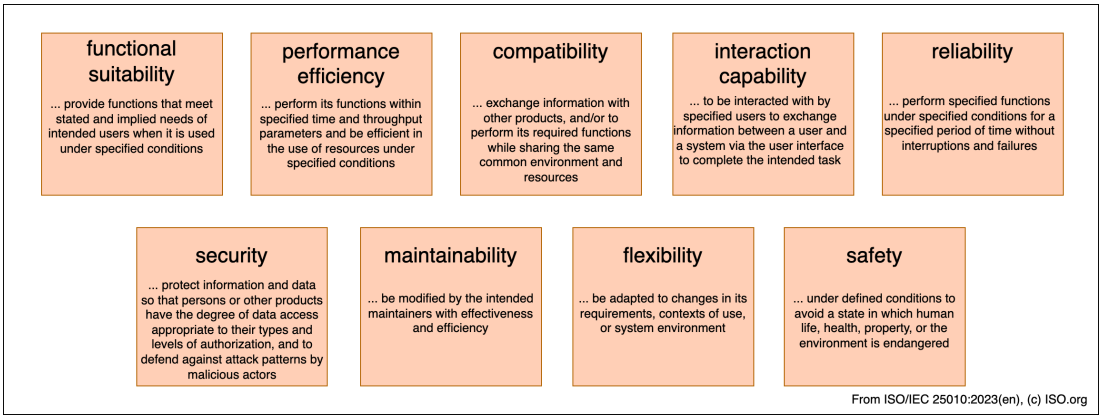

Figure 3.1: ISO 25010:2023 Quality Requirements

NFRQ-5 The testing environment shall work in the GitLab CI/CD pipeline used for QDAcity.

3.2.2 Quality Requirements

The ISO25010 describes a comprehensive framework for quality metrics. The figure 3.1 shows an overview, while the requirements defined in this thesis focus on maintainability and testability.

Maintainability

NFRQ-6 The codebase for offline functionality should follow clean architectural principles that support modularity and separation of concerns. In particular:

- **(a)** Encapsulate all access to the browser data store via a well-defined interface.
- **(b)** Implement synchronization services that encapsulate synchronization business logic for specific domain objects.
- **(c)** Define a communication interface between service worker and the main application.

NFRQ-7 Supporting offline functionality for endpoints should not change main app functionalities aside from the addition of a decorator, controller and synchronization service.

NFRQ-8 The documentation should explain how endpoints are supported with offline functionality.

Testability

NFRQ-9 All functional components, especially offline workflows and synchronization logic, shall be designed for automated testing.

NFRQ-10 The test extension for offline workflows and synchronization logic shall integrate into the current test framework.

4 Architecture

In the following section the architecture is explained in with focus for the offline functionality. The original architecture is based on Knauer's work (Knauer, 2018) and was extended as necessary.

4.1 QDAcity

QDAcity is a cloud-based web application designed to support qualitative data analysis (QDA)[1]. Also known as computer assisted qualitative data analysis software (CAQDAS), this QDA software offers features to support analysis in an individual and collaborative context. Some of these features include interview analysis, speech-to-text transcription form, and analytical statistics. Researchers can perform qualitative data research by uploading relevant data and organizing the teams. QDAcity also supports educational purposes, like course organization and exercises.

4.1.1 QDAcity Architecture

Figure 4.1 shows the different components that QDAcity consists and maps their communication relations. QDAcity consists of three different main components:

Frontend The frontend includes the user interface (UI) as a view and working area of the web application, which is based on the React framework[2]. The SW is also running on the client side in the background. A frontend client communicates with the backend over the QDAcity API and with the CES, with a few exceptions (like authentication), over a websocket connection. This connection to the CES is build using the Hocuspocus library[3], which enables web applications to use collaborative document editing.

[1] qdacity.com
[2] https://react.dev/
[3] https://tiptap.dev/docs/hocuspocus/introduction

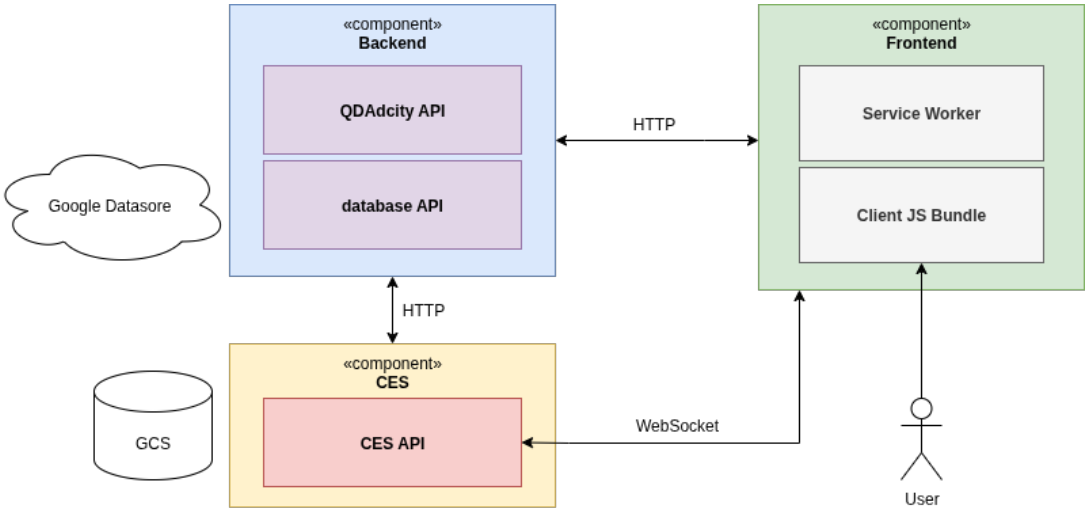

Figure 4.1: QDAcity component communication relations

CES The Hocuspocus-Library is setup in the CES, which includes the configuration of the persistence layer for Yjs documents[4]. The Yjs technology represents a high-performance conflict-free replicated data type (CRDT) library. CRDT allows data to accept updates without remote synchronization, to ensure performance and scalability (Shapiro et al., 2011). This concludes the collaborative feature of QDAcity. Final data is persisted in a Google Cloud Platform (GCP).

Backend The third main component is the backend, which is build upon a Google App Engine Framework[5] in the Java language[6]. This component deals with receiving and persisting responses of the frontend web client, in combination of supporting respective domain objects with the CES together.

4.1.2 Core domain objects in QDAcity

To better understand QDAcity, the table 4.1 presents a small set of the core domain objects in QDAcity. Combined these core objects are necessary to form a `coding system`, which is an abstraction of the information after QDA has completed.

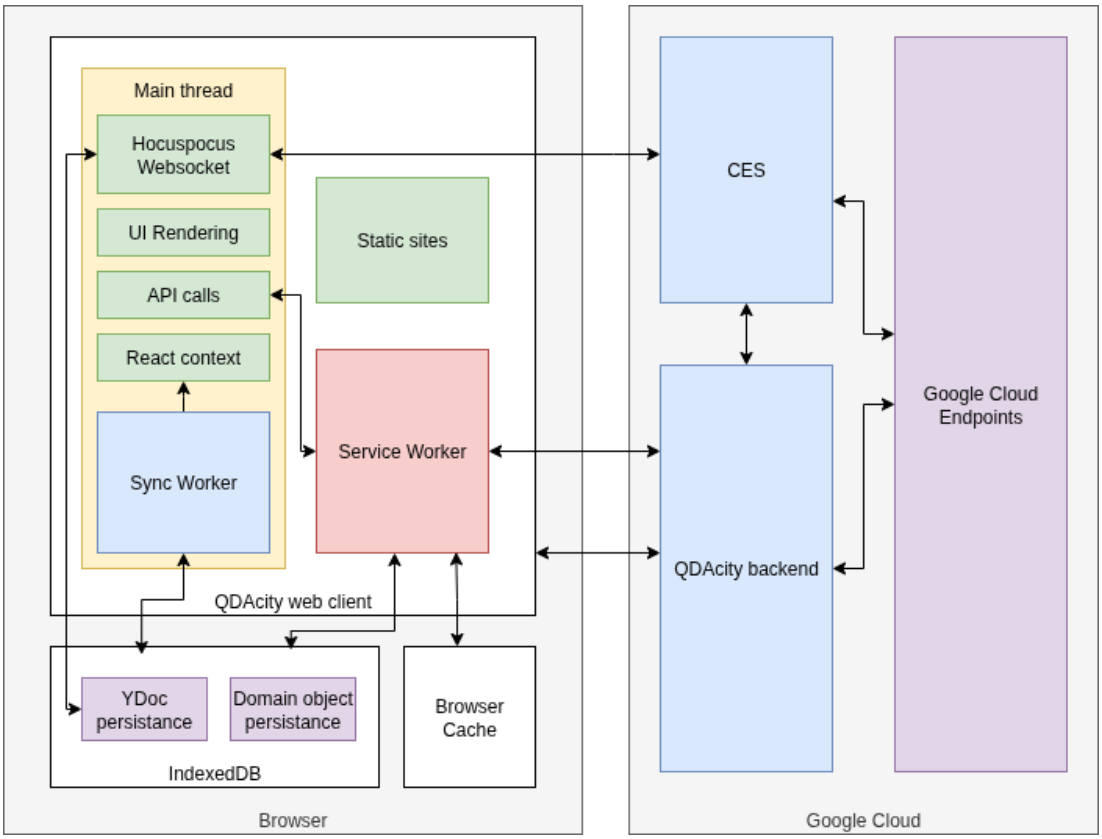

Figure 4.2: QDAcity web client and backend / CES communication flow

Object	Description
`Project`	Root container for all work. Every other object (documents, codes, codings) belongs to a project.
`Document`	Input material for analysis. Can be a text document, PDF, or other supported format. The main analytical work takes place within documents.
`Code`	Conceptual tag used to group and classify specific themes during qualitative data analysis (QDA).
`Coding`	Link between a code and a specific section of a document. Together with the set of defined codes, codings form the `coding system`.

Table 4.1: Core domain objects in QDAcity

4.2 Communication Flow

In figure 4.2 the communication flow between the QDAcity web client and the QDAcity server components, which is running within a Google App Engine (GAE) in the Google Cloud can be seen. The browser runs the JavaScript code for a page in a single thread, called the Main Thread[7]. Within the Main Thread the application loads and runs the main business logic.
This logic communicates changes of *Yjs Docs* over a websocket connection with the CES.

The backend and the CES then deal with the communication to the Google Cloud Endpoints, which are also handling the persistence work for QDAcity.
The SW is not running in the Main Thread and acts more like a proxy server between the frontend client and the backend within the browser. To hold persistent data in the browser, the SW uses the IndexedDB and the cache. Sync Worker also communicate with the IndexedDB, as the domain objects are persisted here, more information on that architecture can be found in figure 4.4. React contexts within the app are also messaged by the Sync Worker, where the domain objects that are created by the Sync Worker also need to be updated in the contexts for the UI components, that are shown at that time.

The Main Thread communicates to the IndexedDB by use of a provider interface from Yjs Doc and synchronizes the information of created Yjs Docs used by the Hocuspocus websocket. More information is found within section 4.4.

[4]https://docs.yjs.dev/api/y.doc
[5]https://cloud.google.com/appengine
[6]https://www.java.com/
[7]https://developer.mozilla.org/en-US/docs/Glossary/Main_thread

4.3 Collaborative Architecture

A fundamental challenge in collaborative systems is conflict resolution: when multiple users edit the same document at the same time, changes can overlap or contradict each other.
For that challenge Saito & Shapiro introduce the *optimistic replication* model, where eventual consistency promises better availability and performance (Saito & Shapiro, 2003). Based on this model CRDT data types have been developed, where replica can accept updates without remote synchronisation (Shapiro et al., 2011), which make them perfect for collaboration or offline synchronization. This is also a technology used by QDAcity.

To support a CRDT datastructure the web app uses the Hocuspocus suite[8], which is a set of tools to support collaboration for web applications and is based on the CRDT implementation Yjs[9]. Within the used implementation CRDT data is structured as *Yjs document* and are represented by the `Y.Doc` object.

In 4.3 the architectural setup for the collaborative system can be seen. For the backend the CES configures a Hocuspocus Server, which is a websocket backend for Yjs support[10]. This server configures multiple extensions, like logging and connection throttling.
Noteworthy is here the Redis extension[11] and the custom QDAcity extension. Through Redis horizontal scaling is possible by allowing multiple server instances to sync changes and states.
The QDAcity extension configures hooks for the Yjs document, for example `onLoadDocument, onStoreDocument`. The CES also implements the QDAcity API client and Google Cloud Storage (GCS) API.
The client establishes communication by opening a websocket connection via a `HocuspocusProvider` provider, when the document has not been loaded into the current state, that is saved within the `providersMap`-map. The React hook is called `useHocuspocus` and holds all the logic for setting up the provider and its configuration.

4.4 Caching & Persistence Architecture

In this section the focus is on the IndexedDB storage, which is already mentioned in section 2.2. QDAcity uses all the persistence strategies to various degrees, except the OPFS.
With the CES and the collaborative feature Yjs and the Hocuspocus suite was

[8] https://tiptap.dev/docs/hocuspocus/introduction
[9] https://github.com/yjs/yjs
[10] https://tiptap.dev/docs/hocuspocus/introduction
[11] https://tiptap.dev/docs/hocuspocus/server/extensions/redis

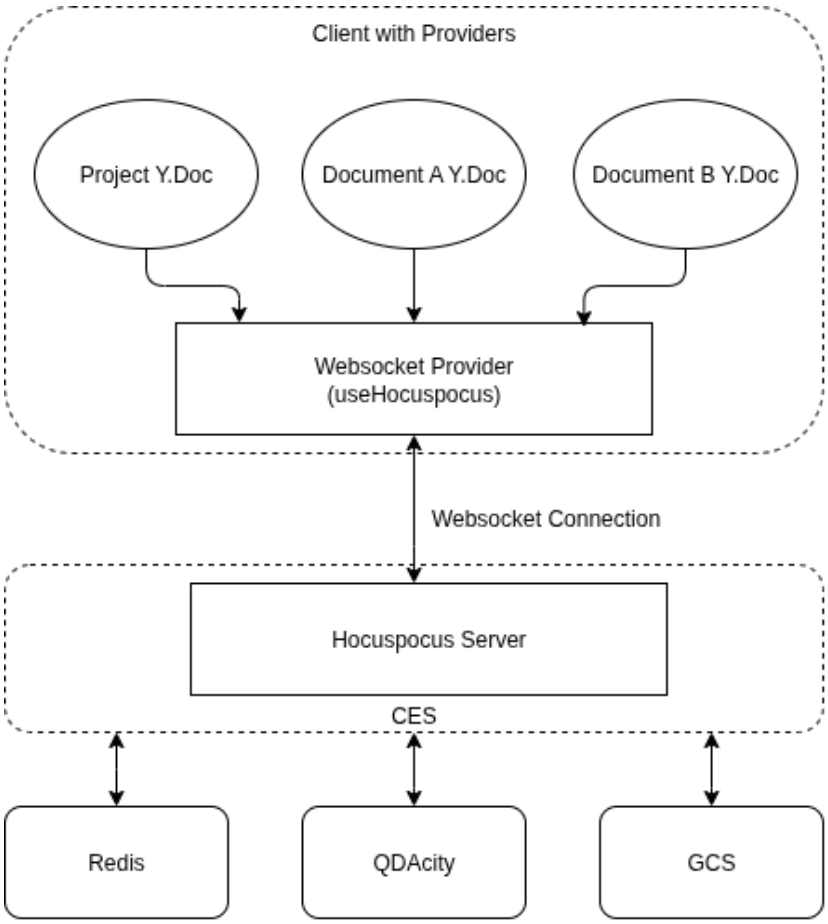

Figure 4.3: Provider-Server architecture for YDocs

introduced to QDAcity. The collaborative feature works without an IndexedDB storage, but to support offline functionality, Yjs uses the `IndexeddbPersistence`[12]. This creates a provider instance, which syncs all changes to an IndexedDB storage within the browser.
Knauer's work also introduced the IndexedDB for persistence, and is thus extended for the Yjs indexedDB support. As seen in figure 4.4 the browser holds multiple databases.
Based on Knauer's work the storage for the entity classes does not change. Each user gets his own IndexedDB database and each entity is held within its own store. In case for Yjs, each `Y.Doc`, that represents a *document* or *project* domain object, is getting its own IndexedDB database. Within these are two stores: `updates` and `custom`. The `updates` store holds the CRDT updates. The `custom` storage can be used with custom YDoc providers, for this thesis this storage is not used.
Additionally there is the `pdfStorage` and the `workbox-background-sync` database.
The `pdfStorage` database was implemented to be able to support PDF upload, when offline. A detailed explanation on the reasons can be found in section 5.3.4.
For `workbox-background-sync` this IndexedDB is created from the Workbox Background Sync module described in section 5.1.3. The queued requests are saved here.

4.5 Testing Architecture

Testing an application is needed to improve and ensure the quality of code and features. To make this happen, QDAcity uses various technologies.
To frame this, the next section focuses on a few of these testing technologies, with the focus on automated testing.

GitLab CI/CD pipeline

Continuous Integration/Continuous Delivery (CI/CD) is a method of continuously integrating code changes into a code base and then deploying this automatically, too. For QDAcity the pipeline technology used here is *GitLab CI/CD*, that is configured with `YAML`-files[13].
This section focuses on the testing infrastructure for this pipeline.

In figure 4.5 one can see the different jobs included in the automatic testing.
The test stages are running independently:

[12] https://docs.yjs.dev/getting-started/allowing-offline-editing
[13] https://yaml.org/

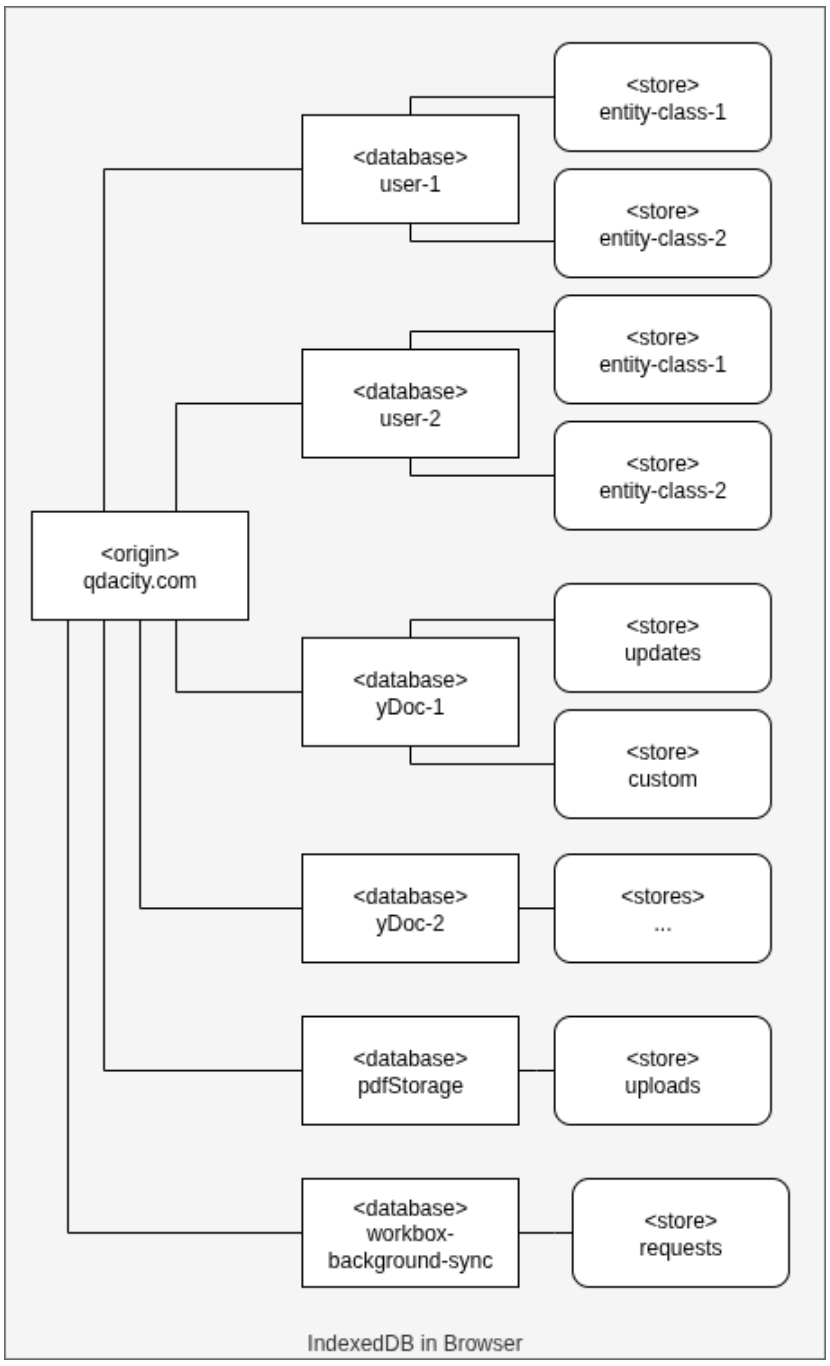

Figure 4.4: IndexedDB architecture

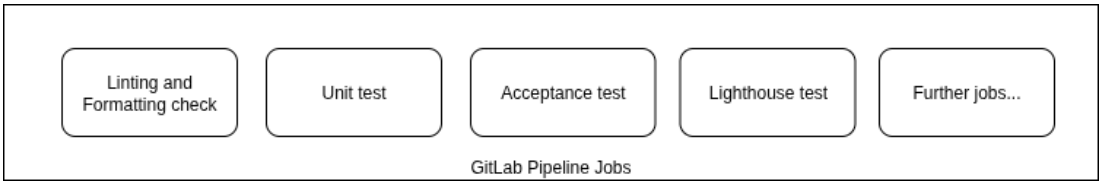

Figure 4.5: GitLab Test Jobs

Linting and formatting check where the code is held up to standards for style and possible programmatic errors by statically analyzing the code.

Unit tests that check individual functions, classes and other component in isolation.

Acceptance tests which validate E2E user workflows with defined acceptance criteria on a production-like build.

Lighthouse tests audit the build site for non-functional qualities like performance, accessibility and search engine optimization (SEO).

In this thesis the focus is on the Acceptance test (AT), because one wants to simulate network loss. This can be achieved and supported by the AT framework *Selenium*, which is used by QDAcity for the ATs. In section 5.4.1 the detail implementation and design is shown.

Dockerization

To run the AT inside the pipeline or for ease on a development machine Docker[14] is used. Docker has the great advantage that the infrastructure runs on the same dependencies for everyone. It can simulate the production environment with containers as necessary.
The script wrapping and executing the Docker containers is called *executeAcceptanceTestsJob_Java.(sh|ps1)*, which is also the same script used within the pipeline to start the execution for the ATs. There are two versions one for execution in Powershell (Windows) and one in bash (Linux). The script used in the pipeline is bash-based.
Especially for ATs the container approach makes sense, because when trying to test multiple different types of browsers or AT parallelization can be better scaled by Docker in the pipeline. Selenium provides a the Grid architecture[15] to support such scaling and uses docker-selenium[16] images for different kind of browsers.

[14]https://www.docker.com/
[15]https://www.selenium.dev/documentation/grid/architecture/
[16]https://www.selenium.dev/blog/2024/multi-arch-images-via-docker-selenium/

Another problem is the release count for browsers, e.g. when looking at the dates for the Chrome browser releases[17], one can see that new browser versions approximately happen all four weeks. Here the container approach makes sense, to have the ATs run on specific versions and check on which version has potential failures.

[17] https://developer.chrome.com/release-notes

5 Design and Implementation

This chapter shows the design and implementation for the offline functionality. It covers the SW implementation, changes to the main application and the implementation of the offline support in the Selenium AT environment.

5.1 Service Worker

The main entry point for the SW is the `sw.ts`. This file holds the core logic of the SW. As the browser can't execute TypeScript code. The `sw.ts` will have to first be transpiled into JavaScript by webpack[1]. Webpack bundles and links code parts together and compiles them into static assets, that can be delivered to and executed by the browser. The resulting `sw.js`-file it then registered as described in section 5.1.
To achieve the compilation for Webpack the sw.ts is set into the entry part of the `module.exports` in the *webpack.config.js*. Additionally the resolve extension for `.ts`-files was added in the module configuration too. This will compile the `.ts`-files from TypeScript to JavaScript and extends the already `.tsx`-support to full TypeScript support. TSX is the TypeScript version of JSX[2], which is a ECMAScript syntax extension commonly used in React apps, which enables developers to write HTML in React code[3].

The reason the SW now supports TypeScript is to improve the code quality, because the typization feature helps improve the handshakes for functions and objects by setting clear types. Also a dynamic Workbox route registering strategy can be implemented by the TypeScript decorator feature, mentioned by the `CacheMethodDecorator` in section 5.3.2.

5.1.1 Lifecycle

[1] https://webpack.js.org/
[2] https://facebook.github.io/jsx/
[3] https://legacy.reactjs.org/docs/introducing-jsx.html

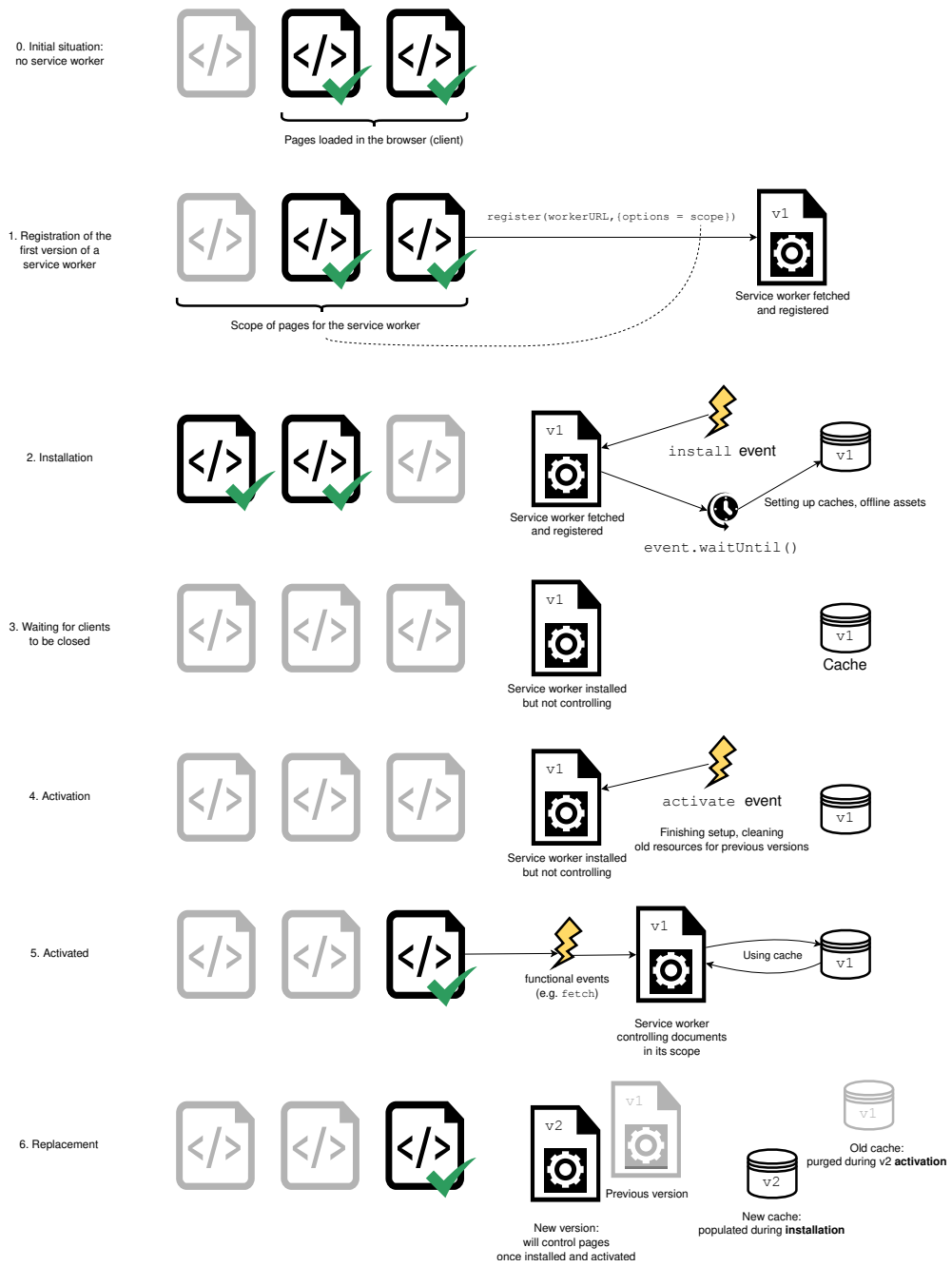

Figure 5.1: Service Worker Lifecycle (Mozilla Developer Network, 2025b)

```
1 navigator.serviceWorker.register('/sw.js', { scope: '/' })
```

Listing 1: Service Worker registering

The SW lifecycle starts with the registration of the worker as seen in figure 5.1. In QDAcity registration goes over a callback in the `index.js`-file in the frontend, to then load and register theSW. The code executed compacts to the one shown in listing 1, enhanced with error control. The scope, that the SW can deal with, is defined to the whole possible range of URLs[4]. The general start of the registration is run in a `window.onload`-function, which runs, when the browser *window load* event is triggered[5]. With the registration, the code for the SW is executed. Here the SW first registers different event listeners:

Install The event listener for the install-Event executes `skipWaiting()`, which forces the waiting SW to the become active SW [6]. That is needed to ensure newly installed SWs to become active, that are else staying in a waiting state.

Activate The activate event listener executes a `clients.claim()`[7], which enables the SW to be the controller of all pages within its scope, else the controller change would only happen on a next page load.

Message This listener deals with receiving any `message`-Event. For the SW implementation this has no life cycle reason, but deals with handling a register route logic and is part of the `CacheStrategy` implementation described in section 5.3.2.

5.1.2 Service Worker Proxy

In Figure 5.2 the SW proxy depiction shows the flow from an API request from the client to the QDAcity API for the case that a custom controller strategy is used. The Main Thread first calls a *fetch*-Event on a request. This *fetch*-Event comes from the Swagger API client through the `Promisizer`-wrapper, which gets the specified API parameters, also from the Swagger QDAcity client and executes them.
This *fetch*-Event will be intercepted by the SW, which handles it based on its configuration. In case there is a Workbox route or a `CacheStrategy`-decorator configured for that endpoint. This flow will be handled by the `ResponseHelper`.

[4]https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration/scope
[5]https://developer.mozilla.org/de/docs/Web/API/Window/load_event
[6]https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope/skipWaiting
[7]https://developer.mozilla.org/en-US/docs/Web/API/Clients/claim

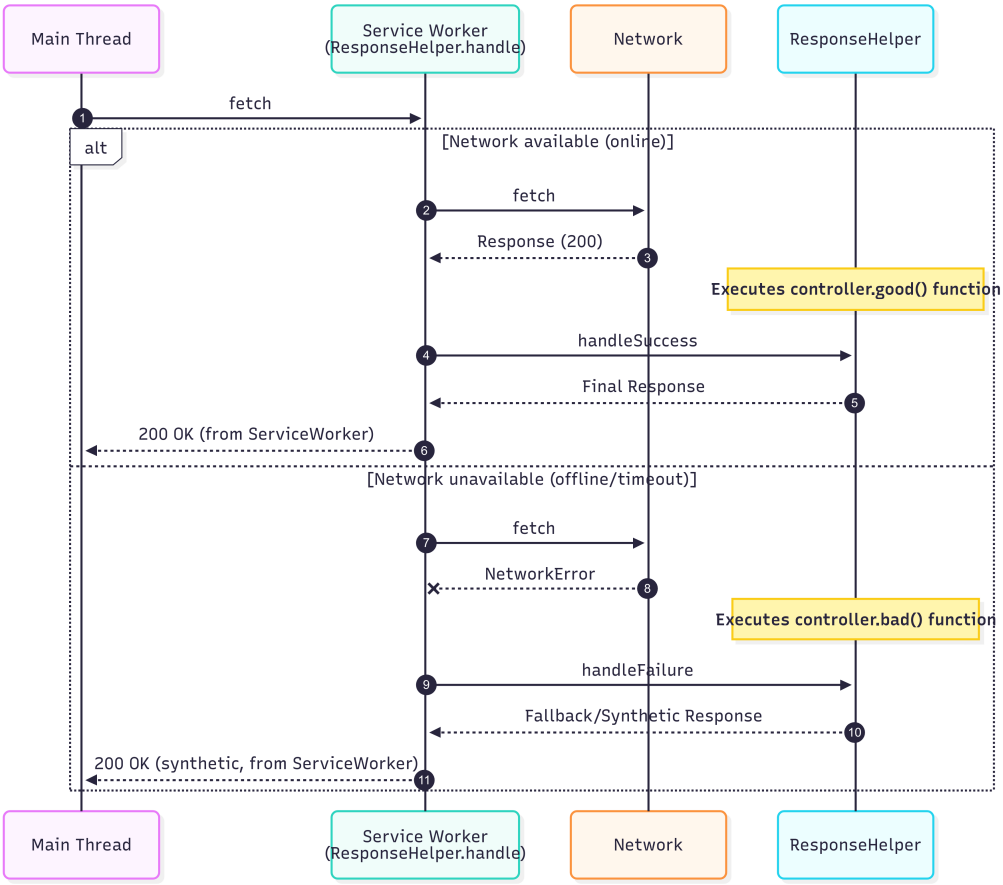

Figure 5.2: Service Worker Proxy Diagram

See section 5.3.2 for reference on how endpoints are decorated and section 5.1.3 for a detailed Workbox description. In case nothing is configured, the client will receive a network error, but still tries to pass it through the SW as a proxy. There are multiple different strategies on how the route is managed these are all described in the Workbox section. The `ResponseHelper` and the controllers are described in section 5.2.1. In short these controllers manage the request and either pass it to the IndexedDB in the `good()` case or give a synthetic or fallback response (like an *offline not supported*-error) back.

5.1.3 Workbox

Workbox[8] is a library for SWs by the Chrome Development Team. This library helps to create PWA functionality for web applications by supporting different parts of a browser in the form of modules. The following section deals with the

[8]https://developer.chrome.com/docs/workbox

used modules of the Workbox-Library.

Workbox Precaching

A feature of SWs is to save files during the installation phase of the SW, this is known as *precaching*. The `workbox-precaching` module[9] simplifies the caching during a SW's install phase. It collects and deduplicates the assets every time the SW is installed and activated. So the user has the latest assets loaded, and only downloads them when changes have been occurring.
The implementation of such a precache usage can be seen in listing 2, where the `precacheAndRoute()`-function from the module is used. Here the function receives two parameters one is an `Array<PrecacheEntry | string>`, where the implementation sets the url and revision parameters for the `PrecacheEntry`, the other is the `PrecacheRouteOptions`. The `PrecacheEntry` comes as an object where the url is set as `DISCOVERY_URL`, which links to the `<app-path>/openapi-$API_VERSION$.json` -document which holds the API information on the URLs. The other component is the revision parameter, which is needed to create versioning for the cached inputs. That means that every time the openapi-API-document changes, the precache redownloads that file. The last options parameter is `ignoreURLParametersMatching`, which is set to a Regular Expression, that ignores all URL parameters for the cache. For example the URL `/about.html?utm_campaign=abcd` will remove all parameters after the *?*-char, so only the `/about.html`-route will be cached.

```
precacheAndRoute(
    [
        {
            url: DISCOVERY_URL,
            revision: '$DISCOVERY_HASH$',
        },
    ],
    {
        ignoreURLParametersMatching: [/.*/],
    }
);
```

Listing 2: Workbox Precaching

[9] https://developer.chrome.com/docs/workbox/modules/workbox-precaching

Workbox Routing

Request handling for the network is done over the `workbox-routing` module[10]. The Workbox routing works when the network request causes a *fetch*-event. The SW acts as a proxy and receives this request and its URL. After that the Workbox routing mechanism tries to respond by matching the request URL with the beforehand registered routes, and continues with the configured strategy.
The routes are registered in the `sw.ts` and match with another route by checking a match callback, which is a callback function that just matches the route. See Listing 3 for example.

```
1 const matchCb = ({url, request, event}) => {
2     return url.pathname === '/special/url';
3 };
```

Listing 3: Example callback function

```
1 registerRoute(/.*\.js/, new NetworkFirst());
```

Listing 4: Example registerRoute

The match callback can be supplied as a Regular Expression format, for example seen in listing 4. The network handling strategy is also supplied to the registering to the route, which can be checked at 5.1.3.
A restriction to use the `registerRoute()` is that routes and request that don't match the default `GET`-requests, like `POST`-requests, need to explicitly be defined, with the `POST`-parameter, as seen in listing 5.

```
1 registerRoute(matchCb, handlerCb, 'POST')
```

Listing 5: Example POST registration template

Workbox Strategies

To deal with different situations Workbox supports various strategies to deal with network responses. Jake Archibald shows some supported ideas in his offline cookbook (Archibald, 2014), that are used in the `sw.ts` implementation.

[10]https://developer.chrome.com/docs/workbox/modules/workbox-routing

Offline strategies are linked in the enum to the Workbox strategies. The enum `CacheWorkboxStrategy` represents the Workbox strategies and are used for the `CacheStrategy`-decorator. They are mentioned later in section 10. Workbox uses the strategies as a handler input when registering a route, as seen in section for Workbox Routing 5.1.3.

Noteable Workbox strategies are as follows:

- NetworkFirst
- CacheFirst
- NetworkOnly
- Custom strategy

NetworkFirst tries to first reach the network and get new data from the network before retrieving data from the cache. This is generally the most used strategy for `GET`-requests, the data needs to be up to date before wanting to load new one.

In opposition to that is **CacheFirst** which first checks the cache for input, before making a network request in case there is a cache miss. There is only one good use for this strategy: when trying to access the static files for the cache. In listing 6 the `CacheFirst` registering is shown.

```
registerRoute(/.*\textbackslash.cache\textbackslash..*/, new
↪ CacheFirst());
```

Listing 6: CacheFirst route registering

NetworkOnly With this strategy HTTP-responses are not handled further, but the request is just forwarded to the backend, e.g. in case of logging. Routes used for the Workbox background synchronization are also configured with a `NetworkOnly` strategy, where `POST` requests are also handled a special case. See section 5.1.3 for more details.

For **custom handling** of the requests, Workbox supports a custom `RouteHandler` in its `registerRoute()`-function. The registered Workbox route will then handle the request based on the configured controller, that are mentioned in section 5.2.1.

Workbox Background Sync

POST requests are normally not supported, by the standard methods of Workbox. To handle these one can add custom response handlers, in form of controllers. These are mentioned in section 5.2.1. In some cases, for example logging, there is

no need to synchronize data or have a special kind of response handling. Here the strategy would be: To save the request in a persistent storage and the repeat it, when the network connection is re-established. To make this possible a queue is introduced with the workbox-background-sync module[11]. Requests that have no requirement on being synchronized and are stateless can be accumulated within a queue in the IndexedDB, and then repeated, on a later date.
In the `sw.ts` a `BackgroundSyncPlugin` was added, with which a registered route can be extended. The plugin is added to strategies and holds a retention time of 24 hours. Listing 7 shows how the `NetworkOnly`-strategy is extended with the background sync plugin.

```
registerRoute(
    regex,
    new NetworkOnly({
        plugins: [bgSyncPlugin],
    }),
    httpMethod
);
```

Listing 7: BackgroundSyncPlugin implementation for a route

Service Worker - Main App Communication

Within a browser exist many ways of communication within the browser contexts, some of them include `BroadcastChannel` to have many-to-many connections, `MessageChannel` for a one-to-one pipe and the SW `Clients interface`, where messages can be sent to specific or all clients. Clients represent an executable context[12] within a browser, like Worker[13], SharedWorker[14] or WindowClient[15].

For the implementation for the offline functionality the `Clients interface` is used, where the SW can send messages to all the clients. These are auto-discovered by the SW and can be messaged with `client.postMessage()`. To filter the clients the SW uses `clients.matchAll(...)`-method. To receive messages, the SW listens on the *message*-Event with an even-Listener, see listing 8.

Where the *callBackfunction* is a placeholder for the handling logic. Within this handling logic the SW registers the routes for Workbox with a `registerMessageRoute()`-function and else sends the message along to the `MessageHandler`, which handles

[11] https://developer.chrome.com/docs/workbox/modules/workbox-background-sync
[12] https://developer.mozilla.org/en-US/docs/Web/API/Clients
[13] https://developer.mozilla.org/en-US/docs/Web/API/Worker
[14] https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker
[15] https://developer.mozilla.org/en-US/docs/Web/API/WindowClient

```
1  self.addEventListener('message', callBackFunction);
```

Listing 8: Example event-Listener for the message event

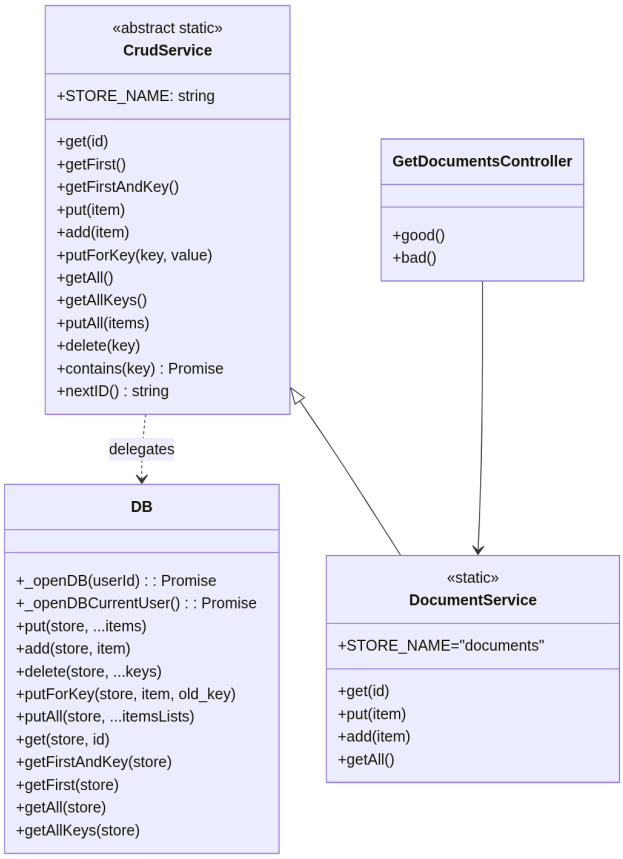

Figure 5.3: Class diagram Controller-Service-DB

the message. The `MessageHandler` is a handler which just sets the *token*, as the auth-token for the SW or resolves the message in a blank `Promise.resolve()` callback.

The `registerMessageRoute()`-function holds the differentiation logic for the different `CacheWorkboxStrategy`-enum that are mapped to the Workbox Strategies in the section 5.1.3. The message that comes from the `CacheStrategy`-decorator is described in section 5.3.2.

5.2 Service Worker Controller-Service-DB layer

5.2.1 Controller

The SW controller are the core to handle responses individually. This is done with help of the `ResponseHelper` class. With the help of this class the routes are registered, where the `ResponseHelper.handle`-method acts as a `RouteHandler`-object for the `registerRoute(...)`-method from Workbox and is the wrapper for the specified controller. Within the `handle`-method the request is first cloned, and then fetched, because requests can only be sent a single time. Depending on what response the network gives, the follow-up strategies are decided. As seen in figure 5.2 the `ResponseHelper` class executes either the `controller.good()`-method for a successful response or the `controller.bad()`-method for failure.

Every controller implements these two functions with following purposes:

good() The intention of this function is to save the original entity into the IndexedDB over the service layer described in section 5.2.2. The structure should return the original element that comes in, but could also pass as an additional hook for the entity before executing the final endpoint request call.

bad() Here a synthetic response is returned. In case the network is unavailable. The SW redirects here with the help of Workbox and over the `ResponseHelper`-class the beforehand registered controller and its `bad()`-method is executed. To create a synthetic response the endpoint knowledge of the actual endpoint returns is needed.

5.2.2 Service

The service layer is the abstraction layer to the `DB.ts`-file. All IndexedDB stores have an own `CrudService` implementation.

In figure 5.3 the `CrudService` provides the abstraction to the specific work object layer. Here the *id* is generated for the work objects that is used as the key for the IndexedDB store. The other functions map the store with the `STORE_NAME` variable for the `CrudService`. The service is used by the controller class to handle the IndexedDB inputs for the *good* and *bad* cases.

5.2.3 DB implementation

The `DB.ts`-file holds the abstraction layer to the IndexedDB for the user-datastores. This file creates the IndexedDB stores for work objects, when they are not existing.

To use the IndexedDB the *idb*-library from *jakearchibald* is used[16]. This small library hold some usability features for the IndexedDB and mostly mirrors the IndexedDB API. In the context for this thesis, this library was updated to version *8.0.3*.

The core work of the `DB.ts`-file is to create the stores, like `openDB()` and do transactions with the IndexedDB file, they map a more extended *CRUD* mechanism, with support for `getAll()` and `keys`.

To create a store `openDB()` uses the *userId* and the *DB_VERSION*. The database is named after the *userId*. To upgrade the IndexedDB, this technology uses a versioning system. That is the reason for the *DB_VERSION*. This variable holds a number to the schema version and thus enables upgrades from one schema to another.

5.3 Main application

To support offline functionality the main application needs some changes. In the following some of them are presented. The `CacheMethodDecorator`, which decorates the QDAcity API endpoints, the Synchronization Workers (Sync Workers) and other modifications, like the PDF IndexedDB to support the offline functionality better.

5.3.1 useHocuspocusProvider

As mentioned in the architecture section 4.3 the Hocuspocus suite is used for the collaboration feature. The implementation on the client side is within the `useHocuspocusProvider.js`-file, which holds the `useHocuspocusProvider`-hook. The provider hook enables to initiate new websocket connections and creates the Yjs documents that are used throughout the main application. This is also the place where the offline support for the Yjs document objects starts. To persist Yjs documents within the browser the Yjs Docs library provides an IndexedDB database adapter: the `IndexeddbPersistence(docName: string, ydoc: Y.Doc)` provider from the *y-indexeddb*[17]. This adapter needs two arguments the docName, which is created by combining the *document* entity id and the *project* entity id, and a `Doc` data structure, which holds the CRDT information. This is simply created with a `new Doc()` object. With this implementation the *y-indexeddb* provider syncs the updates to the IndexedDB and they are thus persisted within the browser.

Another change to support the offline functionality is given by the *load*-state of

[16] https://github.com/jakearchibald/idb
[17] https://github.com/yjs/y-indexeddb

the Yjs document, where the state also needs to be set when the provider is only loaded from the IndexedDB provider.
The state is set to the `isLocalProviderSynced` state, which combines together with the `isOnlineProviderSynced` into the `useMemo` React hook `isLoading` shown in listing 9. The `isLoading` state is for example used to show the loading state of UI components, like the rich-text editor.
The `useMemo` updates everytime either the `isOnlineProviderSynced` or the `isLocalProviderSynced` state changes.

```
1  const isLoading = useMemo(
2      () => !(isOnlineProviderSynced || isLocalProviderSynced),
3      [isOnlineProviderSynced, isLocalProviderSynced]
4  );
```

Listing 9: isLoading State in useHocuspocusProvider.js

5.3.2 CacheMethodDecorator

The `CacheMethodDecorator.ts`-file holds the `CacheStrategy` decorator, which is used to decorate endpoints for Workbox configuration.

```
1  export enum CacheWorkboxStrategy {
2      NetworkFirst = 0,
3      CacheFirst = 1,
4      Controller = 2,
5      UrlCacheFirst = 3,
6      NetworkOnly = 4,
7  }
```

Listing 10: CacheWorkboxStrategy enum

In figure 5.4 one can see the workflow for the `@CacheStrategy`-decorator. And in listing 11 shows an example of the `getDocuments`-endpoint being decorated. The registering starts with the `fetch`-Event of an endpoint. Before the QDAcity endpoint method is executed, the decorator is hooking before the execution.

The decorator sends a *REGISTER_ROUTE*-message, with a *UUID* that is created with the variable called *messageId*. The SW receives on the *message*-event and when this is a message of type *REGISTER_ROUTE*, then the route is registered based on the strategies that are given within the message.

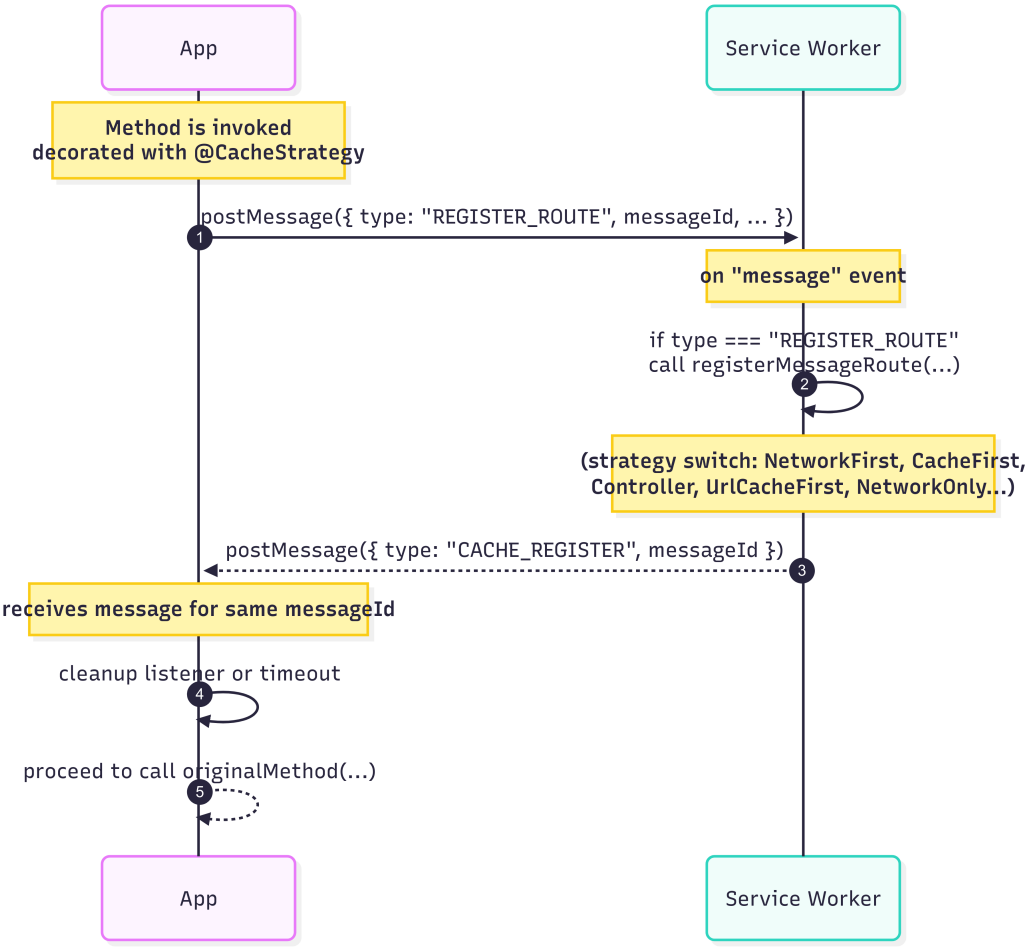

Figure 5.4: CacheStrategy Registration Sequence Diagram

```
@CacheStrategy('documents', CacheWorkboxStrategy.Controller,
    'GetDocumentsController', false)
static getDocuments(projectId: any, projectType: any) {
    const apiMethodCallback = () => {
        return qdacityApiClient.then((apis: { documents: {
            getDocuments: any } }) =>
            apis.documents.getDocuments);
    };
    return
        Promisizer.makeResponseHandlerPromise(apiMethodCallback,
        {
        projectId: projectId,
        projectType: projectType,
    }).then((body: { items: any }) => body.items);
}
```

Listing 11: Example getDocuments endpoint decorated with a CacheStrategy decorator

In the `sw.ts` the `getHttpMethodInfo(tag, operationId)` checks the QDAcity API for the specific endpoint, a *tag* is for example 'documents' where the *operationId* holds the method name from the decorated endpoint as it is the same as the *operationId* necessary for the QDAcity API endpoint.

In QDAcity the Endpoint-API-files on the backend like `DocumentEndpoint.java` decorate the endpoints with an *@Operation()* decorator, which takes the *operationId* as a parameter. Together the *tag* and the endpoint method name map to the target endpoint that can be registered to Workbox.
The `getHttpMethodInfo()`-method encapsulates a call to the QDAcity endpoint, where the API specification comes from the *openapi-$API_VERSION$.json*-file from the `SwaggerClient`.

When *REGISTER_ROUTE*-message was successful the SW sends a *CACHE_REGISTER* message back to all the clients, with a messageId.
Now the decorator, running in the Main Thread, knows that the route has been successfully registered. Else a timeout will trigger and the an error is thrown. This is useful, to know what kind of endpoint has problems with registering or is not configured correctly. After the registering the decorator returns to the *originalMethod(...)* and continues with the normal API call.

The most common used registration strategy is the `Controller`, this handles the endpoints by giving it to the controller-class mentioned in 5.2.1. Else these strategies map to the ones that Workbox provides. These are described in sec-

tion 5.1.3. A notable mention is the `UrlCacheFirst`-strategy that has additional parameters and differs by trying to use the cache first before making a network request.

5.3.3 SyncWorkers

The offline synchronization work starts in the `App.jsx`-file. Where a *useEffect()* React hook registers an event listener that listens on the *online*- and *offline*-events. These events come from the network change on the browser and are fired when the browser loses network connection or reestablishes it. This is implemented by adding a callback function like this: `window.addEventListener('online', setUserIsOnline)`, where the *setUserIsOnline*-callback is then triggering the *startOfflineSync()*-method, which holds an entrypoint to the `offline-sync-worker.ts`.

The `offline-sync-worker.ts` then goes through a list of the registered synchronization services (sync service), which are accumulated within an array. Each service holds a key that is mapped in the *endpointsMap*-Record described in listing 12. This *endpointsMap* holds the mapping information on what kind of API endpoints the different functions declare, to the specific CRUD endpoints of the QDAcity API.
For example the *documents*-key holds the frontend endpoint reference to the *DocumentsEndpoint*, where the create, update and delete functions are mapped in this record.
A sync service implements the `EntitySyncService` interface, which holds the key as *endpointsKey* and synchronization methods for cases: create, update and delete. The entities for the specific key are read from the IndexedDB store. And then processed based on what kind of flags they include within the `offlineMetaData`-field. That is set up by the controller from section 5.2.1. With these fields it is decided, what happens with the entity and which sync-flow is processing it. In listing 13 one can see how the service methods are called depending on the `offlineMetaData`-fields that are set.

Example with DocumentsSyncService.ts

The sync services hold highly specific synchronization logic, in the following an example for the *document* work object is explained.

The `DocumentsSyncService` sync service holds the logic to synchronize *document* domain object. There are multiple dependencies that this service is touching some of them include, what document endpoint, the Yjs document persistence and React Contexts.

```
const endpointsMap: Record<string, any> = {
    documents: {
        endpoint: DocumentsEndpoint,
        createFuncName: 'insertDocument',
        updateFuncName: 'updateDocuments',
        deleteFuncName: 'removeDocument',
    },
    uploads: {
        endpoint: UploadEndpoint,
        createFuncName: 'insertUpload',
    },
    ...
};
```

Listing 12: EndpointsMap Record

This specific sync service holds different workflows depending on the *document* type. The types currently supported are:

TEXT type, where the document holds plain text.

PDF type, where the document is a PDF file.

For a document of *TEXT* type, that was *created* without network connection, the sync service entrypoint is thus the *createSync*-method, which then decides on the type, what kind of flow the entity from the IndexedDB need to go through. For the *TEXT* type the `handleDocumentCreate()`-method is used. Here the flow is executed based on a key that is set up in this method. This key maps over the *endpointsMap* and links the QDAcity API endpoint file `DocumentsEndpoint.ts` to the method used for the create-path, in this case the `insertDocument()`-method. Within the `handleDocumentCreate()`-method the entity is thus created by calling the `insertDocument()`-method of the `DocumentsEndpoint`, with the fields from the already offline created *document*. When this *document* has been successful the next synchronization steps can follow.

Syncing Yjs document

Because the Yjs document is created by using an id as a name, the one generated offline does not have the correct naming pattern. This is done by the `useHocuspocusProvider`-hook in the `useHocuspocusProvider.js`-file, mentioned in section 5.3.1. This hook is also creating the Yjs documents, when there is no network connection. As there is no possibility to create a *document* object with a specific id, the controller from section 5.2.1 should create a synthesized id for

```
async function asyncDoStoreSync(storeName: string, service: any)
↪ {
    let entities = await DB.getAll(storeName);

    // handle create, update, delete syncs based on flags
        //set by the controller
    for (let entity of entities) {
        if (entity.offlineMetaData?.created) {
            await service.createSync(entity);
        } else if (entity.offlineMetaData?.updated) {
            await service.updateSync(entity);
        } else if (entity.offlineMetaData?.deleted) {
            await service.deleteSync(entity);
        }
    }
}
```

Listing 13: Example entity loop for synchronization based on offlineMetaData

created offline entities.
The resulting id mismatches with domain objects from the backend. For that reason the Yjs documents need to synchronize.

This is resolved by synchronizing a newly created IndexedDB database adapter `IndexedDBPersistence(...)` with the offline created `IndexedDBPersistence`, by creating these two providers.
When the offline created (dirty) `IndexedDBPersistence(...)` has reached a *synced*-event, the update is first encoded to a `Uint8Array` binary array by calling `Y.encodeStateAsUpdate(<doc>)` on the dirty Yjs document and then copied to the new provider with `Y.applyUpdate(<newDoc>, <update>)`. This enables to have the older data moved to a new Yjs document.
When the `useHocuspocusProvider`-hook uses this newly created database adapter, it will automatically get the data, because of the now matching provider name. To conclude the Yjs document sync the now obsolete dirty provider is cleared and is deleted. This is done by calling `provider.clearData()`, when the new provider has finished syncing.

Yjs document dependencies

Within QDAcity the Yjs document implementation covers both the *Project* domain object and the *Document* domain object. As noted in table 4.1 the *Code* and *Coding* also combine into the core objects for QDAcity. In a detailed look these objects are also covered by the Yjs document collaboration and are saved

within the Yjs document for the *Project* object. Still the *Coding* holds a reference to the position within the *Document*.
For the offline functionality this is a problem, because as mentioned in section 5.3.3, the *id*-field is different between offline and online created Yjs documents. This raises the need to also change the *id* within the *Project* related Yjs document. To change the *documentId* within this Yjs document a similar approach to the Yjs document syncing is covered. First one creates a new `IndexedDBPersistence(...)` provider to get the Yjs document information. The *Codings* are saved within the *Project* Yjs document as a map and is extracted from this by using the `projectDoc.getMap('codings')` operation. This returns a `YMap` object that can be iterated over. To map the *Project* Yjs document codings from the old *documentId* to the new Yjs document a transaction is executed, when the provider fires the *synced*-event. This can be seen in listing 14. Here the *codingsMap* is iterated and the specific *coding* changed within the transaction.

As this is a transaction the *updates*-store still hold references to the Yjs document, because the websocket-connection there includes direct hooks on this connection. That means, that as soon as the connection fulfills and the Yjs documents, that are persisted with the CES, start syncing with the ones from the client, these Hocuspocus-hooks take action.

This causes another problem because the *saturation*-calculation for the *document* is based on the *ProjectLogs*-endpoint that are filled by these hooks. This happens within `handleCodeChanges(...)`-function that is called for the *afterTransaction*-event on the Yjs document *document*. To solve this issue, the `invokeProjectLogBuilder`-function is extended by a filter *checkInputAndOmitFunction()*-function to omit the faulty offline *codings*.

```
1  projectDoc.transact(() => {
2      Array.from(codingsMap.values()).forEach((value) => {
3      const coding: any = value;
4      if (coding.toJSON().documentId === oldDocument.id) {
5          coding.set('documentId', newDocument.id);
6      }
7      });
8  });
```

Listing 14: Transaction from old documentId to new in Project Yjs document

React state cleanup

In React the state for work objects is saved by using the `useState`-hook. With this hook components can save state variables. QDAcity uses a React Context/Provider pattern for this. That allows consuming components to subscribe to the context that is given by that provider. When the state changes by normal app usage, e.g. an object create call. The state will change based on the return data of that create call. As sync services are not within the standard app flow and also don't run within the direct component tree under the providers. There is a problem of how the state changes, when the synchronization happens.
To resolve this issue the providers *open* the context to changes, this is done by adding event listeners, that listen to `CTX_UPDATE` messages. Within the event listeners callback function the context change is executed. In listing 15 an example of such a context change function logic is shown.

Within the sync service a `CustomEvent` is dispatched that holds the new context information within the *payload* variable. In case the provider or context does not exist anymore, because the React context just lives as long as the component lives, the event is going into empty space, which does not matter because the change target also does not exist anymore.

```
1 setDocuments((docs) =>
2     [...docs, toAdd].filter((d) => d.id !== toRemove.id)
3 );
```

Listing 15: Example documents context change function

5.3.4 Additional adaptations for offline functionality

This section describes additional changes that were added to support offline functionality. The offline capability of QDAcity is a legacy feature, that was neglected due to architectural changes. With that, problems arise that don't fit into the schema of creating a synthetic response and then synchronizing it, that was described in the chapters previously. One such problem is the upload of data, like Portable Document Format (PDF) files, where the challenge was to create another abstraction layer for the uploaded chunks. Another problem was the document editor itself, where legacy document workflows created needs, to adjust the implementation for offline support. In the following the descriptions of these two workarounds are shown.

IndexedDB File Upload Extension

As seen in the figure 4.4 the `pdfStorage` database was added. The reason for that is to create a new abstraction layer in between the upload and the document editor.
In the file upload workflow for QDAcity a URL is first fetched from the API endpoint `DocumentsEndpoint` as a *signed URL*. This *signed URL* is a URL from the GCS endpoint and needed to upload the file. After receiving this URL the app can then upload chunks of data, and receive feedback if this upload was successful. For an upload without network connection, this is obviously not feasible. That's why the `pdfStorage` IndexedDB database support was added. The workflow changes to uploading the file to the IndexedDB and uploading that sequentially afterwards over the *signed URL*-workflow.

When downloading the same thing happens, first a signed URL from the GCS is fetched and then the file blobs are downloaded based from that URL. The interesting part is that every time a file is set active with `setActiveDocument()` in the `CodingEditorProvider.jsx` the app, downloads the blob from the GCS with a `downloadDocumentBlobs()`-wrapper method. That means for every time the `setActiveDocument()`-method was called, the file was downloaded again over the network.

Adding the *CacheStrategy* `UrlCacheFirst` just resolves this issue. The *signed URL* is cached by Workbox and the new fetch pulls the data from the cache.

But for true offline support there is a more complex logic needed. The file upload is not supported in this way. Therefore the files got another abstraction layer in the form of an IndexedDB. Now the upload / download routing happens first over the IndexedDB and then to the GCS. The IndexedDB deals like a proxy in between. The `DocumentSyncWorker` then also handles the upload part in the synchronization. When the app has network connection again the upload runs with the *signed URL* logic to the GCS.

Slate Editor Normalization Workaround

The Slate[18] is a powerful rich text editor framework and chosen as the editor within the UI for QDAcity in the frontend client.
To support the offline capability this editor needed some change. This section is having a detailed look into this workaround.
Because the legacy documents creation system were used with HTML-formats, the document endpoint is still created with an empty paragraph `'<p></p>'` as the *text*-field. This paragraph is passed to the backend, where the CES converts it into a space-char string. The reason for that is unclear, but the Slate editor

[18] https://docs.slatejs.org/

needs a root node, which is the starting point of an editable document. Here a space-char can fit.
In the case of being offline, this roundtrip to the backend does not happen.
Therefore a normalization-function was introduced that could create a root-node, in case the document for the Slate editor does not have one. This workaround led to the addition of a newline, when the sync services synchronized the YDocs as mentioned in section 5.3.3. The `'<p></p>` created input that should be empty and therefore was removed, where an empty text was set in its place, that does not create space characters from that paragraph.

To fix this issue the text document is set to an empty string, where the normalization-method can now add the root-node for the Slate editor.

This workaround also caused additional changes in the AT. Here the `TextEditor.java` Selenium component, removes this additional space-char, when asserting the document length. This is not needed anymore with this workaround.

5.4 Testing

This section describes how the testing is implemented for the offline functionality. The main testing framework is Selenium and its AT capabilities.

5.4.1 Offline Acceptance tests

In QDAcity the testing framework for these automated tests is *Selenium*.
Selenium supports multiple common browsers, like Edge, Chrome, Firefox etc.
Tests are grouped into suites, and are loaded in the Maven `pom.xml`. For QDAcity the *acceptance-test-online* suite is loaded for the AT in the `pom.xml`.
This links to the suite configuration file `AcceptanceTestSuiteOnline.java`, which divides online and offline tests into two separate packages.
In figure 5.5 one can see how the `OfflineAcceptanceTest` is inherited from the `AcceptanceTest`, this decouples the original ATs.
Importantly the offline AT implements a webSocket handler that is used to send Chrome Devtools Protocol (CDP)[19] commands to the browser. The section 5.4.2 explains the CDP in detail.
To note here is that `OfflineAcceptanceTest` *just* supports Chromium-based browsers, because of this feature.
The chosen browser is *Chrome*. This was designed, because offline testing suffices for one browser and also setting browsers "*offline*" works slightly different for each browser type.

Testing offline capabilities could also be achieved by shutting the servers down.

[19]https://chromedevtools.github.io/devtools-protocol/

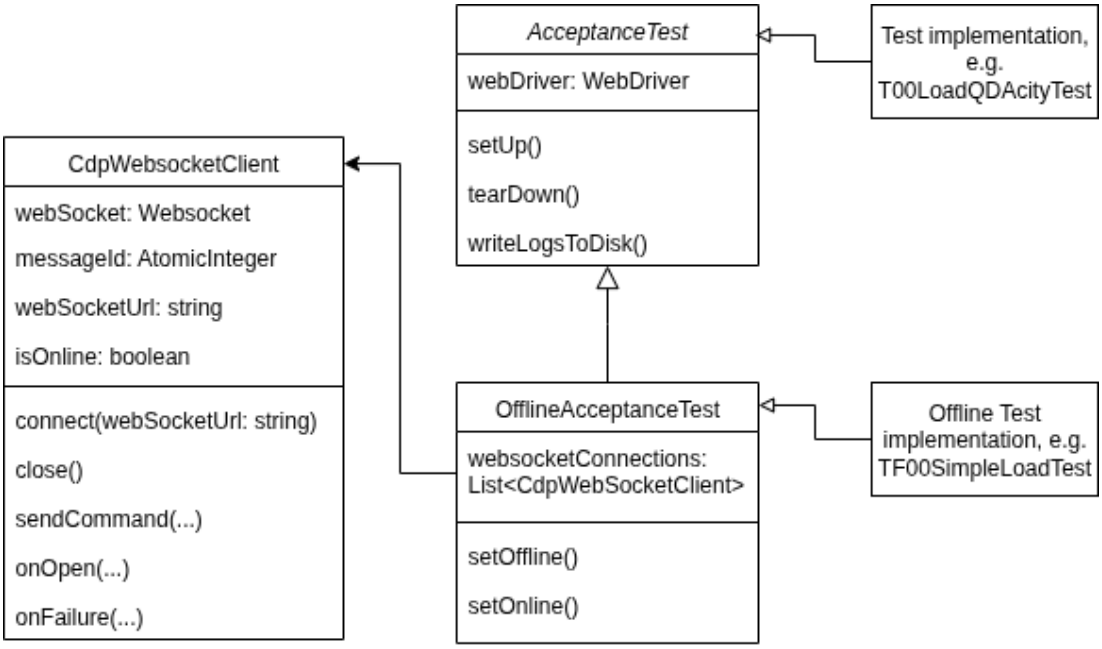

Figure 5.5: OfflineAcceptanceTest Class Diagram

But there are too many disadvantages to this approach, e.g. the server needing to have some kind of endpoint for switching them off. Or the GitLab CI/CD pipeline needing root-access for `kill`-commands. This holds problems, because one can't have root-access easily within a GitLab CI pipeline, especially when they are managed/shared pipeline runners. And also the most important reason that testing short terms of having no network connection and then getting online again would take a lot of time until the backend servers are running again.

5.4.2 CdpWebSocketClient

In Chromium-based browsers DevTool UI interface uses the CDP[20] to debug browsing contexts[21]. Through the CDP, developers can, for example, inspect elements, monitor performance, or modify network settings.

The CDP works by sending a *Command* to a *Target* over a *Session*:

A **Command** is an input for the CDP that control the CDP functionalities for the browsing context.

A **Target** represents a browsing context, where a command can be sent to like tab, window, iframe, SW etc.

[20] https://chromedevtools.github.io/devtools-protocol/
[21] https://developer.mozilla.org/en-US/docs/Glossary/Browsing_context

A **Session** is the open connection to a browsing context. Every session is mapped with a unique id.

To note here commands are just valid within a session, they are thus *session-scoped*. As soon as the session ends or breaks, the default configurations are re-enabled by the browser.

The CDP support for Selenium is just rudimentary. It can be noted that the framework just supports the CDP temporarily till the implementation of the BiDi bridge[22] has finished. The Selenium team tries to give support to the CDP as applicable, but it is removed in the future [23]. Selenium also wraps the CDP functions, but only covers a subset of functionality. Here Selenium makes it hard to support different browsing contexts. With Selenium CDP support on can open `DevTool`-sessions, but due to the wrappers architecture the support for multiple open sessions is not possible without multi-threading these.

Another problem is that the Selenium framework just supports the last three Chrome versions, the reason for that is that the DevTool-Protocol has no longterm support assurance and can have changes for every version change. As the rudimentary support and documentation for the *network emulation* did not bring a successful Selenium implementation, another way was devised and lead to directly connecting the browser CDP with a websocket connection.

But as arbitrary debug websocket connections are not allowed one needs to add the `--remote-debugging-port` argument to access the CDP. This is done by adding this Chrome option in the `WebDriverHelper.java`-class.

The class `CdpWebSocketClient` provides a light weight interface to establish a websocket connection with the Chromium debugger. The implementation uses the OkHttp library[24], which is a HTTP client for Java and offers support for websocket connections.

Upon connection, the client maintains an atomic counter to generate unique message identifiers, as required by the CDP specification[25]. The response for the messages will have the same id from the websocket, that makes it possible to assign the responses to definite targets. Messages without an id are protocol events. Each CDP command is serialized into JSON via Jackson[26] and transmitted to the connected websocket endpoint.

The CdpWebSocketClient is split into three main functionalities:

- **Connection management:** The client exposes a `connect()` method to

[22]https://www.selenium.dev/documentation/webdriver/bidi/
[23]https://www.selenium.dev/documentation/webdriver/bidi/cdp/
[24]https://square.github.io/okhttp/
[25]https://github.com/aslushnikov/getting-started-with-cdp/blob/master/README.md
[26]https://github.com/FasterXML/jackson

initiate a connection to the WebSocket URL obtained from the Chromium debugger endpoint. The debugger url is the combination of the *remote-debugger-url* argument and the *localhost* url. For example a `GET` request onto `http://localhost:9222/json` lists all available websocket for the debugger. The `onOpen` callback marks the client as ready for communication, it sets the *isOnline*-boolean and the *websocket*-variable. While the `close()` ensures a proper shutdown of the connection.

- **Command execution:** Commands are dispatched through the wrapper `sendCommand(...)` method in the `CdpWebSocketClient`. Each command to the websocket consists of an incrementing `id`, a `method` string specifying the CDP domain and a `params` map holding the command arguments. The command is serialized using the Jackson `ObjectMapper` and its `writeValueAsString()`-method and then transmitted over the active websocket.
- **Error handling:** At least failures during connection or message exchange are logged via the `onFailure()` callback

Integration into OfflineAcceptanceTest.java

The `CdpWebSocketClient` integrates into the `OfflineAcceptanceTest.java` file within the `setOffline()` and `setOnline()`-methods.

The `setOffline()`-method first waits for the invisibility of the a NavBar message that is set when the offline synchronization has finished in the QDAcity web app. It is found with the id *offlineSyncFinishedStatus*. The reason for that is that by setting the web app online again, the sync services start. These services also change the context, which in combination also changes the DOM. This causes tests to have a *StaleElementReference*, because tests are referencing elements with a pointer and these are not showing to the correct element anymore, when the DOM changes.

Another issue is that the SW is another target separate from the *page* target that is represented the QDAcity web applications tab. Therefore the command to emulate the offline network connection `Network.emulateNetworkConditions` needs to be sent to all targets. To do that the first step is get the *webDebuggerUrl* for the page, this is done by connecting onto the remote debugg url with the *windowHandle* from the webDriver, seen in listing 16.

This provides a session to the page target. With that two commands are sent to the websocket to emulate an offline connection. The first is to enable the network domain by sending a `Network.enable` command. After the *Network* domain for the CDP is enables one can send a `Network.emulateNetworkConditions` to the page target. The parameters for the network emulation are build within

```
1 "ws://localhost:9222/devtools/page/" +
  ↪ webDriver.getWindowHandle()
```

Listing 16: Example remote debugger DevTool websocket URL

the `getNetworkConditionsForWebsocket`-method, where one can set parameters like *latency*, *downloadThroughput*, *uploadThroughput*, *connectionType* or an *offline*-boolean.
These are hard set for simplicity reasons and just support a simple online/offline concept.
When this is done, the page tab is offline. But requests to the backend are still going through because even if the page has an emulated connection the SW acts as a proxy.
For that reason one needs to also emulate an offline connection for the web worker targets. This is done by first getting a target list from the remote debugger endpoint and then filtering through for a *service_worker* type. To support the filtering for the response a `TargetEntry`-record is used. This record includes all the fields for the targets. Also an `ObjectMapper` is used again to deserialize the response and serialize request back to JSON. The mapper also configures some features, so that it does not break on unknown properties or null values. When the filtered targets are known the same procedure as for the *page* target is executed. First is to enable the Network domain and then send the emulation request.

To set the targets online again, the `setOnline()`-method just closes all the websockets that have been accumulated within the websocketList. The list was filled for every emulated websocket. Because the commands are session-scoped, the default domain configurations are resetted. Atleast the `setOnline()`-method also waits for the sync services to have been finished, and the visibility for the NavBar text notification.

GitLab CI/CD pipeline

To support the webDriver within the gitlab pipeline one needs to augument the webDriver by the `Augumenter`-class. This augumentation enhance the interfaces[27], and make it possible to run the remote-debugger within the pipeline. This enables the RemoteWebdriver to pull in the implementation for all interfaces that match its capabilities[28].

[27] https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/remote/Augmenter.html
[28] https://www.selenium.dev/documentation/webdriver/drivers/remote_webdriver/

6 Evaluation

This chapter will go through evaluate the functional and non-functional requirements, that were defined in chapter 3. These are assessed to determine if the implemented solution has been fulfilled. To note is that the QDAcity implements a lot of endpoints and require careful considerations for the synchronization. This evaluation reflects about the API endpoints that are migrated to support offline functionality.

6.1 Evaluation of functional Requirements

The evaluation for the functional requirements is split into four parts covering the sections defined in the section 3.1. The table 6.1 shows an overview for these requirements.

Type	Requirement	Not met	Partial met	Fulfilled
General	FRQ-1			X
	FRQ-2			X
Offline Support	FRQ-3			X
	FRQ-4		X	
	FRQ-5		X	
	FRQ-6			X
	FRQ-7		X	
	FRQ-8			X
Persistence	FRQ-9			X
	FRQ-10		X	
Synchronization	FRQ-11			X
	FRQ-12			X
	FRQ-13		X	

Table 6.1: Overview of functional requirements status

6.1.1 General

FRQ-1 and FRQ-2 describe two requirements that give the user feedback within the UI. FRQ-1 is shown as an text indicator within the navigation bar, if the user is offline. Additionally here there is also an indication for the synchronization status. This can be seen in the appendix in figures 2 and 3 For that reason the FRQ-1 is fulfilled For FRQ-2 the app shows a UI popup, when the user / app sends a request that is not supported. This popup shows the user that this request is not supported yet. And fails the request. This UI needs to be setup in the controller and decorated API endpoint, like described in section 5.3.2, this is also stated in the requirement thus FRQ-2 is fulfilled.

6.1.2 Offline Support

To evaluate the offline support requirements the app was tested by using the Chrome Devtools network emulation capabilities. With this network emulation the browser can be set into offline mode.

FRQ-3 is fulfilled and is easily explained as the user first needing to log in to get to the core business logic for QDAcity. Also the offline functionality, even though the *"offline"* indicator in the navbar is shown when the network loses the connection, the logic is not affected by that. To enter and edit the data of domain objects user first need to configure the offline functionality for the specific endpoints. FRQ-4 is still just partially met, as not all domain objects are fully supported yet.
Supported functionalities are:

- create, updated, delete a document, as `TEXT` or `PDF` type
- create, update, delete a code
- create, update, delete a coding

FRQ-5 deals with endpoints like `AnalyticsEndpoint.logEvent` as these are also not fully configured this requirement is also partially met. The implementation to meet this requirement is the Workbox background-sync tool. This is described in section 5.1.3.

FRQ-6 is fulfilled, as this is the standard Workbox handling feature for the endpoint. The problem here lies that not all endpoints are migrated yet to support all `GET` endpoints. It is still fulfilled, because the network returns an error in all cases not supported yet.

FRQ-7 is partially met. Not all requests are supported yet. The *document* domain object is for example checked with the AT `TF03DocumentsTest.java`. Where the tests check for the connection loss.

FRQ-8 is fulfilled. The SW is registering. One can see that within the DevTools console. The SW is printing a *Service worker registered*-message, which is sent when the SW is activated.

6.1.3 Persistence

FRQ-9 is fulfilled. As the SW is storing data in the IndexedDB or in the cache. This can be seen within the DevTools, under the *Application*-tab one can see the *Cache storage*, used for the site data and `GET`-requests. A screenshot from the QDAcity app can be seen in figure 4. The underlining architectural description for this is noted in section 4.4.

The UI indication for domain objects is just partially met (FRQ-10). There exists a *'*'* indication for the *document* domain object. But not for the editor or other domain objects that use the Yjs Doc as base for their persistence.

Another indication that is not within the UI can be found within the IndexedDB user store, where the domain objects have the *offlineMetaData*-field that is set by the controller and where one can see, what *should* happened for this object.

6.1.4 Synchronization

Once the connection re-establishes the connection, the app starts the synchronization process in the render loop as shown in chapter 5, within section 5.1. To show working synchronization AT are used. Another test is to manually delete all the storage data, from within the DevTools. With this approach the client needs to reload everything from the server. This shows that the changes for the synchronization have been persisted onto the server. For that reason FRQ-11 is fulfilled.

To check FRQ-12 the app was tested manually on a Chrome browser [Version 139.0.7258.127 (Official Build) (64-bit)]. Here about 15 documents were added offline, the connection was emulated to offline and then back, so that not all documents managed to be synced. When re-establishing the connection the synchronization continued. And the rest of the missing synchronizations resumed. The figure 5 shows the states of *documents* for this workflow shortly after re-establishing and loosing the connection again. The connection loss mid-sync the synchronization throws an error, within the synchronization loop this error causes the connection to not clean up the domain objects that have not been synced. And starts resyncing, when the app goes online again. Thus FRQ-12 is fulfilled.

FRQ-13 is partially met. Because the CRDT strategy just works for all domain objects that use the Yjs library it fullfills for others not, where an independent sync strategy is needed. This holds for all domain objects that can be updated

via a controller. For offline created domain objects there is no conflict resolution, because the created work objects don't fall into a conflict.

6.2 Evaluation for Non-functional Requirements

The non-functional requirements checked against the architecture and mostly discussed as the topics for these are not hard testable. The table 6.2 shows the overview for these.

Type	Requirement	Not met	Partial met	Fulfilled
Technological	NFRQ-1			X
	NFRQ-2			X
	NFRQ-3			X
	NFRQ-4			X
	NFRQ-5			X
Maintainability	NFRQ-6 a)		X	
	NFRQ-6 b)		X	
	NFRQ-6 c)	X		
	NFRQ-7	X		
	NFRQ-8		X	
Testability	NFRQ-9		X	
	NFRQ-10			X

Table 6.2: Overview of non-functional requirements status

6.2.1 Technological Requirements

As the decorator pattern is using TypeScript as base all endpoints that use that, need to be converted to TypeScript, that can be seen by the ending ".ts" that is used to annotate TypeScript files. Therefore NFRQ-1 is fullfilled.

NFRQ-2 is also fullfilled as the base of the offline functionality is using the SW and its Workbox implementation as the core for the offline functionality. This is described in section 5.1.

Described in section 5.3.1 is the use of the `IndexeddbPersistence`, which comes from the Yjs Docs library. The sync services also provide support as described in section 5.3.3. The Acceptance Test provide a validation feedback for this. That means NFRQ-3 is fullfilled.

NFRQ-4 and NFRQ-5 are fullfilled as described in the implementation section 5.4.

All technological requirements are met.

6.2.2 Quality Requirements

The ISO 25010:2023 are a mayor metric for the quality of a software implementation. In this section the requirements for quality are discussed and checked against the architecture.

Maintainability

To check the maintainability, the requirements hold some special focus for the architecture. Discussing the clean architecture for NFRQ-6 holds these insights:

(a) As the architecture section 4.4 and the figure 5.2.3 show. The IndexedDB data store are encapsulated by the `DB.ts` interface. This interface is well-defined. Still the `pdfStorage`-IndexedDB does not fit into this category. Here the usage of the IndexedDB is not encapsulated well and still needs some work.

(b) The synchronization service layer deals with all the synchronization for a domain object. The modularity and separability could be enhanced. The IndexedDB is accessed directly and the service layer is more a Proof of Concept than a clean implementation. The synchronization has a clear start and end, it works over messages with the context of the main application as shown in section 5.1.3. But the `pdfStorage` implementation has no clear interface. For this reason this is just partially met.

(c) The communication interface between the SW and the main application is not well defined. There exist message types for the communication between these two, but to create an interface, that is typed. The whole application would need to be migrated to TypeScript to create a nice model layer, that can be used by all components independently. For this reason this requirement is not met.

NFRQ-7 is not fullfilled. The support for offline functionality has also changed some different business logic for the main application, see section 5.3.4 and 5.3.4. If and how the support for offline endpoints also change other business logic in the future is to be seen. For now the requirement is seen with a pessimistic view.

The documentation is given by code comments. For this reason the requirement NFRQ-8 is just partially met.

All in all the maintainability for the implementation is met with a skeptical view. There are a lot of dependencies for supporting offline capability, and changes to business logic can also extend to the synchronization and controller logic.

Testability

The testability was one of the main concerns to make the offline functionality viable for the future product.

To discuss the requirements for this quality, one can check the implementation and architecture in sections 4.5 and 5.4. The requirmement NFRQ-9 is only partially met, because the end-to-end testing currently does not support mid-sync loss.

NFRQ-10 is fullfilled. Acceptance tests are for the E2E testing are running within the GitLab CI/CD pipeline and are integrated into the current test framework of Selenium.

Generally having the offline capabilities tests running automatically within the pipleine will enhance the testability of the app, which is a milestone for this quality.

7 Conclusion

In this final chapter, the work done in this thesis will be shortly revisited and concluded. The original problem question stated to provide a still functioning working environment in QDAcity, when the network connection was lost. This is only partially fullfilled as not all domain objects are fully supported yet. There are still some future ideas for extension to support a more "complete" offline experience. The service-worker and Workbox approach can just cover the support for the API endpoint, but not for architectural problems that are discovered within QDAcity. Some of them include "authentication" for an offline user etc or unseen structural changes like for the Slate-Editor or a PDF IndexedDB support solution.
Still this thesis shows a complete cycle of what is necessary to support and maintain this functionality in the future. It provides insights into what dependencies offline functionality for a collaborative framework have. How to support libraries like Hocuspocus and Yjs docs and how to provide a solution for these.

The end-to-end testing offline support is also something, that other software applications could use, as Selenium support for that is just halfhearted. As complexity for software applications rises, there is a valid need to make heavy investments into supporting the quality. Especially for QDAcity the future testing quality is necessary, as there are still a lot of legacy systems in place that add complexity.

The future for the offline feature is reshaped and workable again with this thesis, as Knauer's work was partially lost because of architectural changes over the time.

Appendices

A File System API Browser Compatibility

Browser compatibility

api.FileSystemHandle

Report problems with this compatibility data • View data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	WebView on iOS
FileSystemHandle	✓ 86	✓ 86	✓ 111	✓ 72	✓ 15.2	✓ 109	✓ 111	✓ 74	✓ 15.2	✓ 21	✓ 109	✓ 15.2
isSameEntry	✓ 86	✓ 86	✓ 111	✓ 72	✓ 15.2	✓ 109	✓ 111	✓ 74	✓ 15.2	✓ 21	✓ 109	✓ 15.2
kind	✓ 86	✓ 86	✓ 111	✓ 72	✓ 15.2	✓ 109	✓ 111	✓ 74	✓ 15.2	✓ 21	✓ 109	✓ 15.2
move ⚠	✗ No	✗ No	✓ 111	✗ No	✓ 15.2	✗ No	✓ 111	✗ No	✓ 15.2	✗ No	✗ No	✓ 15.2
name	✓ 86	✓ 86	✓ 111	✓ 72	✓ 15.2	✓ 109	✓ 111	✓ 74	✓ 15.2	✓ 21	✓ 109	✓ 15.2
queryPermission 🧪	✓ 86	✓ 86	✗ No	✓ 72	✗ No	✓ 109	✗ No	✓ 74	✗ No	✓ 21	✓ 109	✗ No
remove 🧪 ⚠	✓ 110	✓ 110	✗ No	✓ 96	✗ No	✓ 110	✗ No	✓ 74	✗ No	✓ 21	✓ 110	✗ No
requestPermission 🧪	✓ 86	✓ 86	✗ No	✓ 72	✗ No	✓ 109	✗ No	✓ 74	✗ No	✓ 21	✓ 109	✗ No

Tip: you can click/tap on a cell for more information.

✓ Full support ✗ No support 🧪 Experimental. Expect behavior to change in the future.

Figure 1: Browser compatibility for File System API

B Evaluation screenshots

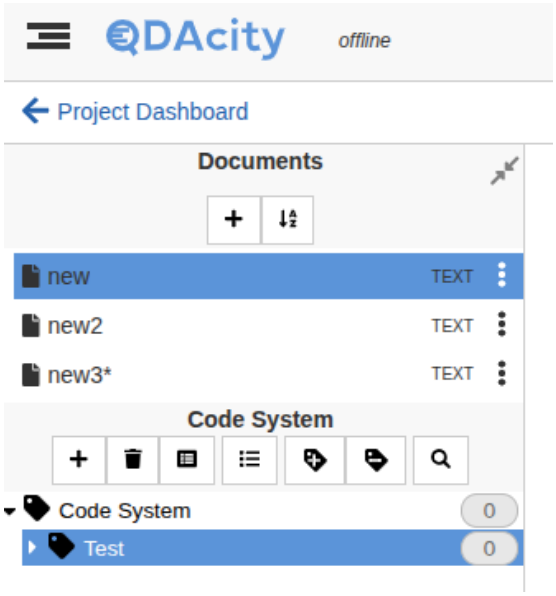

Figure 2: UI offline indicator for documents screenshot

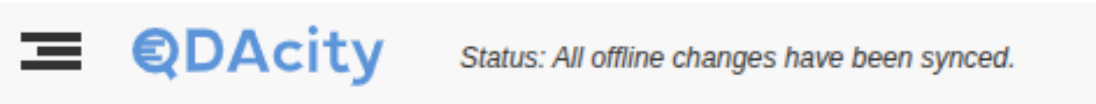

Figure 3: Synced navbar text screenshot

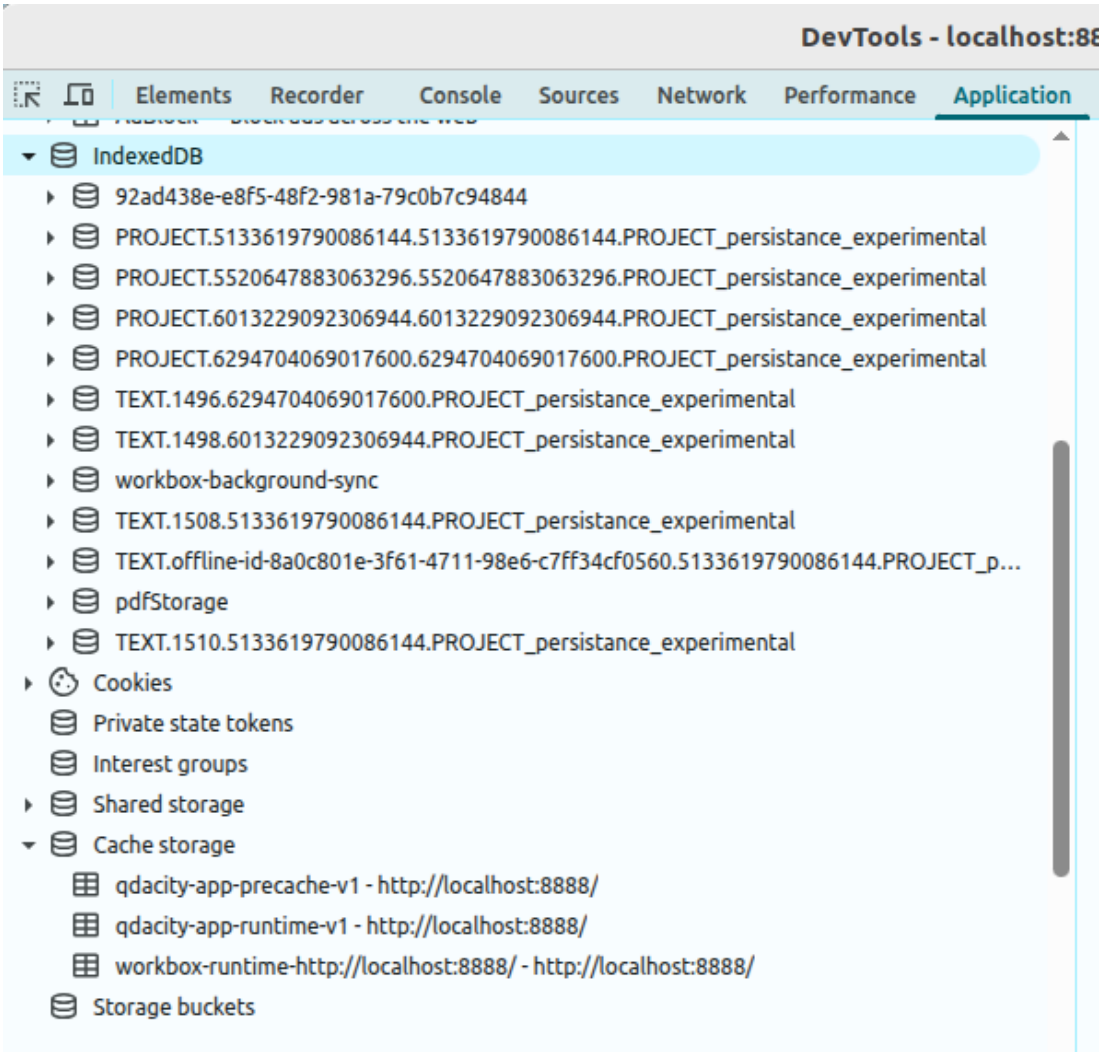

Figure 4: DevTools IndexedDB and Cache Storage screenshot for the QDAcity app filled with data

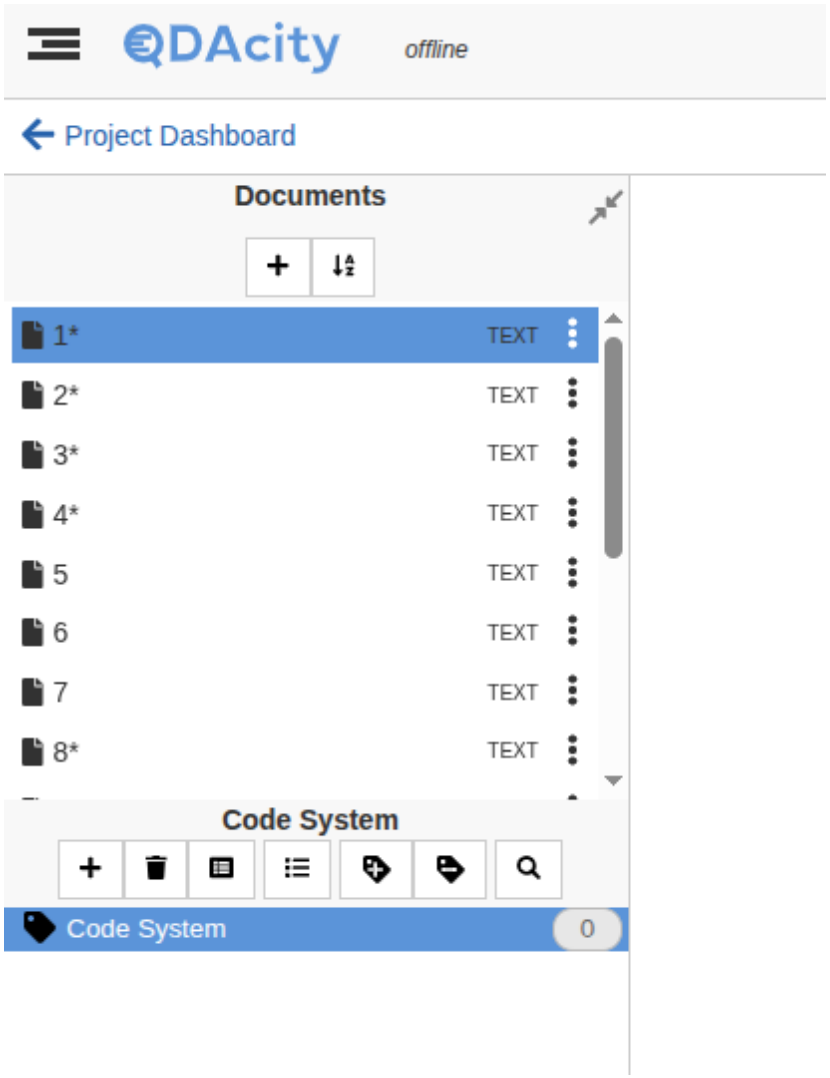

Figure 5: Mid sync connection loss example screenshot

References

Archibald, J. (2014). *The offline cookbook* [Accessed: 2025-08-11]. https://jakearchibald.com/2014/offline-cookbook/

Chromium Project. (2025). Multi-process architecture [Accessed: 2025-08-21].

Grand View Research. (2025). *Progressive web apps (pwa) market size, share trends analysis report by application, by region, and segment forecasts, 2024 - 2030* [Accessed: 2025-08-05]. https://www.grandviewresearch.com/industry-analysis/progressive-web-apps-pwa-market-report

Kaufmann, P. A. (2021). *Domain modeling using qualitative data analysis* [Doctoral dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg].

Knauer, S. (2018). *Offline experience for web applications* [Master's thesis, Friedrich Alexander Universität Erlangen]. https://oss.cs.fau.de/2018/11/20/final-thesis-a-visual-uml-editor-for-qdacity-2-2-3-2/

LePage, P., & Web ev, G. (2020, May). *Persistent storage* [web ev]. Retrieved August 24, 2025, from https://web.dev/articles/persistent-storage

MDN contributors. (2025, April). *Client-side storage* [MDN Web Docs, Learn Web Development]. Retrieved August 23, 2025, from https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Client-side_APIs/Client-side_storage

Mozilla Developer Network. (2025a). *Progressive web apps (pwas)* [Accessed: 2025-08-05]. https://developer.mozilla.org/de/docs/Web/Progressive_web_apps

Mozilla Developer Network. (2025b). *Using service workers* [Accessed: 2025-08-11]. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers

Saito, Y., & Shapiro, M. (2003). Optimistic replication. *ACM Computing Surveys*. https://doi.org/10.1145/1057977.1057980

Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011). Conflict-free replicated data types. In X. Defago, F. Petit & V. Villain (Eds.), *Stabilization, safety, and security of distributed systems*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-24550-3_29

SOPHISTen. *Master schablonen für alle fälle*. 2024.