Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Medieninformatik

Bachelorarbeit

von

Eni **Veshi**

Erweiterung eines webbasierten Zeiterfassungssytems um die Fähigkeiten einer Progressive Web App und eines integrierten Video Streamings

Extension of a web-based time recording system with the capabilities of a progressive web app and integrated video streaming

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Medieninformatik

Bachelorarbeit

von

Eni **Veshi**

Erweiterung eines webbasierten Zeiterfassungssytems um die Fähigkeiten einer Progressive Web App und eines integrierten Video Streamings

Extension of a web-based time recording system with the capabilities of a progressive web app and integrated video streaming

Bearbeitungszeitraum: von 02. November 2021
bis 31. März 2022

1. Prüfer: Prof. Dr.-Ing Christoph Neumann

2. Prüfer: Prof. Dr. Dieter Meiller

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Bestätigung gemäß § 12 APO

Name und Vorname
der Studentin/des Studenten: **Veshi, Eni**

Studiengang: **Medieninformatik**

Ich bestätige, dass ich die Bachelorarbeit mit dem Titel:

Erweiterung eines webbasierten Zeiterfassungssytems um die Fähigkeiten einer Progressive Web App und eines integrierten Video Streamings

selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Datum: 26th April 2023

Unterschrift:

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Ostbayerische Technische Hochschule
Amberg-Weiden

Bachelorarbeit Zusammenfassung

Studentin/Student (Name, Vorname): **Veshi, Eni**
Studiengang: Medieninformatik
Aufgabensteller, Professor: Prof. Dr.-Ing Christoph Neumann
Ausgabedatum: 02. November 2021 Abgabedatum: 31. März 2022

Title:

Erweiterung eines webbasierten Zeiterfassungssytems um die Fähigkeiten einer Progressive Web App und eines integrierten Video Streamings

---

Zusammenfassung:

Diese Bachelorarbeit präsentiert die Erweiterung eines webbasierten Zeiterfassungssystems namens Crew Active mit den Fähigkeiten einer Progressiven Web App und integriertem Video-Streaming unter Verwendung der WebRTC-Technologie. Das Hauptziel des Projekts besteht darin, das bestehende System durch die Integration zusätzlicher Funktionen und Möglichkeiten zu verbessern. Die Architektur der Progressiven Web App ermöglicht es dem System, auch im Offline-Modus verwendet zu werden und Benutzern einen unterbrechungsfreien Zugriff auf ihre Zeitmanagementdaten zu bieten. Die Video-Streaming-Funktionalität ermöglicht Echtzeitkommunikation und Zusammenarbeit. Die Arbeit beschreibt und diskutiert den Entwicklungsprozess, technische Herausforderungen und die Bewertung des Systems. Die Ergebnisse zeigen, dass das erweiterte Crew Active-System eine bessere Möglichkeit zur Zeiterfassung bietet als herkömmliche webbasierte Zeiterfassungssysteme, da es benutzerfreundlicher ist und mehr Funktionen bietet. Die Arbeit schließt mit Vorschlägen für zukünftige Verbesserungen.

Schlüsselwörter: Bachelorarbeit, Erweiterung, webbasiertes Zeiterfassungssystem, Crew Active, Progressive Web App, WebRTC-Technologie, zusätzliche Funktionen, Offline-Modus, Video-Streaming, Echtzeitkommunikation, Zusammenarbeit, Entwicklungsprozess, technische Herausforderungen, Bewertung, Zeitmanagement, zukünftige Verbesserungen.

Summary:

This bachelor thesis presents the extension of a web-based time recording system called Crew Active with the capabilities of a Progressive Web App and integrated video streaming using WebRTC technology. The project's primary objective is to extend the existing system by incorporating additional features and functionalities. The Progressive Web App architecture allows the system to be also used in offline mode, providing users with uninterrupted access to their time management data. The video streaming functionality enables real-time communication and collaboration. The thesis describes and discusses the development process, technical challenges, and evaluation of the system. The results show that the extended Crew Active system is a better way to manage time than traditional web-based time recording systems because it is easier to use and has more features. The thesis concludes with suggestions for future improvements.

Keywords: bachelor thesis, extension, web-based time recording system, Crew Active, Progressive Web App, WebRTC technology, additional features, offline mode, video streaming, real-time communication, collaboration, development process, technical challenges, evaluation, time management, future improvements

Contents

Abbreviations and Acronyms

API Application Programming Interface

CAGR Compound Annual Growth Rate

CLI Command Line Interface

CSRF Cross-Site Request Forgery

CSS Cascading Style Sheets

DOM Document Object Model

GPS Global Positioning System

HTML HyperText Markup Language

ICE Interactive Connectivity Establishment

IP Internet Protocol

iOS iPhone Operating System

JSON JavaScript Object Notation

MERN MongoDB, Express, React, Node.js

NAT Network Address Translation

NoSQL Not only SQL

PWA Progressive Web App

SEO Search Engine Optimization

SPA Single-Page Application

SSR Server-Side Rendering

STUN Session Traversal Utilities for NAT

TURN Traversal Using Relays around NAT

UI User Interface

URL Uniform Resource Locator

WebRTC Web Real-Time Communication

XSS Cross-Site Scripting

Chapter 1

Introduction

1.1 Background

Time recording systems are essential in today's fast-paced work environments. They play a critical role in various industries and contexts, such as project management, freelancing, and remote work, where effective time management is crucial for success. These digital systems help employees track their time on tasks, streamlining workflows and eliminating the need for manual timekeeping. Traditional time recording systems have been web-based platforms, offering solutions only when users are online. This reliance on an internet connection presents a significant limitation, as there may be instances when users need access to their data due to connectivity issues, hindering their productivity. Also, when users want to have a video conference, they often have to use different platforms, which makes their workflow even more complicated. Businesses increasingly seek comprehensive, tailored solutions for their time management and communication needs. These solutions should ideally be accessible offline, allowing users to continue working without interruption and seamlessly integrate video conferencing capabilities to streamline communication. A more robust and user-friendly time recording system can be developed by addressing these gaps, ultimately enhancing productivity and collaboration across various industries.

1.2 Motivation

The motivation of this bachelor's thesis is to extend an existing time recording system with an offline feature so that users can complete the most important tasks even if they are offline and do not have access to the internet. Resolving the problem will be achieved by incorporating advanced web app features. Additionally, the possibility that the user can communicate in real time with video streaming will make the system more engaging and interactive. This approach benefits the business because it will reduce costs and archive excellent efficiency and accuracy in time tracking.

1.3 Objective of the bachelor thesis

This bachelor's thesis aims to extend the Crew Active web-based time recording system by incorporating the features of a Progressive Web App and integrating video streaming through WebRTC technology. The objective is to improve the user experience and give a more inclusive solution for time management and record keeping.

The specific objectives of the thesis include the following:

1. To develop and implement an offline-capable Progressive Web App version of the Crew Active time recording system, addressing the limitations of its current online-only functionality.
2. To integrate video streaming functionality using WebRTC technology, allowing users to communicate and collaborate in real-time.
3. To optimize system performance and address technical challenges during development, ensuring a seamless user experience.

This thesis aims to give a practical solution for businesses wishing to increase the efficiency and accuracy of their time-tracking systems. The findings of this study will contribute to the development of innovative time management tools and provide valuable insights for companies interested in implementing similar strategies.

Chapter 2

Literature Review

This literature review aims to provide an overview of the essential technologies, concepts, and advancements in web application development playing a crucial role in the project context. The project focuses on extending the existing Crew Active web application that leverages modern frameworks and libraries to deliver a seamless user experience, incorporating responsive design, efficient state management, and real-time communication capabilities.

Web Application: A fundamental understanding of web applications, their architecture, and the technologies used for frontend and backend development.

Single Page Application (SPA): An explanation of SPAs and their advantages and drawbacks, focusing on improving user experience through reduced page reloads and seamless interactivity.

React: A popular JavaScript library for building user interfaces, allowing efficient component-based development, performance optimization, and state management.

Redux: A state management library that can be used with React to facilitate centralized and predictable state handling in web applications.

Progressive Web Apps (PWAs): A discussion of PWAs, combining the best web and native applications, offering enhanced performance, offline capabilities, and improved user experience.

Native Application: An overview of native applications specifically designed for individual platforms, providing platform-optimized performance and user experience.

Video Streaming: A review of video streaming technologies, focusing on WebRTC and its widespread use in various industries and applications, such as video conferencing, online learning, customer service, and gaming.

2.1 Web Application

A *web application* is a program [56] hosted on a web server and accessed through a web browser. By entering the web app's URL into a browser, the user triggers a GET request from the browser to the server hosting the web app. The browser reads the static files sent back from the server, including HTML, CSS, and JavaScript.

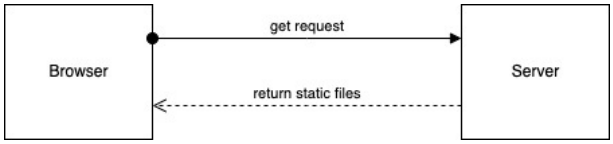

Figure 2.1: Web Application Architecture

Web applications are entirely portable and do not require any further setup. The most used programming languages for web applications are frontend languages like JavaScript, HTML5, and CSS, responsible for the user interface and interactivity, and backend languages like Python, Java, and Ruby, which handle server-side processing, data management, and communication with the frontend. The rise [29] of server-side JavaScript systems like Node.js has also increased JavaScript's usage on the backend. Popular web application frameworks and libraries, such as React, Angular, Django, or Ruby on Rails, assist developers in creating web applications more efficiently by providing reusable code and pre-built components [21]. Responsive design is crucial in web applications, enhancing the user experience across different devices and screen sizes [22]. CSS media queries allow web applications to adjust their layout and appearance for any screen size [36]. Security is a vital aspect of web application development. Following best practices, such as validating user input, securing data transmission, and regularly updating software components, is essential to create reliable and trustworthy web applications [9]. Web applications can be viewed on various browsers and devices, allowing many users to share the same program version and automatically deliver the most recent updates.

2.2 Single Page Application

Single-page applications (SPAs) are online apps that load new content into a single web page as the user interacts with the program instead of loading new pages from the server [53]. Because users are not sent to a new page every time they interact with the app, SPAs offer a more fluid and instantaneous experience. SPAs aim to deliver an outstanding user experience by imitating a "natural" environment in the browser – no page reloads, no extra waiting time [20]. They rely heavily on JavaScript to load content, with just one web page loading all other content using JavaScript. Most SPAs use frontend JavaScript frameworks and libraries like Angular, React, or Vue to build user interfaces and manage the application's state on the client side [34]. These tools provide the necessary functionality to control the state of an application and update pages in real time. However, single-page applications also have some drawbacks. They require a long time to load because the framework of the SPA ships

its JavaScript code with the initial request. Code splitting can improve the loading time by breaking the application into smaller chunks loaded on demand. Another issue is that SPAs can negatively impact SEO (Search Engine Optimization) because they initially deliver an empty HTML file, which takes time to load the JavaScript code to populate the page content. Search engine crawlers often read only the empty HTML file [8]. To overcome this challenge, developers can use server-side rendering (SSR) or pre-rendering techniques to improve the initial page load and make the content more accessible to search engine crawlers [54]. SPAs are particularly suitable for use cases like dashboard interfaces or interactive tools, where the application needs to provide a seamless and responsive user experience with minimal page reloads.

2.3 Exploring React: Components, Performance, and State Management

React is a JavaScript library that allows building user interfaces for web applications with better structure and making many performance optimizations in the background [18]. It was created by Facebook and released in 2013 [16].

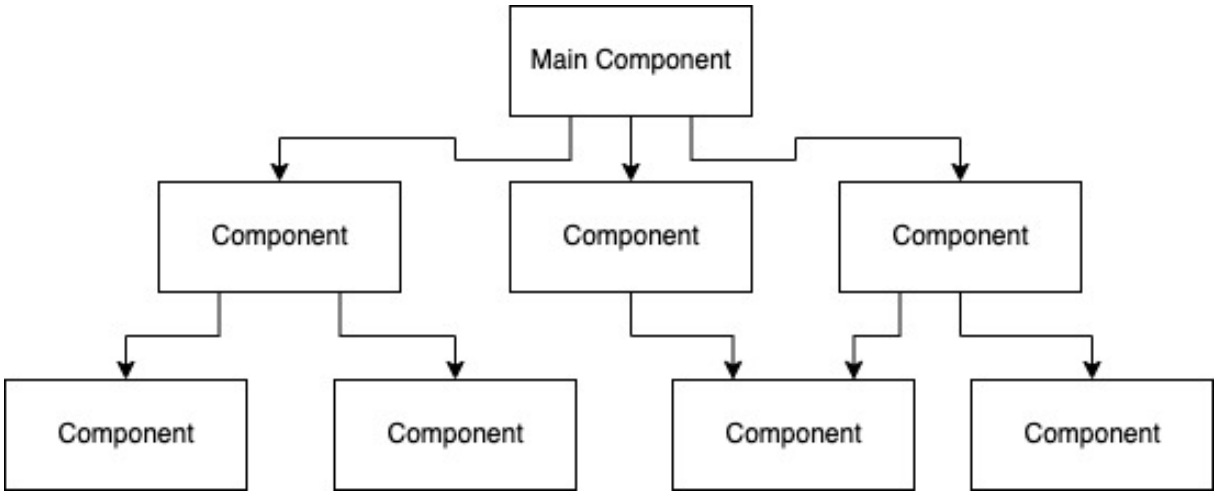

Figure 2.2: Nested React Components

Components are the building elements that form the foundation of React and may be used to create any user interface. To build more complex applications, React has a feature to combine and nest the components and reuse them. As the user interacts with an application, React quickly adds new content to the page using a virtual DOM (Document Object Model). The virtual DOM is only a lightweight copy of the real DOM so it can be updated considerably more quickly than the real DOM itself. React rerenders the components only if their states change [19]. That will improve the application's performance if it has been used correctly. Nesting and reusing components is a good approach, but also it can cause rerendering on components that do not need to rerender.

For small applications, it is enough that developers use the states of React. However, in large applications, it is recommended to use third-party libraries developed to handle

the state problem. One of the best libraries that exist until now is Redux [47]. It will boost the developing process by giving the possibility of a better way to debug the application's state.

React is a library, so it depends on other libraries. For example, another library must be integrated to make routing possible. This allows developers to create and integrate their own libraries as needed. React also has a big community where if problems occur, it will be straightforward to find a solution to that problem.

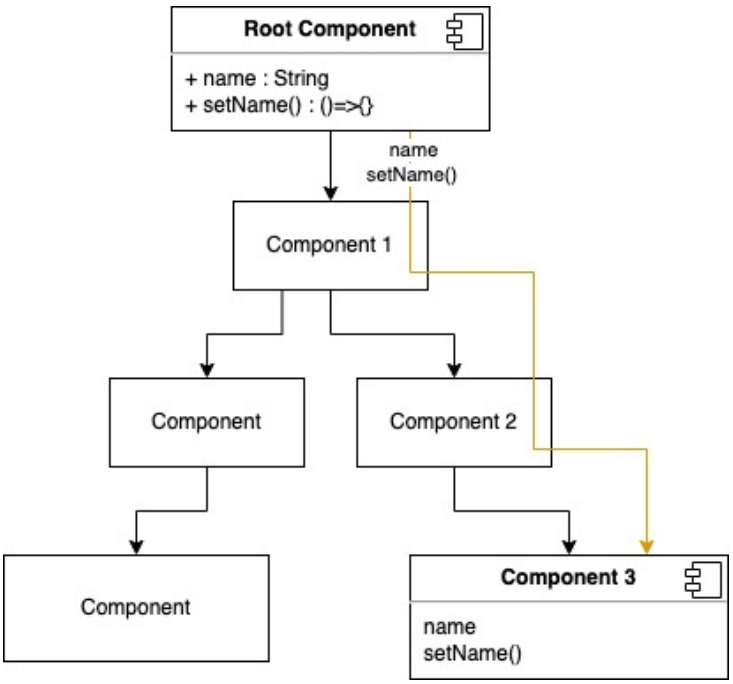

Figure 2.3: React prop drilling

In Figure 2.3, we can see a prop drilling in React. Prop drilling is when a parent component can pass its state or functions to the children's components. In this example, the name will be the state, and the setName() function will be the function to change the name state. If we want to change the name in component 3, we must pass the function setName through component 1, component 2, and then component 3. Changing the state at component 3 will cause a rerender to all components 1, 2, and 3 with the state name passed through prop drilling.

2.4 Redux

Redux is a library for handling application states that may be combined with the React framework [18] to create modern JavaScript apps. They are making it easier to control the state and debug complicated programs by providing a centralized and predictable method for managing the application state.

State management in Redux lets us get to any state from any component we want. We do not need prop drilling, so we avoid a lot of rerenders that the prop drilling

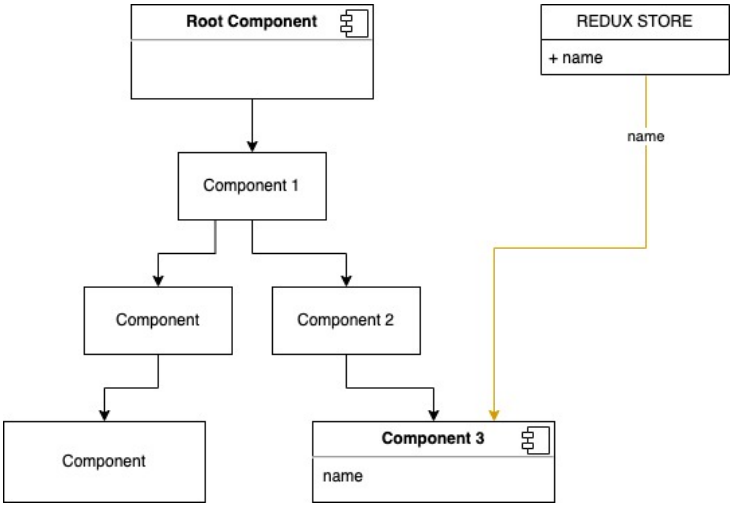

Figure 2.4: Access Redux state from React components

would cause. As shown in Figure 2.4, if we change the name state in component 3, redux will cause to rerender of all components that have the state name. This example in Figure 2.4 will cause only a rerender of component 3. In Redux, the store is a JavaScript object containing the entire application's state. The store must handle all state management and provide an interface for updating and reading the data. Reducers [47, 48] are special functions that update the state by taking the current state and an action as input and returning a new state as output.

An action is a simple JavaScript object representing the goal of modifying the current state. A type attribute specifies the action's nature and any other information that may be required. The Store receives the actions and calls the relevant reducer to change the state.

Redux's middleware, actions, and reducers add features like logging and asynchronous actions.

2.5 Progressive Web Apps

Progressive web apps (PWAs) are a type of web application that leverage cutting-edge web technologies to replicate native app experiences within a web browser. PWAs can be accessed online through a web address, but they can also be downloaded and installed on a user's device for offline usage [12].

Push notifications, offline caching, and background synchronization are just some of the current web capabilities made possible by PWAs. By these characteristics, PWAs may offer consumers a quick and stable experience, even in poor connectivity settings. Furthermore, PWAs can respond to the screen size of their viewer and function well on both desktop and mobile devices.

Many significant organizations, like Alibaba, Pinterest, Forbes, and Tinder, have recently adopted PWAs to improve their customers' online experiences [59].

- Alibaba: Progressive Web App boosts mobile web conversions by 76% for Alibaba.com, resulting in higher conversions across all browsers. Additionally, there was a 14% increase in monthly active users on iOS and a 30% increase on Android. Moreover, the interaction rate for adding to the home screen was four times higher [68].
- Pinterest: Pinterest's PWA resulted in a 40% increase in total time spent, a 44% growth in user-generated ad revenue, and a 60% increase in core engagement [40].
- Forbes: Forbes' PWA mobile site has reduced content rendering time from 6.5 seconds to 2.5 seconds and led to higher engagement rates, with users spending up to 40% more time per session and viewing 15% more pages per session [6].
- Tinder: Tinder achieved significant improvements with their PWA, including more than halving loading times from 11.91 seconds to 4.69 seconds, resulting in increased engagement across the board. Furthermore, their PWA is 90% smaller than their native app [41].

PWAs allow web developers to give their consumers an experience on par with native apps regarding speed and reliability. PWAs use cutting-edge web technologies to provide features such as push notifications and offline caching, which were previously exclusive to native applications.

2.6 Native Application

Native applications are specifically designed and developed mobile apps for individual platforms like iOS or Android [24]. These apps are installed directly on devices, allowing them to fully utilize their capabilities, such as the camera, GPS, and accelerometer. Native apps offer a platform-optimized performance and user experience since they are created using programming languages and frameworks unique to the platform.

Without an internet connection, users can still enjoy the seamless experience provided by native apps. These applications often follow strict security guidelines and can take advantage of device-specific security features, resulting in better security [44]. Developers use languages like Swift and Objective-C for iOS and Java or Kotlin for Android when creating native apps, generally distributed through app stores such as the App Store or Google Play Store.

On the other hand, native app development has challenges, including needing separate codebases for different platforms, leading to more extended development and maintenance times. Furthermore, native apps must be approved for distribution in app stores, which can sometimes be time-consuming or limiting.

Native applications deliver a high-performance and customized user experience, leveraging platform-specific features and optimizations. Although development and distribution may present particular challenges, native apps remain a popular choice for

businesses and developers aiming to provide the best possible experience for their users.

2.7 Video Streaming

Technology to broadcast videos in real-time has been around for a while, giving viewers instantaneous access to visual information. One of the essential technologies for online video streaming is real-time online communication (WebRTC).

WebRTC was initially developed by Google in 2011 and released as open source. Its success as a medium for instantaneous interaction has led to its widespread use ever since [71]. Google Chrome, Mozilla Firefox, and Apple Safari are just a few popular web browsers with integrated WebRTC compatibility, allowing for smoother interaction with web-based applications. WebRTC has established a real-time communication and collaboration standard across many industries and use cases.

WebRTC has seen a significant rise in usage, particularly in video conferencing and remote collaboration, due to the COVID-19 epidemic. Businesses and individuals have resorted to WebRTC-powered applications and platforms to interact and cooperate in real time due to the growing prevalence of remote work and social isolation. According to a report by Grand View Research [49], the global WebRTC market size was valued at 1.74 billion in 2020 and is expected to grow at a compound annual growth rate (CAGR) of 45.2% from 2021 to 2028. The research identifies the COVID-19 pandemic as a significant factor boosting the demand for real-time communication and collaboration solutions.

Applications that use WebRTC include [78, 30]:

Video Conferencing WebRTC is the best option for online meetings and video conferencing since it enables immediate participant interaction. WebRTC's ability to facilitate high-quality audio and video communication has made it a favorite among businesses and organizations searching for effective distance collaboration and communication methods.

Online Learning WebRTC's real-time communication capabilities make it an excellent distance learning and teaching platform. WebRTC facilitates two-way communication, encouraging more student participation in class.

Customer Service Because of its real-time communication characteristics, WebRTC is ideal for supporting needy customers. WebRTC offers real-time contact between customers and helps workers, allowing for a more efficient and effective manner of addressing customer support issues.

Gaming With WebRTC, players may interact with one another in real-time, creating a more enjoyable and interactive gaming experience.

Chapter 3

Progressive Web App

3.1 How Progressive Web Apps Work: A Step-by-Step Guide

Progressive Web Apps are web applications that provide a native app-like experience by incorporating features such as offline access, fast loading, and the ability to be installed on a user's home screen [51]. In this section, a step-by-step guide on how a PWA works is provided, starting from the initial request:

1. The browser sends an initial request to the server for the HTML, CSS, JavaScript files, and Web App Manifest that comprise the PWA.
2. Service worker registration: The application checks if a service worker is registered for the current URL, then a new service worker is registered if there isn't one.
3. Service worker installation: The service worker is installed in the background, separate from the application's main thread.
4. Service worker activation: The service worker is activated and begins to control the pages of the PWA.
5. Asset caching: The service worker uses the Cache API to cache assets and data, such as HTML, CSS, JavaScript, images, and API responses, for offline access. This enables the PWA to load faster and provide a better user experience.
6. First paint: The browser renders the first paint of the PWA, displaying the content to the user.
7. Offline access: If the user goes offline, the service worker continues to control the pages of the PWA and serves the cached data, ensuring a smooth experience for the users, even when they are offline.

The process of a PWA involves initial requests, service worker registration, installation, activation, asset caching using the Cache API, first paint, and offline access. Web App

Manifest also plays a crucial role in making the PWA feel more like a native app. This ensures a seamless and responsive user experience, even when the user is offline.

3.2 Ways to Access Progressive Web Apps

Progressive Web Apps (PWAs) can be accessed in different ways, depending on the user's platform and device capabilities:

- In a web browser: PWAs are primarily accessed through a web browser, just like any other website.
- In a native app container: PWAs can be packaged as native apps and installed on a user's device, providing a native-like experience.
- On the home screen: This approach offers a native-like experience, as the PWA is launched directly from the home screen, like a native app.
- In an app store: PWAs can be submitted to app stores like the Google Play Store, but it may be difficult to get approval from the Apple App Store since it only accepts Progressive Web Apps on a case-by-case basis. However, once a PWA is approved, any future updates do not require review by the App Store, simplifying the update process.

3.3 Fast Loading Speed

Progressive Web Apps strongly emphasize fast loading speeds to enhance user experiences and enable seamless navigation. A fundamental feature that allows PWAs to load quickly is their ability to cache application assets. By storing these assets in the cache, PWAs can load the application without making additional requests to the web server, thus eliminating the need to wait for server responses.

To ensure smooth and speedy loading, developers utilize several performance optimization techniques:

- Lazy loading: This approach only loads a web page's essential components and content when a user first accesses it. Additional content is loaded as the user interacts with the page or scrolls down, reducing the initial load time.
- Code splitting: This technique involves breaking the application's code into smaller, more manageable chunks. These chunks are loaded on-demand, ensuring users only download the necessary code for the current view. This results in reduced load times [43].
- Image optimization: By compressing and resizing images, PWAs can decrease the amount of data that needs to be downloaded, thereby improving load times.

These strategies allow PWAs to deliver fast loading speeds and create a more enjoyable and efficient user experience.

3.4 Offline Functionality

A PWA can also be accessed offline because of its architecture. Progressive Web App has the ability, through a service worker (Javascript file), to run in the background and fetch the cache files for the Application to load when the user has no internet connection.

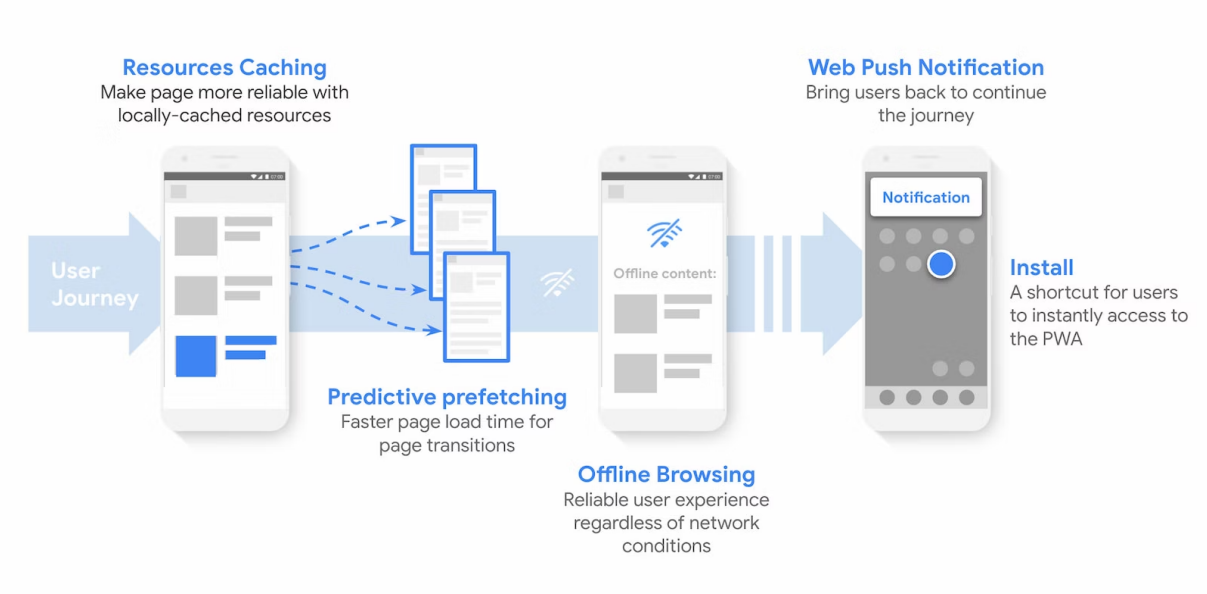

Figure 3.1: Improving the user experience [70]

For example, using a service worker to cache resources and doing predictive prefetching makes the site faster and more reliable. Making the site Installable allows the customers to access it directly from their home screen or app launcher [70]. When the user is offline, the web browser will fetch the static files directly from the cache, giving the user access to the PWA with the last updated caches.

3.5 Add to Home Screen

The "Add to Home Screen" feature of PWAs is essential because it gives the user the feeling that he is using a native application. The user does not have to start a web browser and go to the application's domain because it will be redirected automatically when the icon is clicked. Providing this feature will not make any difference like opening a standard web app because everything runs in a web browser, just that the user will get the feeling that it is opening a native app.

3.6 Push Notifications

Push notifications are crucial in engaging users and enhancing their experience within progressive web apps. Like native applications, PWAs can utilize push notifications to send timely, relevant information to users, even when the app is not currently

open or running in the background [31]. PWAs rely on service workers to manage push notifications. Service workers are scripts that run in the background, separate from the web page, enabling the app to receive and send push notifications across different platforms, such as Android and iOS. This provides a consistent experience for users, regardless of their device. Push notifications can be employed in various scenarios within a PWA, such as alerting users about new messages, notifying them of updates or news, reminding them of upcoming events, or promoting special offers. These notifications help to keep users engaged and informed about important events or updates related to the app. Implementing push notifications in PWAs requires a POST request to the corresponding push notification services of the Apple App Store or Google Play Store, along with a specific device token. This token serves to identify the device rather than the PWA itself, allowing push notifications to be delivered seamlessly. However, there might be some limitations or challenges in implementing push notifications in PWAs, such as platform-specific restrictions or differences in the available APIs. Despite these challenges, push notifications to remain a valuable feature for PWAs, offering users a native app-like experience with timely and relevant information.

3.7 Splash Screen

When launching an application on a device, the splash screen [26] serves as an intermediate frame. This screen is displayed while the application loads all necessary files to build the app, effectively masking the loading time. Just like native applications, PWAs can exhibit static or dynamic splash screens with various animations.

By showcasing a splash screen, the user perceives a shorter loading time for the application to be fully constructed. This enhances the user experience by providing diverse information or a loading indicator during the wait. Incorporating a visually appealing splash screen not only improves the user experience but also establishes a positive first impression of the application.

3.8 Responsive Design

Responsive design plays a crucial role in the success of Progressive Web Apps, as it ensures a consistent and native-like experience across various devices and screen sizes. To achieve this, PWAs must adapt seamlessly to different smartphone, tablet, or desktop displays.

One technique to implement responsive design in PWAs is using CSS media queries. Media queries allow developers to apply different styles and layouts depending on the device's screen size, orientation, and resolution [36]. This ensures that the app's interface remains visually appealing and functional regardless of the user's device.

Managing CSS files can become complex and challenging, mainly fitting to multiple screen sizes and devices. This is where utility-first CSS frameworks, such as TailwindCSS [57], come into play. TailwindCSS offers a collection of pre-defined CSS

classes that can be applied directly to the HTML code, streamlining the styling process. Additionally, TailwindCSS includes a configuration file for managing themes and design standards, making it an organized and efficient solution for responsive design.

When creating a responsive PWA, it's essential to avoid using fixed sizes for elements and instead rely on relative units, such as percentages, to ensure a flexible layout. This allows the app to adapt to various screen sizes and resolutions, providing a consistent user experience across different devices.

Responsive design is a critical aspect of developing successful PWAs. By employing techniques like CSS media queries, utility-first CSS frameworks like TailwindCSS, and using relative units, developers can create PWAs that provide a native app-like experience for users on any device.

3.9 Access to Device Features

Some of the device features [79, 69] that a PWA can access include:

- Media capture: Media capture allows PWA to use the camera and microphone.
- Geolocation: The Geolocation API enables users to retrieve their location.
- Notifications: The Notification API enables PWAs to send and receive push notifications.
- Authentication: Web Authentication API allows passwordless authentication.
- Vibration: The Vibration API will enable PWAs to vibrate a device.
- Contact picker: The Contact Picker API enables PWAs to select users' contacts.
- Network info: The Network Information API provides information about the connection of the device
- Bluetooth: The Web Bluetooth API connects apps to Bluetooth.

Access to device features is a crucial component of PWAs, allowing the application to access and utilize a variety of device characteristics to provide users with a more native-like experience.

3.10 Security

Security is a critical concern for PWAs because they usually handle user-sensitive data. PWAs should follow these practices:

- HTTPS: PWAs, like any other Web applications, have to be served over HTTPS [61] to ensure that all data transmitted will be encrypted and secure.
- Cross-Site Scripting (XSS): PWAs must be designed to protect against XSS attacks. This protection can be archived using modern frameworks like React because they sensitize every input field [33].
- Cross-Site Request Forgery (CSRF): PWAs must be made to protect against CSRF attacks, which involve tricking a user into sending an unauthorized request to a web-based system [28].
- Session Management: PWAs must be designed to manage user sessions securely, including securely storing session information and protecting against session hijacking and other threats.
- Data Storage: PWAs must be designed to store sensitive user data securely, including encrypting stored data and protecting against unauthorized access.

3.11 Discoverability

Discoverability is a critical factor in the success of Progressive Web Apps, as it enables users to quickly find and access a PWA through web searches and links from other sites. Since PWAs are built using standard web technologies like HTML, CSS, and JavaScript, they are easily indexed by search engines, which helps users discover and visit the PWA via web searches.

The role of web crawlers and search engines in the discoverability process is vital, as they index PWAs like any other website. To boost the discoverability of PWAs, developers should include relevant metadata in the HTML headers, such as title tags, meta description tags, and canonical URLs. This metadata provides additional context and information for search engines, which in turn helps improve the PWA's search ranking. In addition, having a well-organized, accessible, and user-friendly website contributes to better discoverability. By adhering to web best practices and optimizing content for search engines, developers can make it easier for users to find and engage with the PWA.

3.12 Service workers

Service workers are a vital aspect of Progressive Web Apps, providing the ability to run JavaScript code in the background, separate from the application's main thread [35]. Service workers allow PWAs to do things in the background, like cache content, handle push notifications, and work offline. Doing tasks in the background gives users a more native-like experience while the PWA is not actively running.

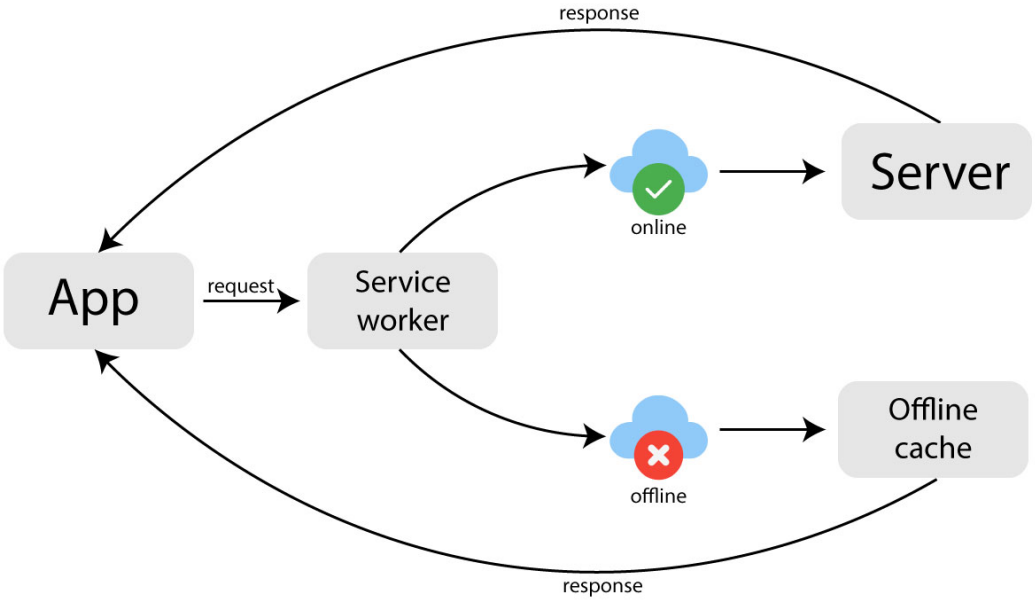

Figure 3.2: Service worker background sync

For instance, a PWA that offers time recording experience can use a service worker to cache tasks or assignments for offline viewing. Caching files enables users to access content offline, making the experience easier and more accessible.

Service workers can also be used to do heavy calculations because they run in the background, so the main threat will not be blocked.

3.13 Service Worker Stages

Service workers undergo several stages during their lifecycle, including registration, installation, activation, idle, and termination [37, 58].

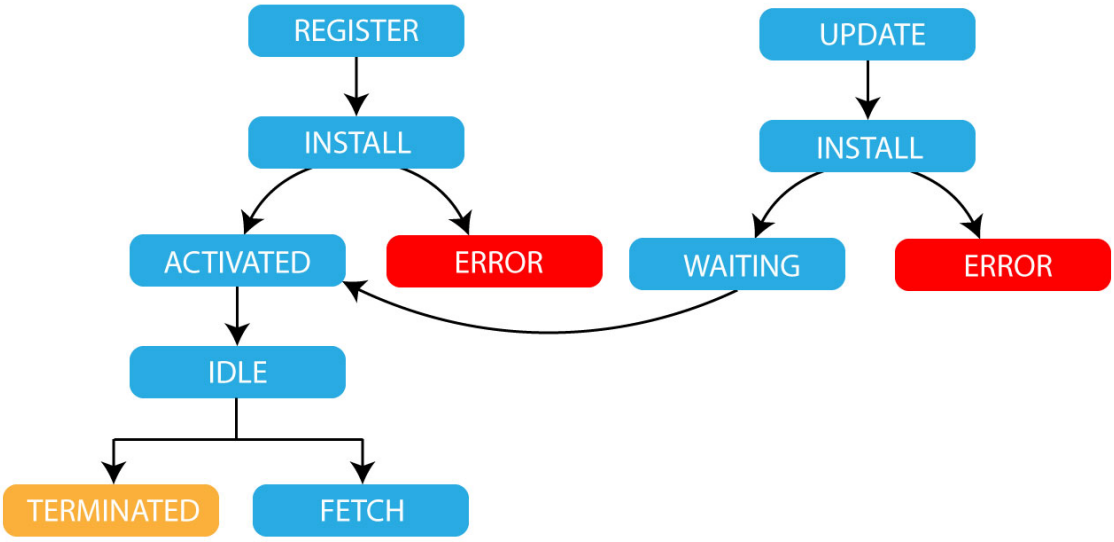

Figure 3.3: Service Worker Lifecycle and Transitions

- This is the first stage of a service worker's lifecycle. The service worker goes through a registration process in the browser by calling the `'navigator.serviceWorker.register()'` method.
- Installation: After the service worker is registered, it performs an installation process in the background. During installation, the browser downloads the service worker script and caches any necessary resources.
- Activation: After installation, the service worker controls the Progressive Web App pages. During activation, the service worker can perform any necessary setup or cleanup tasks, such as removing old caches or updating data.
- Idle: After the service worker is activated, it enters the idle stage. In this stage, the service worker is waiting for an event or message from the main thread of the PWA.
- Termination: A service worker can be terminated at any time by the browser, for example, when the user closes the tab or navigates to a different site. When a service worker is terminated, it cleans up all associated resources, and the service worker is no longer active.

The service worker can only be in one stage at a time. In addition, numerous service workers can be active simultaneously in a PWA, each controlling a unique objective. A PWA can utilize multiple service workers, each with its tasks and competencies.

3.14 Overcoming the Challenges of Updating Service Workers

Updating Service Workers present several challenges [25] and problems. Some of the problems that can occur when updating Service Workers include the following:

- Compatibility Issues: Service Workers can be updated to include new features or fix bugs. However, these updates may not be compatible with older versions of the Service Worker or the browser. Updates can lead to compatibility issues and a poor user experience.
- Stale Cache: Service Workers will cache data and assets to improve performance and provide offline capabilities. However, when a Service Worker is updated, the cache may become stale, making the user see the old website version.
- Broken Functionality: Updating a Service Worker can break existing functionality or introduce new bugs, leading to a poor user experience.
- User Interruptions: Updating a Service Worker can result in user interruptions, such as a broken user interface. User Interruptions can lead to a poor user experience and a loss of trust in the application.

Updating Service Workers might bring various challenges and issues. To minimize these difficulties, it is essential to properly test Service Worker updates before deployment and keep users informed of any potential changes or disruptions. By implementing these measures, PWAs may provide a stable and consistent user experience even after upgrading Service Workers.

3.15 Informing Users of Service Worker Updates

Informing the users before the service worker is updated is crucial to secure the best user experience. Users can be doing essential tasks, so interrupting will be the worst thing to offer the users [5, 80].

There are multiple ways to alert users about new Service Workers versions:

1. Notification: PWAs can send notifications to the user, even when the app is not running, providing an opportunity to inform the user about updates to the Service Worker.
2. Pop-up: A pop-up message can be displayed to the user, informing them about the update and any changes to the app's functionality.
3. In-app message: An in-app message can be displayed to the user, providing information about the update and any changes to the app's functionality.
4. Release Notes: Release notes can be provided to the user, detailing the changes and updates to the Service Worker and any impact on the app's functionality.

Providing the user with clear and straightforward information and directions for any necessary actions, such as refreshing the application or clearing the cache, is always recommended.

3.16 App Shell Architecture

An application shell is the minimal HTML, CSS, and JavaScript powering a user interface [42, 32].

The application shell should:

- load fast
- be cached
- dynamically displays content

PWA consists of static files and dynamic data. The static files, such as HTML, CSS, and JavaScript, remain constant while the user retrieves dynamic data from the server. Upon opening the application, the static files are used to construct the application shell, which includes the user interface. Subsequently, the application fetches dynamic data from the server. Thus, the application requires two separate requests and waits for the server's response. By caching the static files (App shell), the application only needs a single request to obtain user data and display the content. Performance optimization occurs during the user's second visit to the web application, as the first visit entails two requests and caching of the App shell. For the second visit, the App shell is already cached.

3.17 Manifest File

An application manifest is a JSON document that contains startup parameters and application defaults for when a web application is launched [66, 67]. Browsers utilize the manifest to provide a native-like experience for PWAs, such as allowing the PWA to be installed on a user's device and displaying the PWA in full screen.

The Manifest file contains these keys:

- lang: The lang key specifies the language.
- name: The name key is the name of the application.
- short name: The short name key is the shorted application name.
- icons: The icon key will be an array of all different sizes of icons for the application.
- start URL: The start URL is optional and provides the application's starting route.
- display: The display key specifies the display mode, fullscreen, standalone, minimal-ui, and browser.

- orientation: the orientation key specifies the orientation of the device, portrait, or landscape.
- theme color: The theme color specifies the theme color of the app's user interface.
- background color: The background color specifies the background color of the splash screen.
- related applications: The related application key specifies related native apps available in app stores, along with their URLs and platform-specific IDs.

Linking the manifest file is done through an HTML link tag in the document's head.

3.18 Caching Strategy

Building PWAs, need some caching. It could be assets(css, js, icons, images), responses, or a fallback offline page. So choosing strategies that work best for the application is very important. Common caching strategies are [39]:

- Cache Only: The Cache only strategy will always return the cached files. Cache still returns the cached files if the user is online or offline. If the files are not cached, the application will break because it can not access the resources. This Strategy is suitable for serving resources during the installation of a service worker.
- Network Only: This caching Strategy will never cache any files. The application always retrieves the resources from the server. This Strategy is suitable for dynamic data that must always be up to date.
- Cache First: The Cache first strategy, or Offline first, tries to get the resources from the cache first, and when no resources are found, it will retrieve them from the network.
- The Network first or Online first Strategy tries to get the resources from the internet, and when it fails, the application will get the resources from the cache.

Applying the right Strategy is very important because it improves the user experience, but using it wrong will drastically worsen it.

3.19 Browser Compatibility: Challenges and Solutions

Most of the time, progressive Web Applications will function across various web browsers with variable degrees of compatibility.

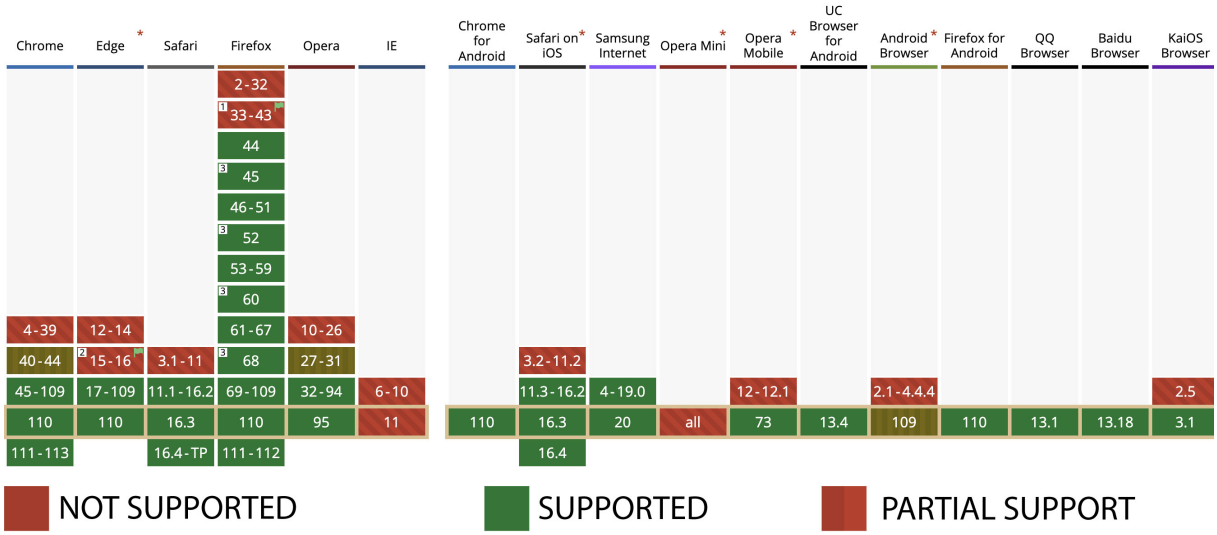

Figure 3.4: Service worker compatibility [1]

Modern browsers, including Google Chrome, Mozilla Firefox, Apple Safari, and Microsoft Edge, support the key features of PWAs, such as service worker, push notifications, and add-to-home screen functionality (Chromium-based) [46, 78, 1].

However, not all browsers will support every feature of the PWA. For example, the old internet explorer browser does not support service workers. To provide optimal user experience, the developer has to implement a fallback choice of the PWA. A fallback option can show a dialog box telling the user to change their browser to use the PWA to its fullest potential. Because of this, it is best to test the PWA on as many browsers as possible and set up a fallback option.

3.20 Workbox

Workbox is a collection of javascript libraries and tools that makes it easy to develop and deploy Progressive Web Apps (PWAs). Workbox offers a simple and extensible API for registering service workers, caching assets and data, and managing background synchronization and push notifications [81]. All the tools that workbox offers make it possible that developers can concentrate more on application logic and user interface.

It is also the most used library for service workers. It is used by 32% of mobile sites and in many build tools and CLIs, including the Angular CLI, Create-React-App, and Vue CLI [81, 82]. There are also plugins to most other libraries and frameworks, such as Next.js.

3.21 Storage Solutions for Progressive Web Apps: Choosing the Right Option

Progressive Web Apps (PWAs) offer a range of storage options for saving and maintaining data within the application [11, 10]. These storage options include:

- IndexedDB: A NoSQL database built into the browser provides a way to store and retrieve structured data on the application's client side [13].
- Web Storage API: A simple key-value store that can store small amounts of data, such as session data, on the application's client side [15].
- Cache API: A low-level API that allows developers to cache assets and data, such as HTML, CSS, JavaScript, and images, for offline access [14].
- Cloud storage: A cloud-based solution, such as Amazon S3 or Google Cloud Storage, can store and retrieve large amounts of data from the cloud [52].

Choosing the right solution will depend on the specific application requirements and the stored data type. For example, a PWA that requires offline access has to use cache API, while an Application that needs more structured data to be accessible offline has to use IndexedDB. Cache API is best suited to save the static files of the application shell, IndexedDb is better for User Content, Web Storage API is better for session tokens, and Cloud storage can be used to synchronize data across multiple devices, which allows users to seamlessly switch between devices without losing their data or progress.

3.22 Business Impact

Every business looks for cheaper development solutions that give the best user experience to make more profits. Here comes PWA, designed to create a fast, reliable, installable, and engaging standard website. A better user experience will help businesses increase customer loyalty and conversation rates and attract more returning visitors.

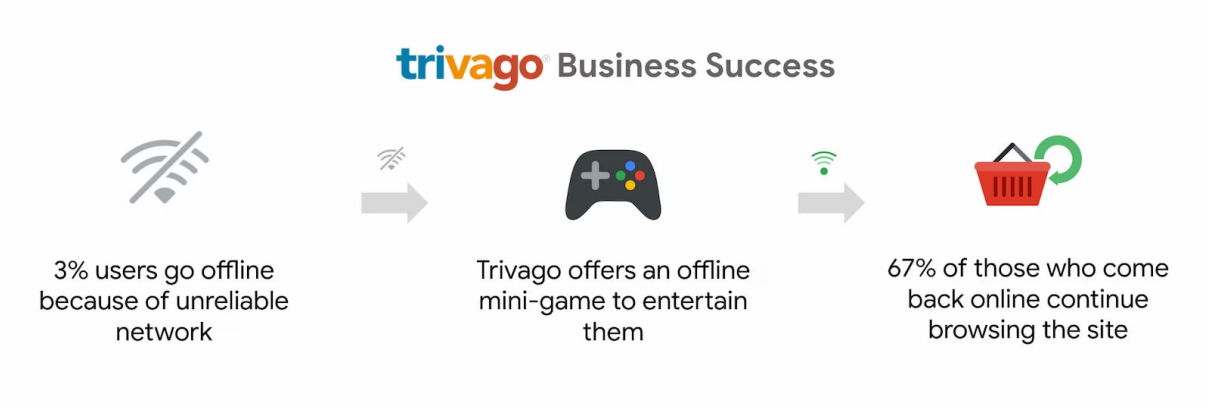

Figure 3.5: Trivago [70] saw a 67% increase in users who came back online to continue browsing.

The Trivago case study [60, 70] highlights the importance of offline browsing. For users who went offline, 67% resumed browsing the site once they were back online, demonstrating the value of a seamless offline experience.

Figure 3.6: Installed users had a 2.5 times higher conversion rate [70]

The Weekendesk case study [76, 70] presents an approach to promoting PWA installation. By encouraging installation on the second page visited, they significantly increased the likelihood of a user adding the PWA to their home screen. Consequently, users who launched the PWA from their home screen were over twice as likely to book a stay through the PWA!

In addition to these benefits, PWAs can be less expensive to develop than native apps because they don't need as much time and money to build. Moreover, since PWAs work on any device with a modern browser, companies can reach a wider audience without developing and maintaining separate apps for each platform.

Chapter 4

WebRTC

4.1 Key Features and Benefits

WebRTC (Web Real-Time Communication) provides an open-source framework for creating browser-based applications that support real-time audio, video, and data communication without plugins or additional software [71].

Key features and benefits of WebRTC:

- Real-time communication: WebRTC enables real-time audio, video, and data communication between browsers, allowing developers to create real-time communication applications [38].
- Browser-based: WebRTC is browser-based, meaning developers can build applications accessible to many users without requiring additional software or plugins [75].
- Open-source: WebRTC's open-source nature provides developers with a flexible framework that can be modified to create real-time communication apps [72].
- Data channels: WebRTC includes data channels, enabling users to send data in real time [77].
- Peer-to-peer networking: WebRTC utilizes peer-to-peer networking, allowing direct communication between browsers without needing a server [23].
- Improved user experience: WebRTC enables seamless real-time communication, even when the connection is slow or unreliable.
- Wide browser support: WebRTC is widely supported by modern browsers, including Google Chrome, Mozilla Firefox, Apple Safari, and Microsoft Edge (Chromium-based) [7].

4.2 Data Channels

Data channels in WebRTC are similar to WebSockets, providing a bidirectional data flow. They offer a mechanism to transfer data, such as text, files, or other information, across browsers in real time without intermediate servers [77, 62].

With the help of a signal channel, peers can share information and set up a connection between themselves. After establishing the connection, peers can exchange video, audio tracks, or data through the data channel. One of the advantages of data channels is that they provide low-latency data transport, enabling browsers to communicate quickly and efficiently. This fast communication is achieved using peer-to-peer networking [23].

Peer-to-peer networking minimizes the delay in standard server-based communication by sending data directly from one peer to another. WebRTC also offers end-to-end encryption for data channels, ensuring the data is secure when transmitted [2].

4.3 Network Address Translation

Network Address Translation (NAT) is a technology that assigns a public IP address to a device. A router typically has a public IP address, while each device connected to the router has a private IP address. NAT translates requests from the device's private IP to the router's public IP with a unique port [27]. While NAT firewalls protect networks from unwanted incoming data, they can pose challenges for real-time communication technologies like WebRTC.

NAT may prevent browsers from establishing direct connections with each other in WebRTC because each browser is behind its own NAT firewall. Achieving a peer-to-peer connection may require intermediary servers, such as STUN (Session Traversal Utilities for NAT) and TURN (Traversal Using Relays around NAT), to facilitate the exchange of network information and establish a real-time communication channel.

4.4 ICE

Interactive Connectivity Establishment (ICE) is used in WebRTC connection to make a peer-to-peer connection possible. ICE has all the necessary information, such as an IP address and port number [27]. This information will be retrieved from a STUN or TURN server to define the best network paths and a stable way to establish the connection.

ICE also responds to changes in the network, like when an IP address changes or the network goes down, by re-evaluating the best network path on the fly.

4.5 STUN

A Session Traversal Utilities for NAT (STUN) server is a network service that determines a browser's public IP address and the port number behind a Network Address Translation (NAT) firewall [27, 74]. To establish a peer-to-peer connection with WebRTC, each browser behind a NAT must share its public IP address and port number. However, the browser may not know its public IP address, and that's where the STUN server comes into play.

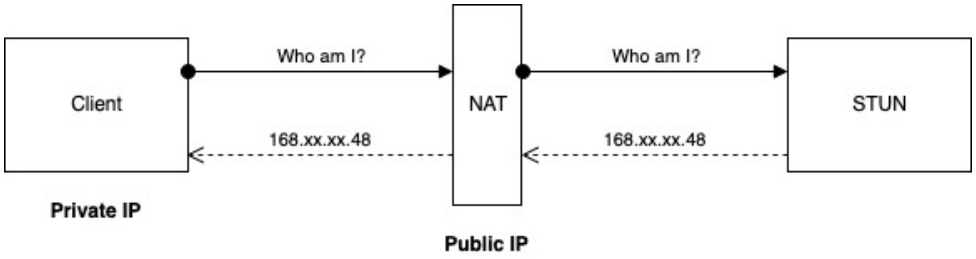

Figure 4.1: Discovering Public IP and Port Using STUN Server [74]

The browser [55] sends a "Who am I" request to a STUN server, which then responds with the browser's public IP address. This information can be used to facilitate peer-to-peer connections for WebRTC applications.

4.6 Security

WebRTC is secure but it is not immune to security risks. WebRTC security is a complex topic that requires preserving the privacy and secrecy of browser-to-browser communication and assuring the validity and integrity of sent data [3].

Some of the essential WebRTC security features include:

- Encryption: WebRTC uses encryption to protect the privacy of communication between browsers. By encrypting the transmitted data, WebRTC helps prevent eavesdropping and tampering with communication.
- Authentication: WebRTC uses authentication to ensure that the browsers communicating with each other are who they say they are. Authentication helps prevent man-in-the-middle attacks, where a third party intercepts and manipulates the communication between two browsers.
- Access Control: WebRTC uses access control to ensure only authorized browsers can access the real-time communication channel. Access control helps to prevent unauthorized access to the communication channel and protects against attacks such as eavesdropping and tampering.

WebRTC is designed to be secure, but security will always be a complex subject, and it is essential to be aware of the security dangers and take the best action for protection.

4.7 TURN

Traversal Using Relays around NAT (TURN) is a protocol that facilitates the transmission of real-time data between two clients located behind different NAT firewalls [74]. TURN is commonly used in WebRTC applications when direct peer-to-peer communication is impossible due to restrictive NAT configurations or network policies. TURN bypasses the Symmetric NAT restriction by opening a connection with a TURN server and relaying all information through that server [4].

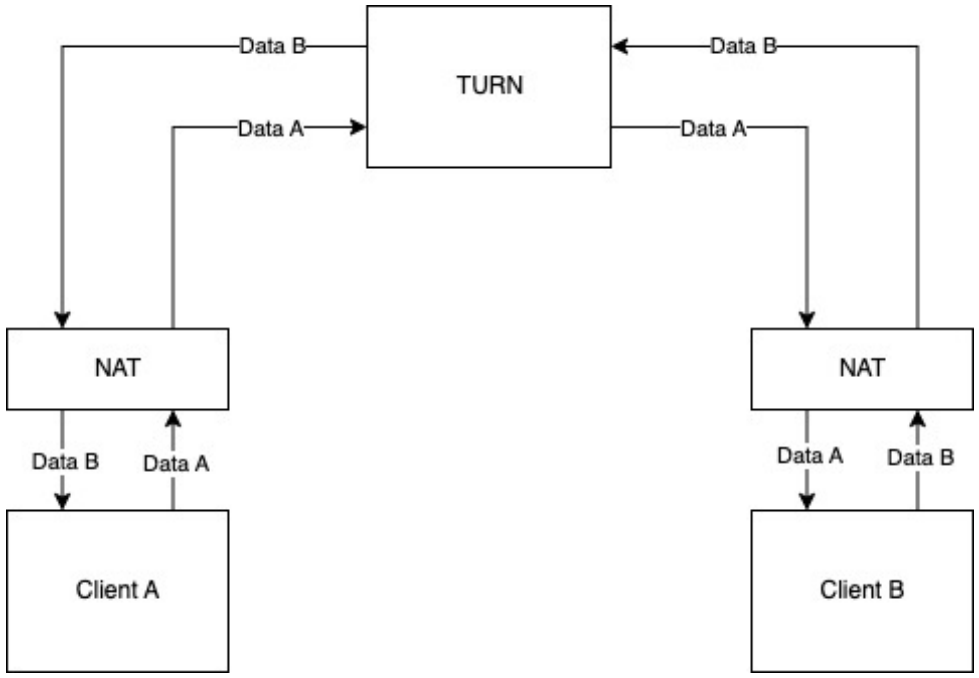

Figure 4.2: Traversal Using Relays around NAT (TURN) and Its Role in WebRTC Applications [74]

When the connection is not possible from the STUN servers, a fallback solution is applied through the TURN servers. TURN servers work like signal servers that forward all packets from client A to client B. In this situation, we will not have a direct peer-to-peer connection. The drawback of using a Turn server is that they need to handle the whole communication and requires appropriate performance and bandwidth.

4.8 Limitations and Compatibility Issues

WebRTC is a powerful technology that lets browsers communicate to each other in real time. However, it has some problems with compatibility. Some of the most significant limitations and compatibility issues of WebRTC include the following:

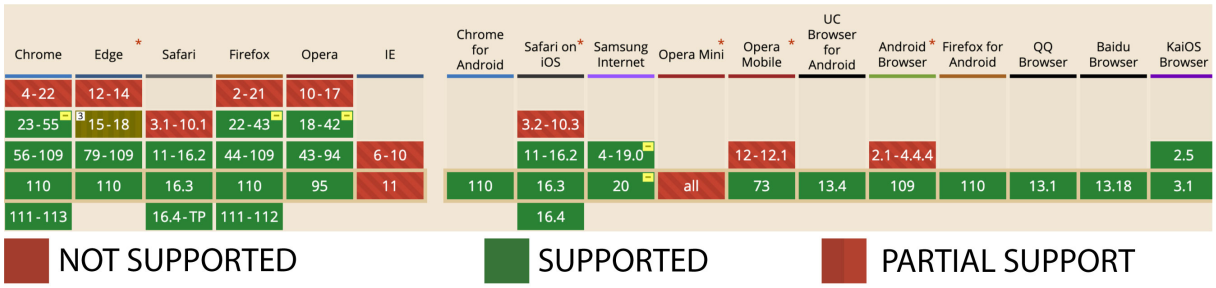

Figure 4.3: WebRTC browser compatibility [1]

Compatibility with browsers: WebRTC has yet to be fully supported by all browsers, and some may have different levels of compatibility and implementation. This can result in compatibility issues for users using other browsers, limiting the overall reach of WebRTC applications [1].

Latency: High-latency connections in WebRTC manifest symptoms such as lagging audio/video playback, intermittent video freezing, and occasional video frame-rate drops, although audio playback typically remains smooth. Additionally, logs may display a high number of picture loss indications, which receivers utilize to request a full frame refresh [63].

Network Conditions: In particular, WebRTC's reliance on peer-to-peer networking makes it vulnerable to the performance of individual users' networks.

Security: WebRTC is designed to be secure but not immune to security threats. In particular, the peer-to-peer nature of WebRTC can make it vulnerable to man-in-the-middle attacks, where a third party intercepts and manipulates the communication between two browsers [2].

Implementing WebRTC without any library is very challenging because the browser will also support WebRTC API, but they have different behaviors. To overcome these differences, a library like adapter.js [73] is recommended. All these limitations and compatibility problems must be resolved to reach the full potential of WebRTC.

Chapter 5

Requirements and app design

5.1 A brief overview of the Crew Active time recording system and its existing features

Crew Active time recording system is a time recording web-based [64] solution designed to assist businesses of all sizes and specializations in managing a wide range of operational aspects. These include employee time tracking, project and task management, communication, location tracking, online booking, and employee absences. Developed to address the diverse needs of various industries, Crew Active is a comprehensive solution for organizations seeking time recording and extensive management and coordination tools.

Crew Active's core functionality is its efficient timekeeping system, which allows employees to log hours, view schedules, plan vacations, and monitor their locations [65]. The system's user-friendly interface enables users to quickly and effectively use its features.

Regarding project management, Crew Active offers features such as project scheduling, task pricing, change order management, and time tracking. Tasks can be assigned to employees and viewed in different time frames and formats within the system, including month, day, and week views. The system also displays specific data for each job, ensuring that workers can always access the most up-to-date information.

Crew Active's real-time chat feature fosters strong teamwork. At the same time, its timeline view presents the progress of tasks and projects in a visually appealing and easy-to-understand format for both employees and supervisors.

One unique feature of Crew Active is the ability for administrators to send task requests to employees, who can then accept or decline the assignment. This empowers employees to manage their time and ensures they take on projects that suit their skills and availability.

The online booking feature in Crew Active allows companies to schedule appointments with clients quickly. Its user-friendly interface makes it simple for customers to book

appointments or request services, streamlining the booking process and reducing administrative costs.

Crew Active offers both web and mobile applications for iOS and Android. While administrators primarily use the desktop version of the software, which has more features than the mobile version, it is essential to have offline capabilities to continue working even without internet access. Introducing PWA features in Crew Active will enable administrators to use the system regardless of their internet connection, allowing them to accomplish critical tasks even when offline and ultimately enhancing the system's overall value to the organization.

5.2 Requirements

The extended Crew Active time recording system should provide the following features:

- Offline Mode: The system should be designed as a Progressive Web App, allowing it to work offline and allowing administrators to continue working even when they do not have an internet connection. This means that administrators should be able to manage task changes when they are offline. This will improve efficiency and reduce downtime, ensuring that users can continue working even when they do not have an internet connection.
- Video Streaming: The system should provide video streaming capabilities using WebRTC technology that allow users to stream video in real-time to other users, improving collaboration and communication.
- User-Friendly Interface: The system should have a user-friendly interface that is easy to navigate and use, allowing users to quickly and easily access the information and tools they need to manage their time and projects effectively.
- Mobile Compatibility: The system should be mobile-compatible, allowing users with the role of workers to access it from their mobile devices, improving accessibility and flexibility.

5.3 Technology Stack

The technology stack used for the Crew Active time recording system comprises several key components and frameworks that work together seamlessly. The MERN stack (MongoDB, Express, React, and Node.js) serves as the core part of the stack. This full-stack JavaScript web application enables rapid development, easier maintenance, and better performance, making it an ideal choice for the Crew Active system.

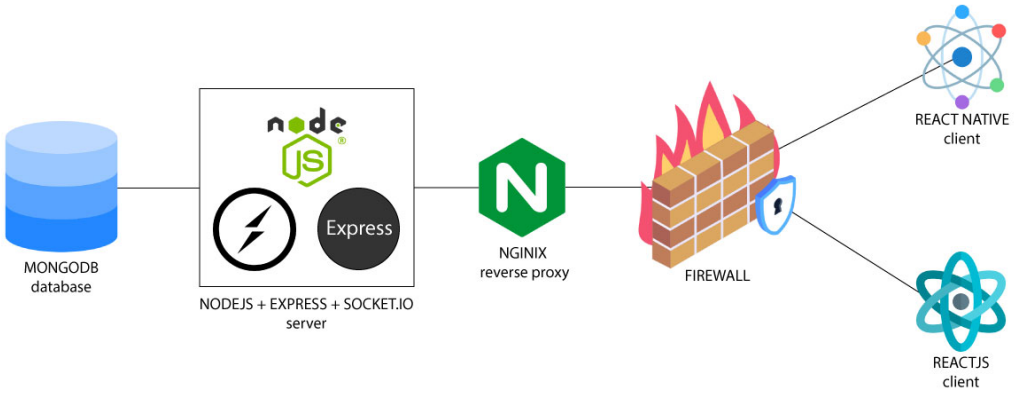

Figure 5.1: Crew Active Technology Stack: MERN, React Native and React

MongoDB, a flexible and scalable document-based database management system, can handle large amounts of data efficiently. Express, the web application framework, provides Node.js developers a robust and flexible way to build web apps. React creates the system's front-end user interface, offering users a fast, responsive interface with reusable components. Node.js is the back-end server, providing a scalable platform for constructing server-side applications. Nginx is a reverse proxy to enhance the system's security and speed.

The components of the MERN stack interact with each other to create a seamless experience. For example, React communicates with Node.js to request data from MongoDB, while Nginx and Express work together to handle web traffic. Socket.IO ensures real-time updates for all connected devices, making it easy for users to collaborate and communicate.

In addition to the MERN stack, the system employs several other frameworks and tools. Material UI is used for designing the user interface, providing customizable UI components that follow Google's Material Design guidelines. Framer Motion is utilized for animation and motion design, boasting an extensive library that simplifies the creation of complex animations and transitions. Tailwind CSS, a utility-first CSS framework, is employed for styling and design, making it easy to style UI components.

Overall, the technology stack for the extended Crew Active system offers a robust and efficient platform for building a comprehensive time-recording solution that addresses the functional requirements outlined in the thesis. This stack enables the developing

of a highly responsive and feature-rich application, meeting the Crew Active system's and its users' needs.

The client side of Crew Active includes a React Native application for iOS and Android devices, as well as a web app developed using React.js, further emphasizing the benefits of using a full-stack JavaScript web application.

5.4 Database Design

The Crew Active Time Recording System's MongoDB database encompasses many entities; nevertheless, this overview showcases the most vital ones to clarify its design.

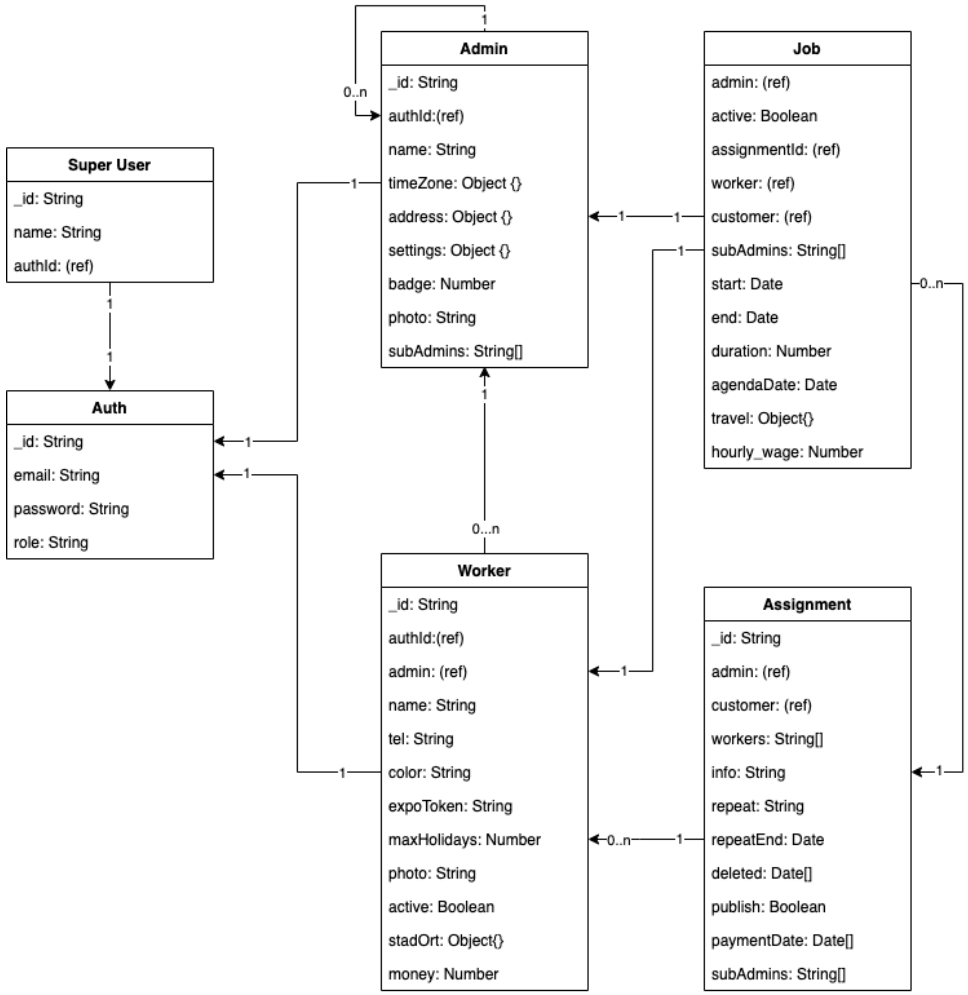

Figure 5.2: A Visual Guide to Database Relationships

The "auth entity" is explicitly designed for authentication, with the ability to incorporate additional roles as the system expands quickly. The "super user" possesses extensive privileges, allowing them to access server and application logs, manage administrators and workers, and support all other users. The system's architecture streamlines task delegation and enhances security and stability.

The "assignment entity" is a critical component of the Crew Active time recording system, representing an administrator's responsibility to create and assign tasks to multiple employees. The assignment object houses essential task information, including work descriptions and due dates. The assignment entity enables administrators to manage assignments, monitor progress, and furnish workers with the necessary details to complete their tasks. Administrators can also share assignments with other administrators, allowing access to the corresponding workers' jobs but limiting access exclusively to the shared assignments.

The "job entity" forms a fundamental element of the Crew Active time recording system, encompassing each worker's location, start and finish dates, supplementary information, and distance traveled. This entity allows administrators to track employees' progress, supervise their locations, and manage their jobs. By accessing this information, administrators can make informed decisions, ensuring employees complete their tasks efficiently. The job entity provides a comprehensive overview of each employee's progress, enabling improved resource planning and management.

The project requirements also include implementing video streaming capabilities using WebRTC. This feature enables the super user to initiate video calls with all users in the system for support and assistance purposes, addressing any issues users may encounter. Additionally, administrators can establish video streams with their workers to facilitate better communication and collaboration.

Within the Crew Active system, the "admin," "worker," and "super user" entities are distinct, each with their unique powers and responsibilities. The login form authenticates all three entities, including the super user. After successful authentication, the program's appearance and navigation are customized based on the user's role, granting access to appropriate features and capabilities in alignment with their rights.

5.5 Calendar View Design with Offline Functionality and Network Status Integration

The calendar view is a crucial feature for administrators, allowing them to assign tasks to specific employees. When a task is created, all assigned employees receive push notifications and update their calendars accordingly.

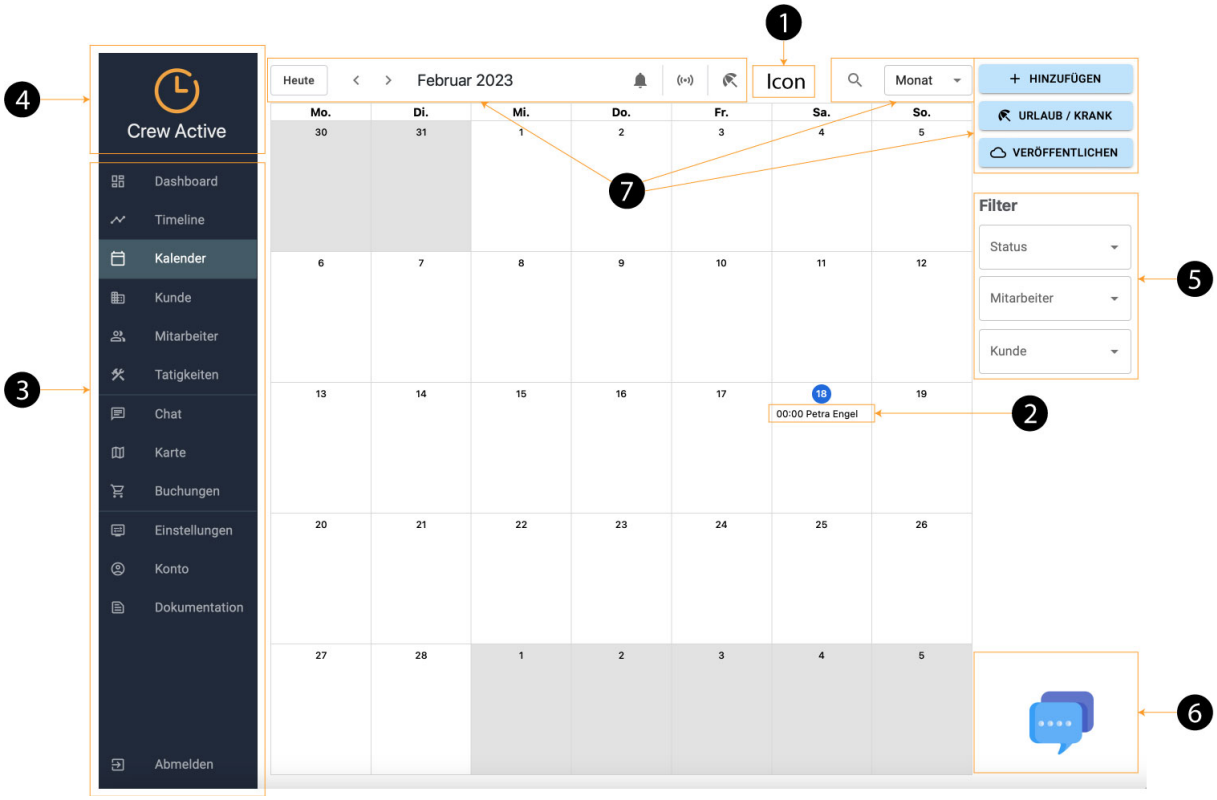

Figure 5.3: Calendar View with Network Status Indicator and Offline Operations Drawer

An essential design requirement for this view shown in Figure 5.3 is incorporating a button with an icon at (1) that indicates the network's online or offline status. The icon will change to a distinct symbol when offline, alerting the user of the lost network connection. Users can continue working offline during this time, and a counter will display the number of completed operations. Clicking on the icon will reveal a list of offline actions in a drawer on the right side. Figure 5.3 showcases a mockup that includes the network status button with an icon at (1), which changes depending on the user's online status. The counter tracks user actions when offline, and clicking the icon opens a right-hand drawer containing a list of offline operations. A sample task created by an administrator is displayed at (2), while the navigation drawer with multiple navigation tabs can be found at (3). The Crew Active logo at (4) turns gray when the network is offline and orange when online. Administrators can apply filters at (5) to view the calendar in various ways, such as selecting one or more employees. The chat button (6) initiates a direct support conversation. Although several actions are available at (7), they are not within the scope of this thesis.

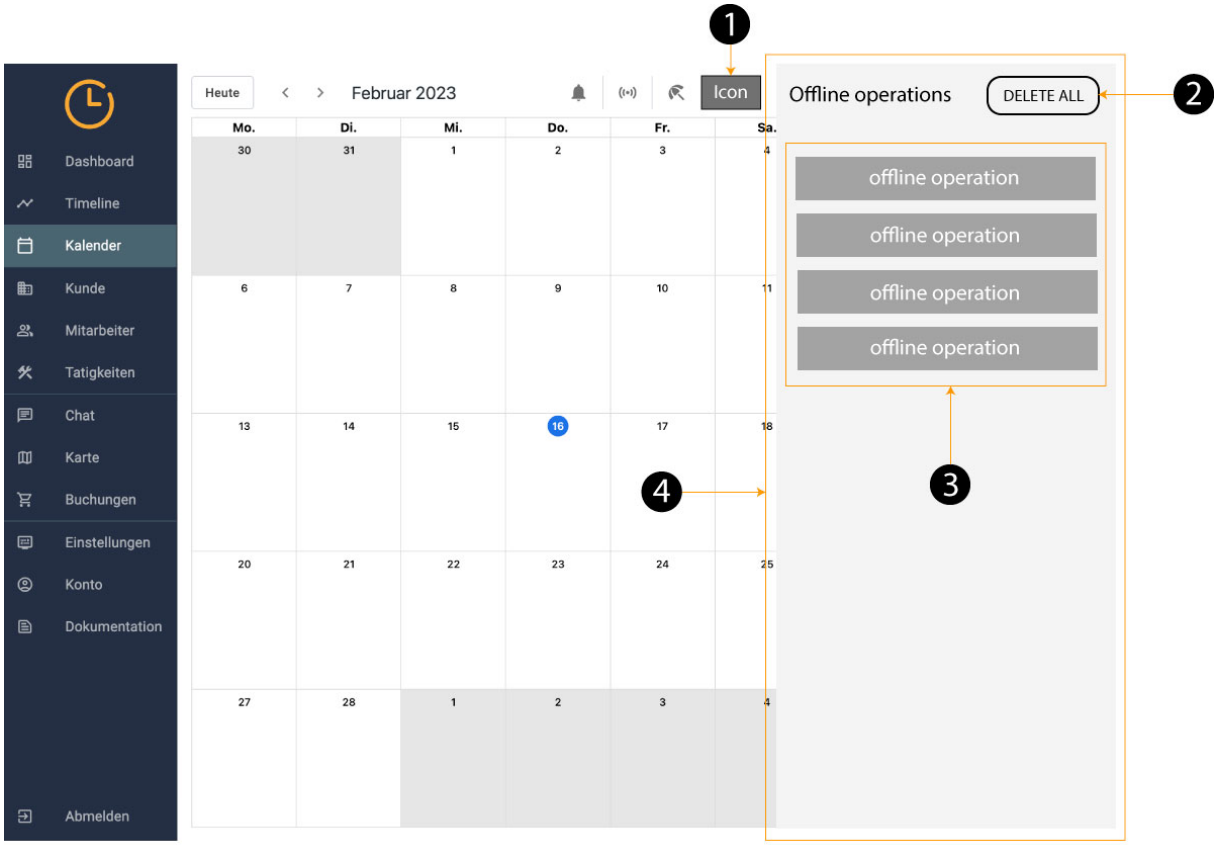

Figure 5.4: Calendar view icon clicked

In Figure 5.4, the clicking of the icon at (1) will open a drawer at (4), which will display a list (3) of all the offline operations that the user has performed while disconnected from the network. At (2), there will be a button with the action to delete all the offline operations.

The offline operations are all the actions the user has taken while offline. For example, if an administrator changes different tasks on the calendar, these changes will happen locally, and an offline operation will appear in the list. The first time the user connects to the internet, all offline operations will be synchronized with the server and removed after synchronization. This way, the user will have better control over what has been synchronized with the server.

5.6 Incoming Call Handling and User Interface

In the Crew Active application context, one of the critical functionalities is facilitating seamless communication between users through video calls. This feature is essential for effective collaboration and coordination among team members. To ensure a user-friendly experience, the application provides an intuitive interface for handling incoming calls.

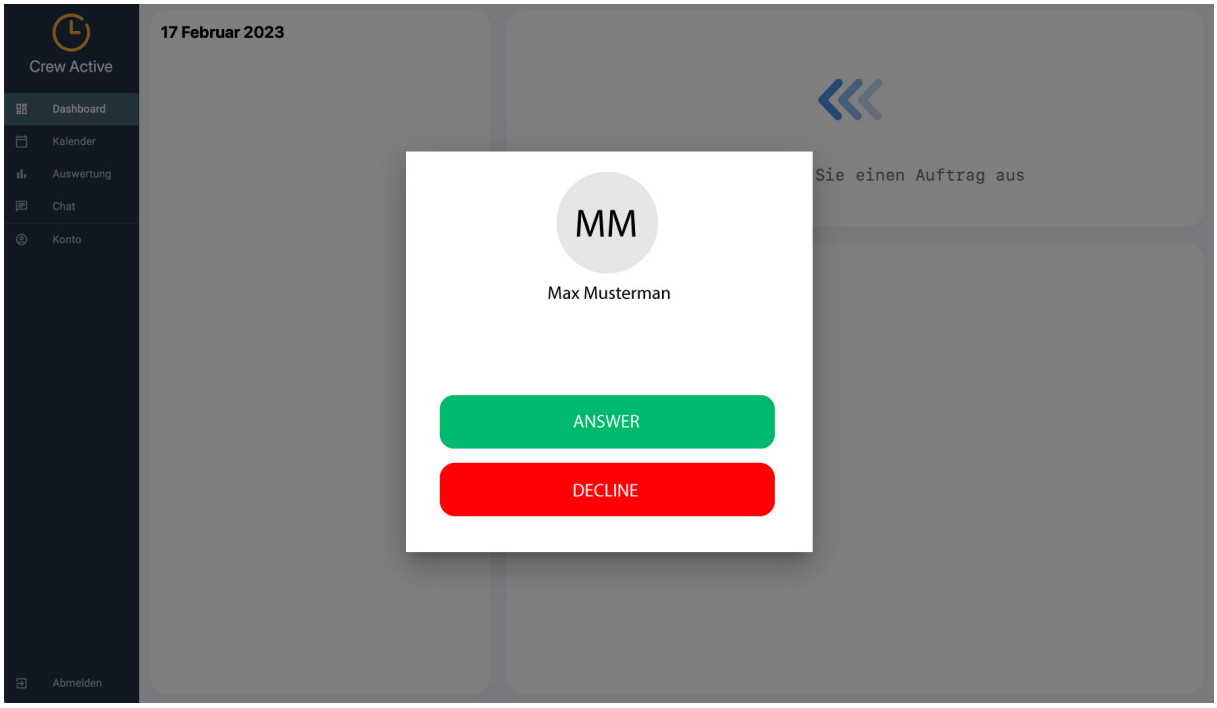

Figure 5.5: Incoming Call Interface in Crew Active Application

Figure 5.5 demonstrates the situation when a user receives a call from another user within the application. Upon receiving the call, a dialog box displays the caller's image (if available) and name. The recipient is presented with two options: accept or decline the call. If the caller cancels the call before the recipient decides, the recipient will be informed that the call was turned down. On the other hand, if the recipient accepts the call, a full-screen conference room will open, granting the user access to various call options for an engaging and productive conversation.

5.7 Video Conference User Experience and Interface Design

Video conferencing is an essential tool that facilitates seamless interaction between team members, partners, and clients, regardless of their physical locations. The Crew Active system has to incorporate video conferencing features to ensure smooth user collaboration and coordination.

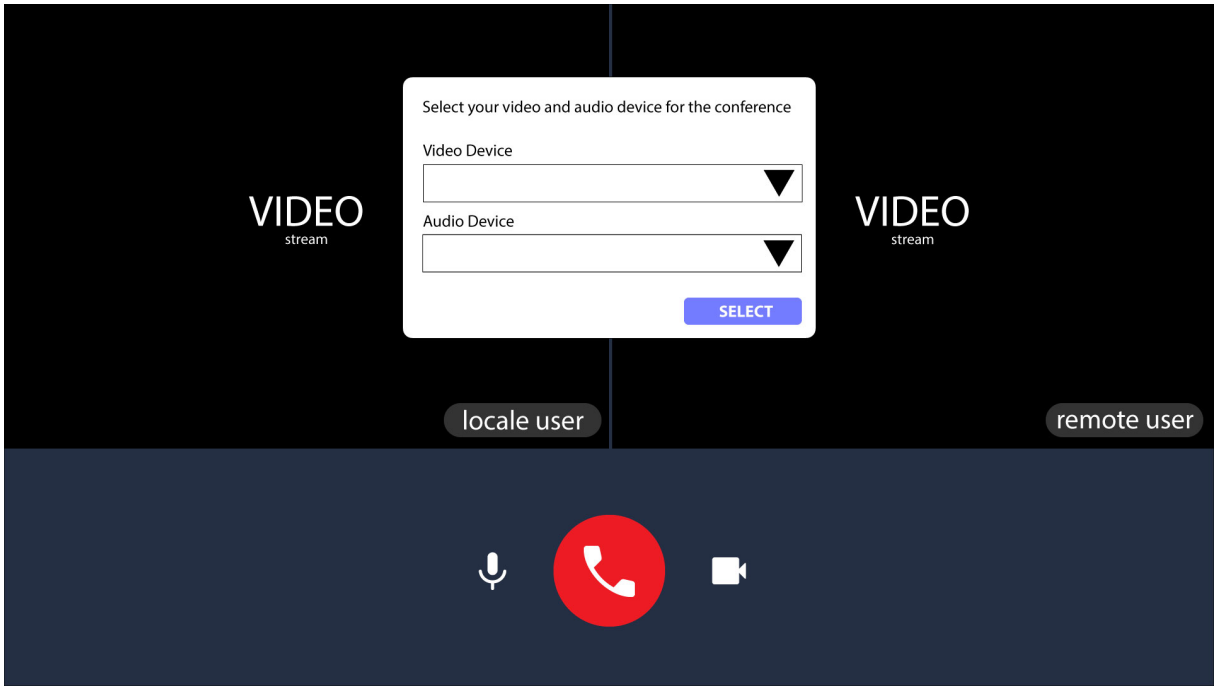

Figure 5.6: Video conference mockup

After accepting a call, the user will witness a view depicted in Figure 5.6. The user shall receive a dialog box initially to select the desired video and audio output for the conference. Once the video and audio outputs are selected, the dialog box will disappear, and the local video stream will be displayed on the left-hand side of the window. In contrast, the remote user's video stream will be displayed on the right side. Note that each user's name will be displayed at the bottom right corner of each video container. In the middle of the bottom of this window, the user can turn off or on the audio or video stream and may decline the call at any given moment.

Chapter 6

Implementation and Results

6.1 Conversion to a Progressive Web App

Implementing a Progressive Web App (PWA) in the Crew Active system, the following steps have been followed:

6.1.1 Register a Service Worker

The service worker caches the application's static assets and serves them to the user, even when the network is unavailable [45]. It first attempts to display cached data and then fetches updated data from the server in the background, ensuring that users can access the most current information, even offline. The offline-first strategy enables continued application use without a network connection.

The implementation of the PWA starts with registering a service worker by creating and registering a JavaScript file in the main HTML file. The client side of Crew Active uses React and is initialized as a React Progressive Web App (PWA) template, which incorporates Workbox internally. This integration [17] simplifies the process of writing and registering the service worker logic. The React PWA template generates all required files, including the service worker file, leaving only the task of changing the service worker registration from the unregister to the register method in the main index.js file.

To ensure the best user experience, it is crucial to address the issue of updating the Crew Active system when changes are made and a new version is deployed. Users are informed of new versions when available by implementing an interval for checking the latest version of Crew Active every five minutes. A dialog box prompts the user to manually close all open tabs of Crew Active and reopen the application to load the latest version. This approach allows users to choose when to update without interrupting their work. To minimize user workflow disruption, the refresh and closing of tabs are not performed programmatically.

6.1.2 Cache Files with Service Worker

The precache feature in a React PWA template caches all the application's static assets in the service worker. The precache feature is typically implemented using a tool such as Workbox, which is included in the React PWA template. When the service worker is registered and activated, it will cache all the files specified in the precache configuration. This way, when the user visits the application, the service worker will serve the cached assets first, improving the loading speed and providing an offline-first experience. The precache feature is handy for assets that do not often change, such as stylesheets, images, and JavaScript files. The precache feature in the React PWA template will cache all public folders in react tree. By caching these assets, the application can load faster and provide a more reliable experience, even when the user is offline.

6.1.3 Manifest File

This file contains all the metadata about the app, such as its name, icons, start URL, display mode, theme color, and background color. The browser uses this information to show the app on the user's home screen and customize its appearance.

```
{
  "short_name": "Crew Active",
  "name": "Crew Active",
  "icons": [...],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#233044",
  "background_color": "#233044",
  "description": "Zeiterfassungssystem für Mitarbeiter",
  "orientation": "any",
  "related_applications": [
    {
      "platform": "Google Play",
      "url": "https://play.google.com/store/apps/details?id=com.workers.imanage",
      "id": "com.workers.imanage"
    },
    {
      "platform": "App Store",
      "url": "https://apps.apple.com/de/app/crew-active/id1560567129",
      "id": "1560567129"
    }
  ],
  "prefer_related_applications": true
}
```

Figure 6.1: The manifest file

The manifest.json file is linked in the index.html file using the link tag.

6.1.4 Offline Functionality

So far, Crew Active can operate offline, is discoverable, and can be installed. Yet, while offline, it only loads static files. Implementing a solution that allows users to access the most recent data (user-specific data) loaded from the internet is essential.

Before exploring this, it is essential to examine the client-side state management process to determine the most effective way to persist all relevant data.

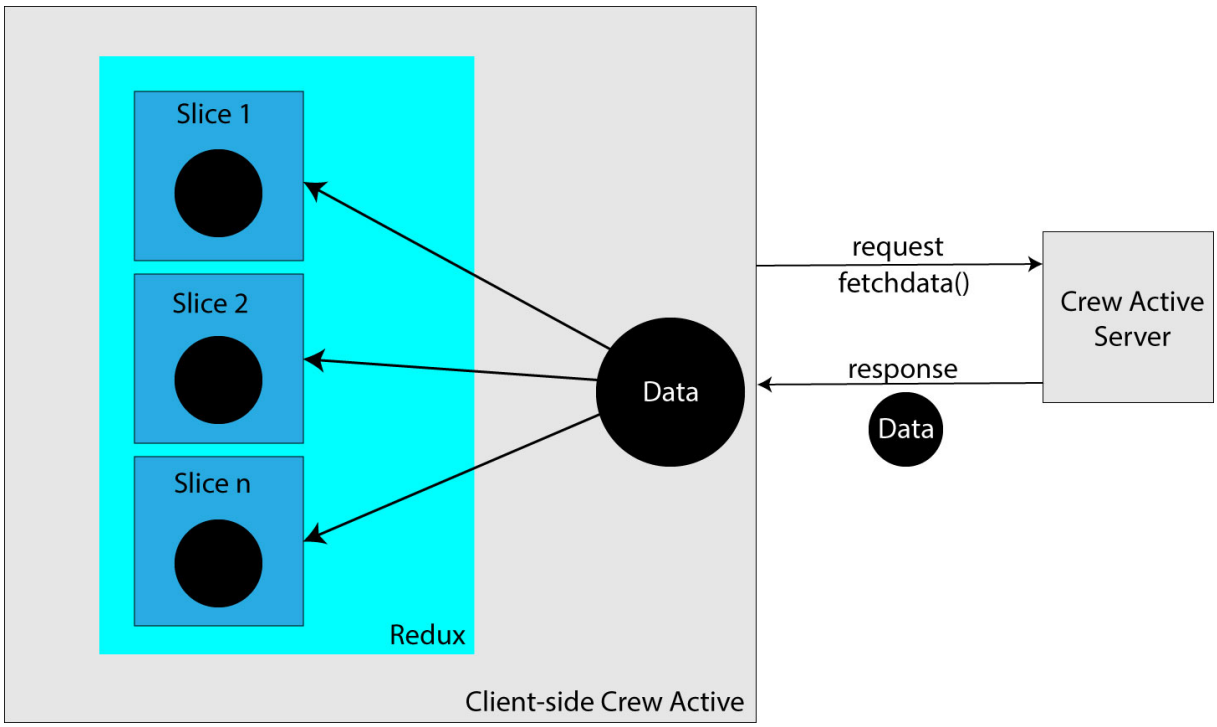

Figure 6.2: Client-side State Management with Redux Slices

In the situation shown in Figure 6.2, the user is already logged in, and a POST request is made to retrieve the corresponding data for the user from the server. The server then sends back all the data required to the user. This data is saved as states using Redux, a popular state management library for React applications, and is split into different slices. This approach preserves the data only when the application tab is open and allows for various other operations. All the necessary data is available now, but it only exists on the client side while the tab is open. The next step in the implementation process involves persisting this data on the client side so that the user can access it even when offline but already logged in.

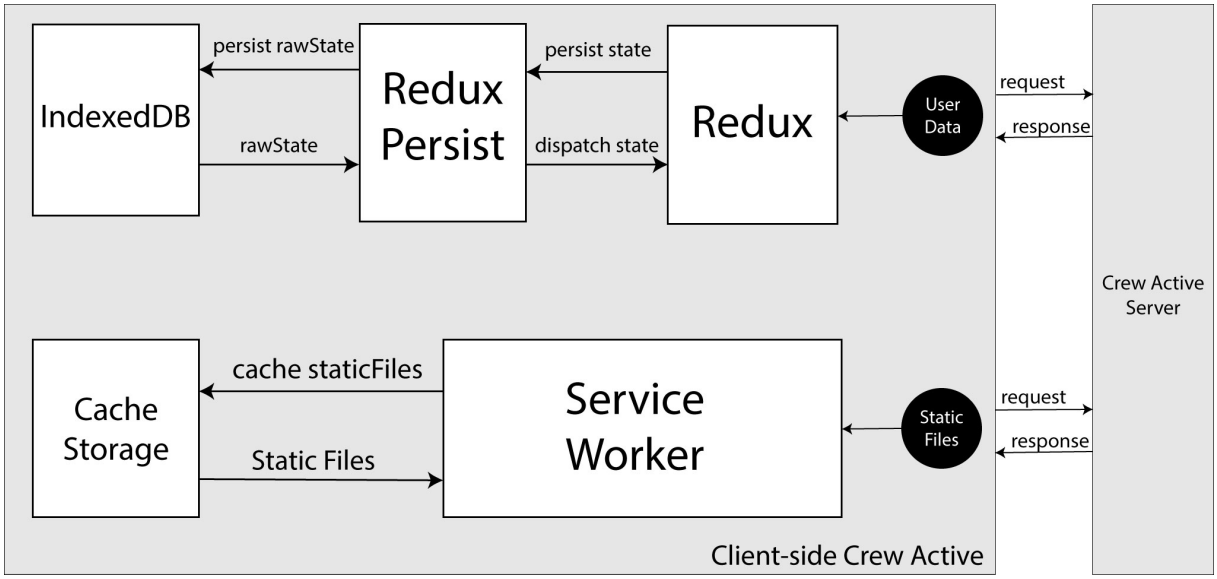

Figure 6.3: Efficient Data Management: Persisting User Data and Caching Static Files

Figure 6.3 illustrates the solution implemented in the extended Crew Active system to enable offline access to static files and the latest fetched user data. When the application is opened in the browser, the system checks if a service worker is registered. If not, it registers the service worker, which is crucial for enabling offline functionality (at this point, everything still needs to be cached). Next, the client side of Crew Active makes a GET request to the Crew Active server for all static files and a POST request for user-specific data. The service worker caches the static files in the browser's cache storage, while the user data from the POST request is saved in Redux state management. Redux Persist, a persisting library designed explicitly for Redux state management, is used to automatically persist changes in the Redux state in IndexedDB storage, ensuring that the data is accessible even offline. When the application is opened again while offline, the system retrieves the cached static files from cache storage and uses Redux-Persist to dispatch the persisted data into the Redux state. In the background, the client side of Crew Active makes a POST request to the Crew Active server to receive the latest data. If there are any changes from the old data, the client updates the data in Redux with the new one and re-renders the client. If there are no changes, everything remains the same, and no re-rendering occurs.

The library Redux Persist [50] is used to persist the Redux state in the client's browser storage, ensuring that the state remains available even after the user closes the browser or refreshes the page. Redux-Persist works by automatically serializing and deserializing the state to and from the browser storage, integrating with the Redux store. This approach makes it easy to retrieve the persisted state and restore the application's state when reloaded. Redux Persist supports multiple storage backends, such as local storage, session storage, and IndexedDB, providing flexibility for different scenarios. IndexedDB storage offers notable advantages, including superior performance and the capacity to manage larger volumes of data compared to alternative storage options.

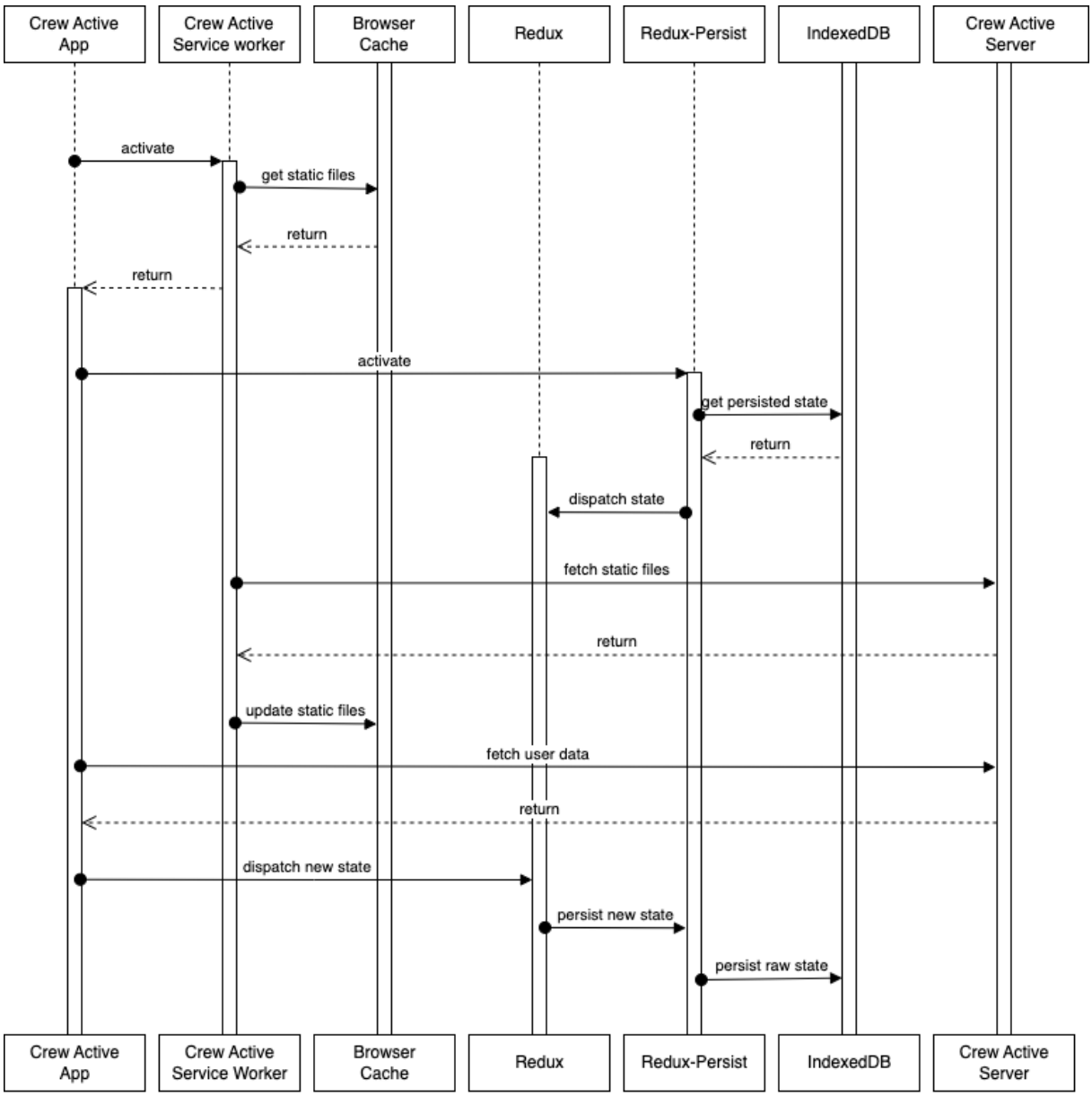

Figure 6.4: Offline Data Management and Synchronization Process

The sequence diagram in Figure 6.4 provides a detailed overview of the steps when users open the application in their browser. First, the service worker is activated, retrieving cached static files from the browser cache storage. The app then activates the Redux Persist library, which populates the Redux state by obtaining the persisted state from IndexedDB, transforming the raw state into a JavaScript object, and dispatching it to the Redux state library. This approach enables the app to launch quickly in the browser with all available content, eliminating the need to wait for a server response.

The service worker then checks for updates or changes to the static files on the server. If updates are found, the service worker displays a dialog informing the user that a new application version is available. The user can close all tabs to load the new version or continue working with the existing version. If the user opts to load the latest version, the cache is populated with it, and the old version is removed.

Simultaneously, the client-side application fetches the latest user data from the server, dispatching the new state to Redux. Redux Persist synchronizes the changes if updates are present, ensuring that the updated Redux state entirely persists in IndexedDB.

This solution enhances the client-side application's performance upon initial opening by displaying the content immediately and fetching new data from the server in the background to update the content. As a result, users enjoy a better experience when opening the application in their browser.

6.1.5 Offline User Activity Synchronization with the Crew Active Server

So far, the Crew Active application can be opened offline in the browser, and the entire application state persists. The next step involves implementing a solution that allows users to perform offline actions, which will be synchronized with the server when the user returns online.

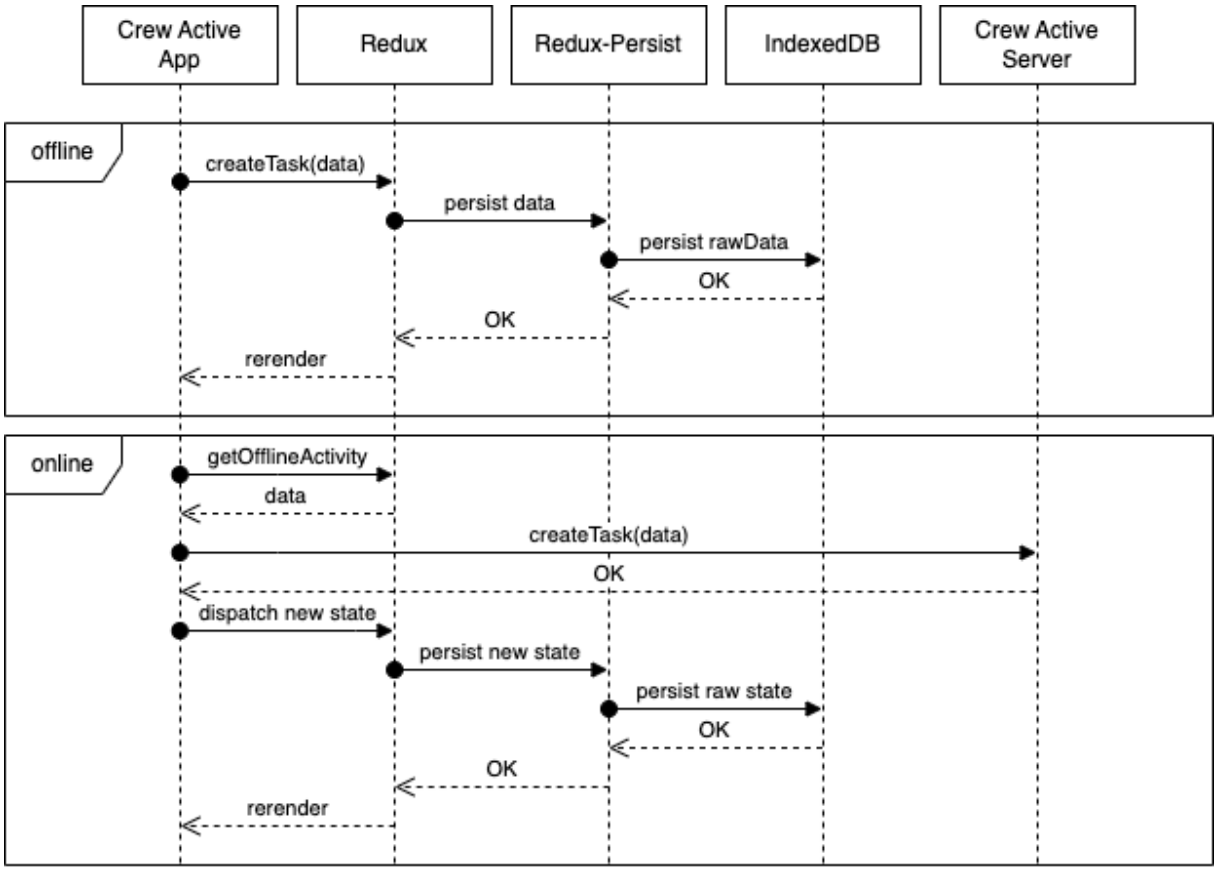

Figure 6.5: Creating a Task Offline and Syncing with Server

Figure 6.5 illustrates the solution for synchronizing offline user activity with the Crew Active server. When a user is offline, they can create a task, and all the required payload for the task creation is saved in the Redux state. Redux Persist automatically persists any changes in the Redux storage, ensuring all data persist locally for later use when the user is online.

Once the user is online, the Crew Active application in the browser checks if the user has any offline activity. If so, the data is retrieved from Redux, and the application requests the server to initiate the offline task.

After receiving the server response, a new state containing the data from the server is dispatched to Redux, and Redux Persist automatically persists in the updated state. This ensures that when the user opens the application in the browser next time, all content is pre-rendered without needing an initial request to the server.

Finally, the application re-renders all components where the data from the created task is displayed.

6.2 Video Streaming

In Crew Active, WebRTC (Web Real-Time Communication) enables real-time client communication. This technology allows for peer-to-peer communication between browsers, eliminating the need for a central server to facilitate communication.

The first step to implementing WebRTC in Crew Active is establishing a connection between the two clients. This is done using a signaling process, where the clients exchange information about their network and media capabilities.

6.2.1 Signal Channel

To facilitate real-time data exchange between users, a signal channel is implemented using Socket.io. Figure 6.6 illustrates a real-time signal channel developed with Socket.io, which enables users to send data to each other seamlessly.

Figure 6.6: Real-Time Data Exchange between Clients Using Socket.io Signal Channel

In this scenario, Client A sends data to Client B. Client A's Socket.io client emits a signal containing the data payload. The Socket.io server captures this signal and broadcasts the data to other connected clients, including Client B. Consequently, Client B instantly receives the data sent by Client A, allowing for real-time information exchange between the two users through the signal channel.

```
socket.on('webrtc-signal', async (data, callback) => {
  const { token, payload } = data;
  const { access, senderID, error } = await authRequest(token);
  if (!access) return callback({ error });
  WebRTCRouter.forwardRequest({ senderID, payload, socket, callback });
});
```

Figure 6.7: Signal Channel Implementation for Secure and Efficient Data Exchange Between Users

Figure 6.7 depicts the creation of a signal channel named "webrtc-signal" that allows users to send data to the server. The client sends data, including a token and payload key, to the server via WebSocket. The server verifies the user's existence in the database and their permission to send and receive data and retrieves the user's unique ID. If the user lacks access, the request is rejected. If granted access, the WebRTCRouter, a class responsible for handling signaling operations, forward the payload to the designated receiver user, whose ID is included in the payload sent by the sender. A callback function provides the sender with updates on the status of their request.

Upon logging in, each user receives data from the server and is assigned a unique identifier (ID). They then establish an open connection with the Socket.io server, enabling it to send data anytime to the user. As shown in Figure 6.7, users can send data to the server, which forwards it to the intended recipient. This establishes a connection allowing users to send data to one another through the signal channel.

Only the unique ID of the receiving user is needed to send data from User A to User B. The payload sent by one user to the signal channel is a JSON object containing the keys "token," "receiver," and "payload." The "token" key identifies the sender on the server, the "receiver" key includes the unique ID of the user receiving the data, and the "payload" key contains the data being sent. The server validates the signaling message by verifying the sender's user ID and checking if the user account is not blocked. If both conditions are satisfied, the server forwards the payload to the intended recipient.

After connecting to the signal channel, users can exchange information to enable audio or video streaming. Crew Active accomplishes this using WebRTC APIs, such as RTCPeerConnection and MediaStream. The RTCPeerConnection API sets up client communication, while the MediaStream API accesses the client device's camera and microphone, allowing clients to share audio and video streams.

6.2.2 Acquiring User Media Streams for Video Conferencing with MediaStream API

Crew Active app on the client side utilizes the MediaStream API to acquire user media streams, facilitating user participation in video conferences through audio and video devices, such as webcams and microphones.

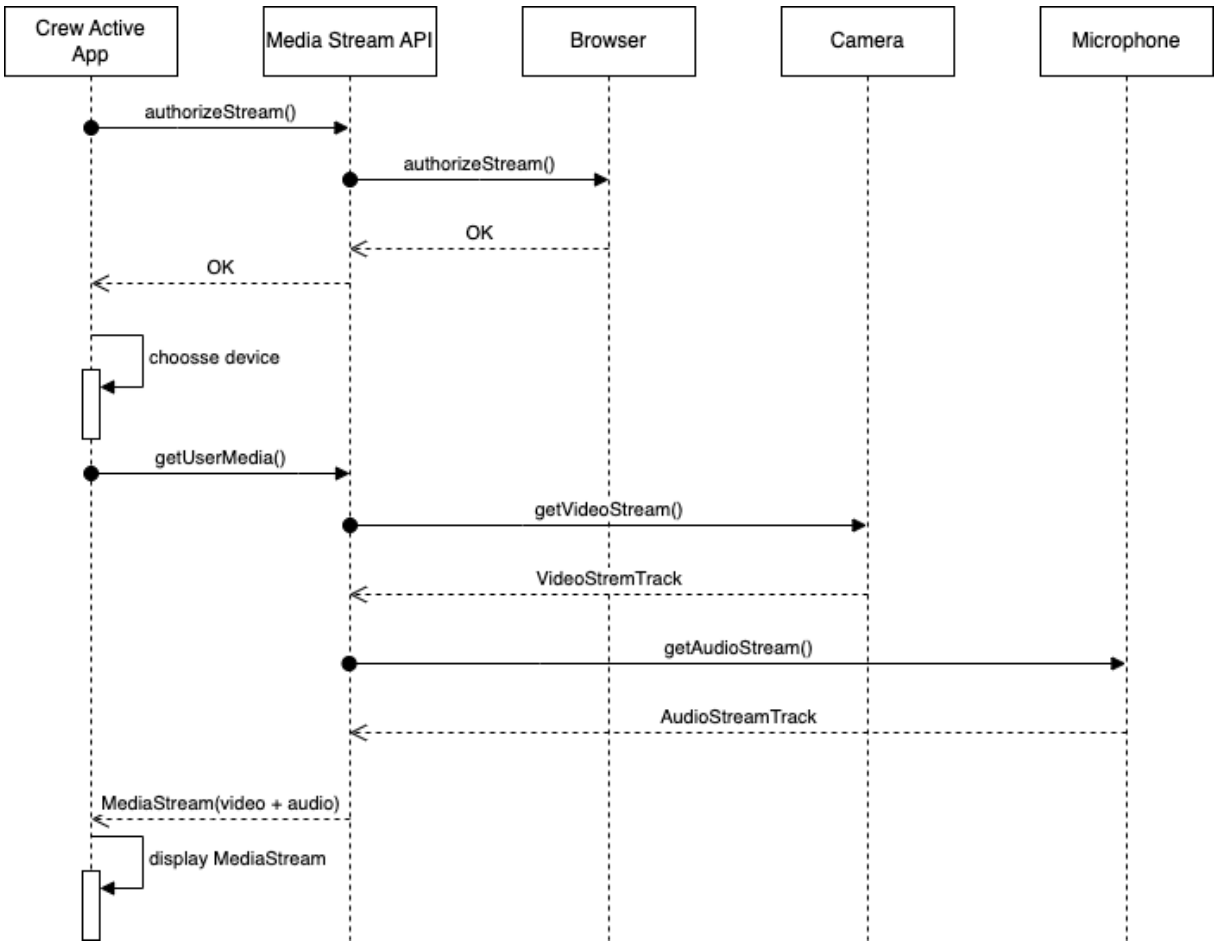

Figure 6.8: MediaStream API Workflow for Accessing User Media Devices

To obtain the media stream, the application prompts the user for permission to access their media devices via the getUserMedia() method. Once approval is granted, a dialog box enables users to select the appropriate audio and video devices during the conference.

A challenge encountered during the implementation process was related to React's component updates, which occur each time the state changes. It was crucial to ensure the getUserMedia() method was called only once, as multiple calls could result in the user's camera and audio access persisting even after the conclusion of the video conference.

To address this issue, the application was designed to ensure that the getUserMedia() method is invoked only once, preventing unnecessary and potentially risky access to the user's media devices.

After the user's selection, the MediaStream API retrieves audio and video tracks from the designated devices. These tracks are then enabled within a video tag in the HTML file, thus permitting users to engage in the video conference using their chosen instruments.

6.2.3 WebRTC Connection Establishment: Process and Optimization

Establishing a peer-to-peer connection using WebRTC, as illustrated in Figure 6.9, involves two potential options for initiating the connection.

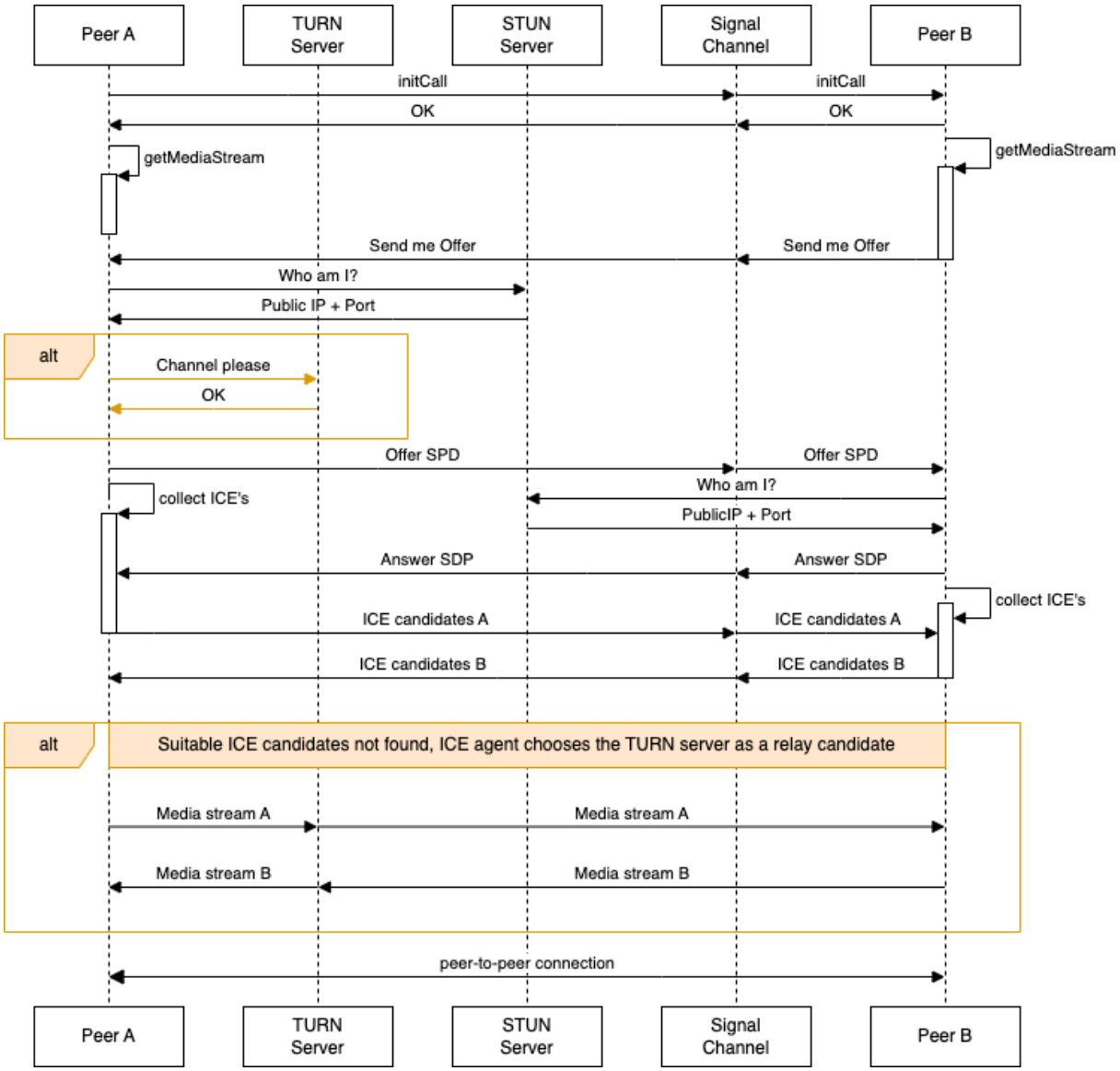

Figure 6.9: WebRTC Connection

Initially, a STUN server obtains the peer's public IP address and port. However, a TURN server becomes necessary when getting a peer's public IP address and port is impossible due to NAT restrictions. It is important to note that when a TURN server is employed, the media stream is routed through the server, thus compromising

the peer-to-peer connection. Usually using a STUN server is typically adequate for establishing a connection.

Moreover, enhancements have been implemented within the system to boost the speed and security of the connection process. Specifically, instead of transmitting each ICE candidate to the other peer individually, all ICE candidates are gathered and subsequently sent to the other peer. This approach facilitates a faster and more secure connection establishment process. The call initiator generates ICE candidates after creating the offer, while the receiver collects their ICE candidates after producing the answer. Overall, these measures contribute to the improved efficiency and reliability of the WebRTC connection process.

To establish a WebRTC connection between peer A and peer B, the following steps are executed:

1. Peer A initiates the call by transmitting a request through the signal channel, which subsequently forwards the request to peer B.
2. Upon receiving the call request from peer A, peer B accepts the call and responds to peer A via the signal channel, indicating that the call has been accepted.
3. Both peers are required to grant access to their video or audio device and make a selection. Once access is authorized, the local stream is displayed in the browser.
4. Following the establishment of the local stream, peer B sends a request to peer A, indicating readiness to receive an offer SDP.
5. Peer A sends a "who am I" request to a STUN server, which replies with the peer's public IP address and port. This information enables peer A to know its public IP address and port, even behind a NAT firewall.
6. Peer A forwards an offer SDP to peer B and starts the collection of ICE candidates.
7. Upon receiving the offer SDP from peer A, peer B generates an answer based on the offer and returns it to peer A. After creating the answer SDP, peer B also starts collecting its ICE candidates.
8. The signaling server facilitates the connection negotiation between peers A and B by exchanging network and device capability information. Peer A sends its ICE candidates to peer B, which reciprocates. The ICE candidates contain details about the peer's network and the various data travel paths between the two peers. The WebRTC library leverages this information to determine the optimal data travel path, ensuring a fast and efficient connection.
9. A peer-to-peer connection allows connected peers to directly transmit video, audio, files, and other data types.
10. In cases where direct peer-to-peer communication between two peers (peer A and peer B) is impossible due to firewalls or network restrictions, a TURN server serves as a fallback solution. The TURN server functions as a relay between the two peers, forwarding media streams between them. If a STUN server is

accessible, it will enable direct peer-to-peer communication, making the TURN server unnecessary. However, if the STUN server fails to establish a direct connection, the TURN server provides a backup solution.

6.3 Applying Code Splitting in the Extended Crew Active System

In implementing the Extended Crew Active System, code splitting and Progressive Web Applications (PWAs) were combined to enhance the performance and user experience of the web application.

Code splitting in React involves dividing the application code into smaller chunks, which are loaded only when required. Only the code defining the home page and user interface was loaded during the first render for the extended Crew Active System. The remaining chunks, corresponding to other pages and features, were downloaded later in the background without interrupting the user's activity.

In addition to code splitting, the Extended Crew Active System also utilized the PWA's caching capabilities for its app shell. This allowed the initial chunks to be loaded from cache storage, eliminating waiting for a server response. The combination of code splitting and PWA caching significantly improved the performance of the Extended Crew Active System.

As a best practice, code splitting was applied to every new route in the system. This ensured the application code had a corresponding code chunk for each route, allowing the system to load only the necessary code for each view or interaction. As a result, the user experience was further improved, contributing to the overall success of the extended Crew Active System implementation.

6.4 Testing

The advanced Crew Active web-based time recording system, incorporating a progressive web app (PWA) features integrated video streaming, was subjected to manual testing to guarantee its proper operation and efficacy. This method was chosen to assess the system's performance from the standpoint of end-users.

Manual testing confirmed the system's functionality, such as logging working hours, monitoring employee attendance, producing reports, and streaming video via WebRTC. Additionally, the PWA aspect of the system underwent manual testing to evaluate its performance and responsiveness.

Besides manual testing, the Lighthouse tool examined the PWA's performance. Lighthouse is an open-source, automated tool intended to assist developers in enhancing the quality of their web applications. It provided valuable insights into the performance, accessibility, and best practices of the PWA implementation.

As WebRTC testing is complex and limited tools are available for testing this technology, end-users primarily conduct the testing manually. They assessed the video stream quality, connection stability, and synchronization between video and audio streams.

Moreover, manual testing covered WebRTC data transfer protocols, error handling, edge case management, and the system's performance under various conditions. Since WebRTC testing is still in its early stages, manual testing remains the most effective method for evaluating WebRTC integration in the extended web-based time recording system, which functions like a progressive web app.

Although the challenges associated with WebRTC testing, manual testing, and Lighthouse have ensured that the system's video streaming and PWA implementation are reliable and meet the end users' needs.

6.5 Discussion of the Technical Challenges and Solutions

Several technical challenges were encountered while developing the Crew Active web-based time recording system, incorporating progressive web app (PWA) features an integrated video streaming. This section discusses these challenges and the solutions implemented to overcome them.

6.5.1 React Component Updates and getUserMedia()

One of the challenges faced during the implementation of the system was the behavior of React components, which update whenever there is a change in the state. This posed a problem when working with the getUserMedia() method. Multiple calls to the process could lead to the user's camera and audio access persisting even after the video conference had ended.

To address this issue, the system was designed to ensure that the getUserMedia() method was called only once. This solution prevented unnecessary and potentially risky access to the user's media devices, thus enhancing the security and reliability of the application.

6.5.2 WebRTC Connection Process

The WebRTC connection process presented some complexities, particularly regarding the collection and transmission of ICE candidates. To improve the speed and security of the connection process, all ICE candidates were gathered before being sent to the other peer instead of sending them individually.

This approach allowed for a more rapid and secure connection establishment process. The call initiator generated ICE candidates after creating the offer, while the receiver collected their ICE candidates after producing the answer. These measures contributed to the improved efficiency and reliability of the WebRTC connection process.

6.5.3 Manual Testing of the System

Given the complex nature of WebRTC technology and the limitations of available testing tools, manual testing was primarily employed to evaluate the system's performance. End-users assessed various aspects of the system, including video stream quality, connection stability, and synchronization between video and audio streams.

The PWA aspect of the system was also manually tested to ensure its responsiveness and proper functionality. In addition, the Lighthouse tool was utilized to examine the performance, accessibility, and best practices of the PWA implementation.

6.5.4 TURN Server as a Fallback Solution

In cases where direct peer-to-peer communication between peers was impossible due to firewalls or network restrictions, a TURN server was implemented as a fallback solution. The TURN server functioned as a peer relay, forwarding media streams between them. Although the TURN server compromised the peer-to-peer connection by routing the media stream through the server, it ensured that communication was still possible in challenging network environments.

Developing the Crew Active web-based time recording system with PWA features and integrated video streaming involved various technical challenges. However, through careful consideration and implementation of practical solutions, these challenges were successfully addressed, resulting in a reliable and secure system that meets the needs of end users.

6.6 Presentation of the results and evaluation of the system's performance

The extended Crew Active application has been designed to function as a Progressive Web Application, which enables users to access it seamlessly via a web browser. Upon navigating to the URL https://application.crew-active.de, users can install the application by selecting the "Add to Home Screen" feature.

6.6.1 Progressive Web App Features: Add to Home Screen, Splash Screen, and App Icon

Figure 6.10 shows the three key features of the implemented Progressive Web App within the system. The PWA offers users a more engaging and native-like experience, combining the best aspects of web and mobile applications.

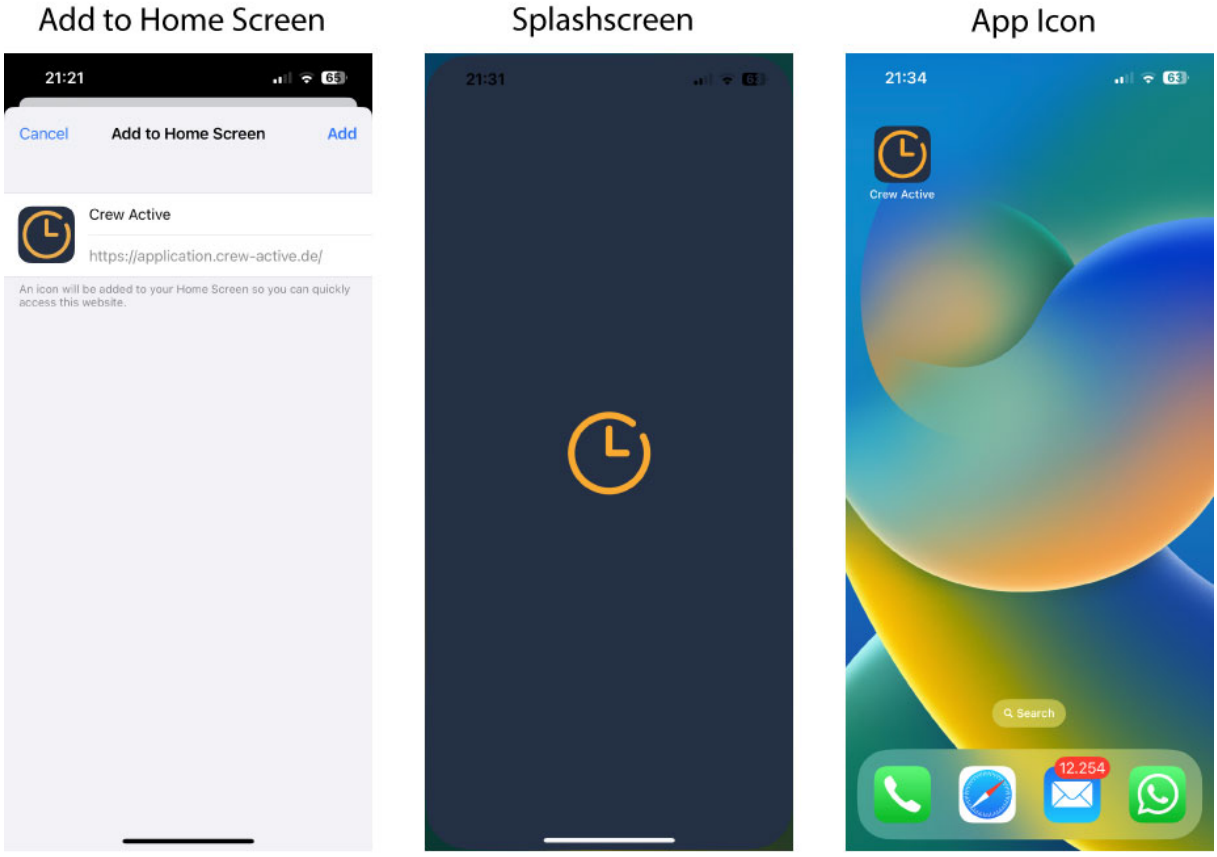

Figure 6.10: Key Features of the Progressive Web App: Add to Home Screen, Splash Screen, and App Icon

The first image in Figure 6.10 displays the "Add to Home Screen" feature, allowing users to easily install the PWA on their devices. This feature provides users with quick access to the application and enhances the user experience.

The second image in Figure 6.10 highlights the splash screen that appears when users launch the PWA. The splash screen serves as a visual placeholder while the application loads, offering a more polished and professional appearance.

Lastly, the third image in Figure 6.10 shows the app icon for the installed PWA. The app icon is visible on the user's device, making it easily recognizable and providing a more native-like experience.

6.6.2 Client-side Implementation: Network Status Indicator and Offline Operations

The client-side implementation features an icon button (1) that dynamically changes its appearance based on the network status (online/offline). As depicted in Figure 6.11, the device is currently offline, and the user has made some changes while disconnected from the network.

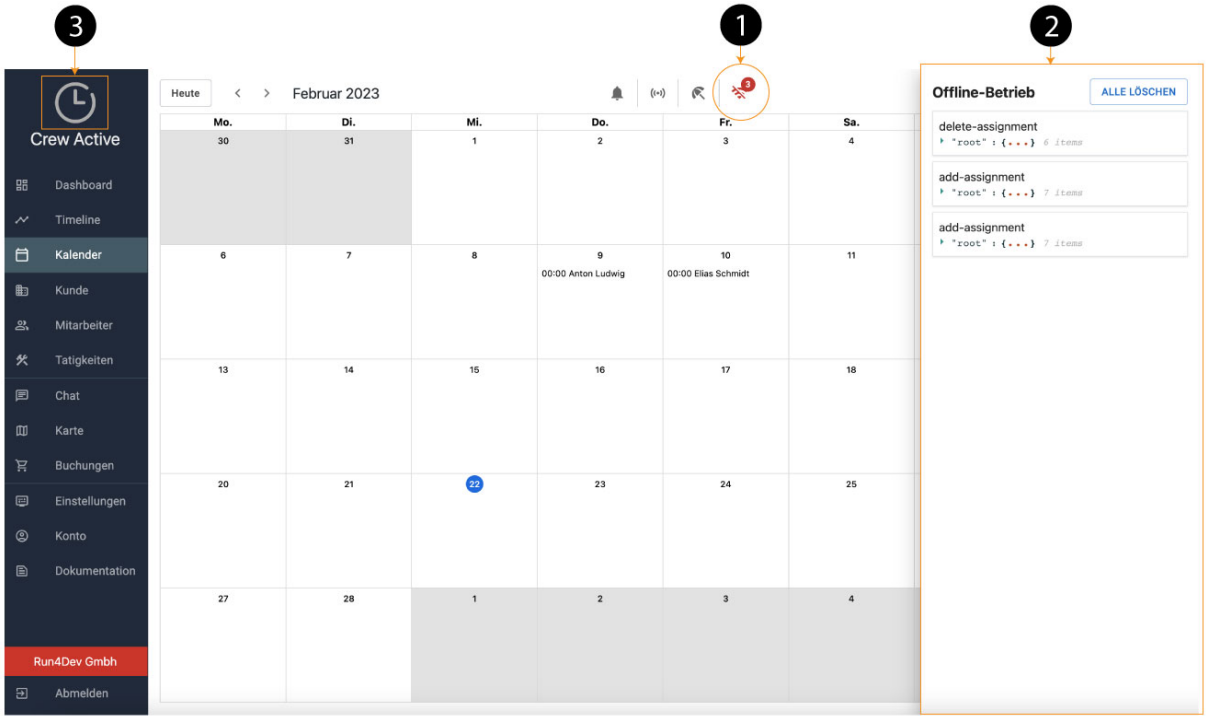

Figure 6.11: Network Status Indicator and Offline Operations Interface with (1) Icon Button, (2) Right-side Drawer, and (3) Crew Active Logo

The counter placed next to the network icon (1) represents the number of changes made in the offline mode. Upon clicking the icon, the right-side drawer (2) is launched, enabling users to view all the operations carried out in the offline mode. The user can also delete all recorded operations or examine each item in the list to learn more about the data that accompanies it.

The Crew Active logo (3), positioned in the top left corner of the interface, appears gray when the user is offline and turns orange upon connecting to the network. Material UI components and icons are utilized in the implementation to ensure the user interface

is easy to use and navigate. This approach eliminates the need to build components from scratch, saving time and resources.

6.6.3 Device Selection Dialog for Video Streaming

This section presents the implementation of a responsive dialog box that allows users to select their preferred audio and video devices for a video streaming application utilizing WebRTC, offering a seamless experience across various devices and screen sizes.

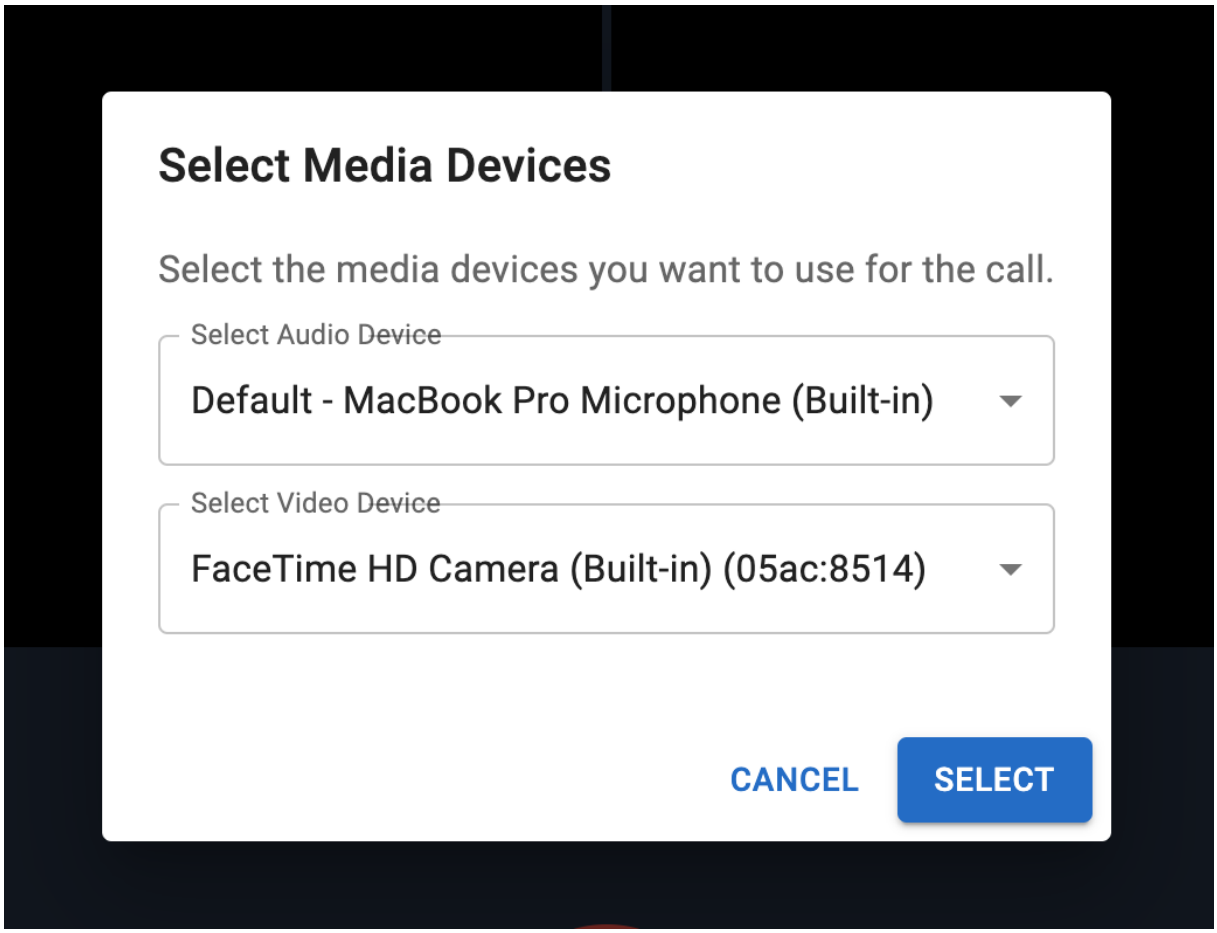

Figure 6.12: Responsive Device Selection Dialog for WebRTC Video Streaming

The dialog box uses TailwindCSS to ensure compatibility with modern web browsers and provide a visually appealing user interface. The dialog box contains two dropdown menus for selecting the audio input device (microphone) and the video input device (camera).

To populate the dropdown menus, the `navigator.mediaDevices.getUserMedia()` method is used, as it ensures compatibility with the Safari browser by requiring users to grant access to the devices before displaying them in the dropdown menus.

TailwindCSS utility classes create a responsive design that adapts the dialog box's appearance based on the screen size without needing CSS media queries. The dialog

box opens in fullscreen mode on mobile devices, making it easy for users to interact with the device selection options.

Upon testing, the device selection dialog successfully displays the available audio and video devices, enabling users to make their selections. The responsive design ensures an optimized user experience across various devices, with the dialog box adapting its size and appearance according to the screen size. The fullscreen mode on mobile devices provides a user-friendly interface for device selection.

6.6.4 Video Streaming Interface

This section presents the results of the developed video streaming interface within the system. The interface has been designed to facilitate seamless user communication using WebRTC technology, offering an engaging and user-friendly experience. The following discussion and Figure 6.13 provide insights into the design and functionality of this interface, highlighting the key features and responsive design aspects implemented.

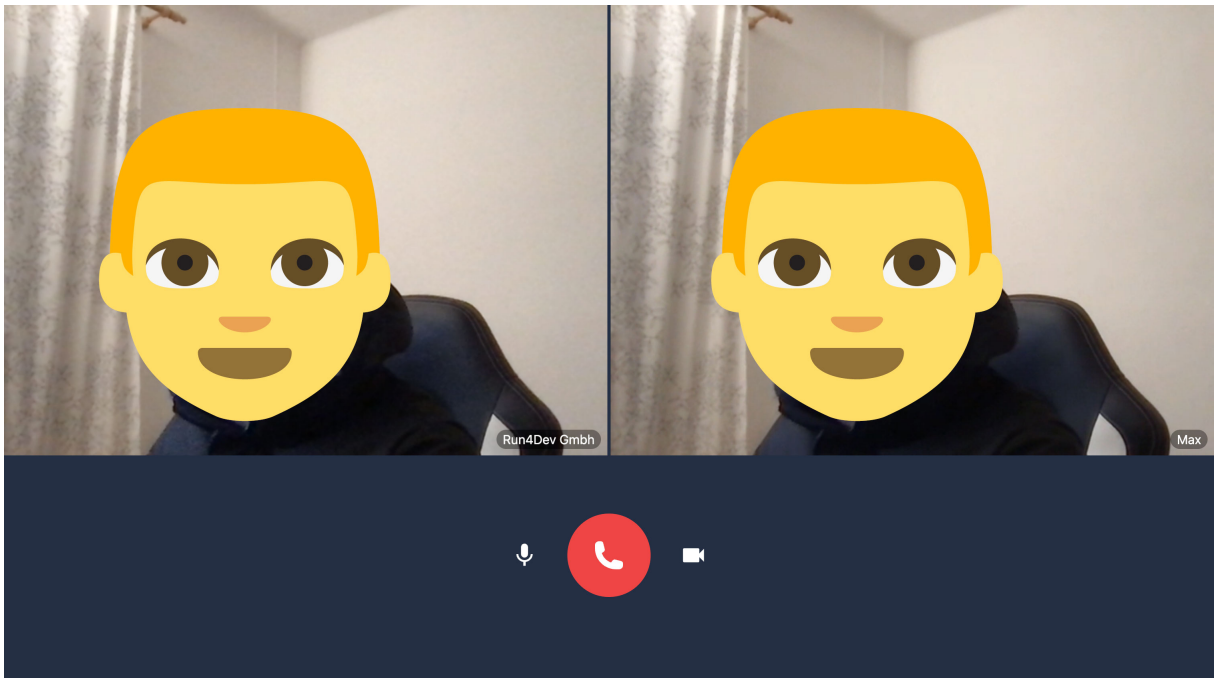

Figure 6.13: Responsive Video Streaming Interface with Local and Remote User Streams and Control Options

Figure 6.13 shows the result of the video streaming interface implemented using WebRTC technology within the system. The figure displays a scenario where two users engage in a video streaming session. The local user's video stream is positioned on the left side, while the remote user's stream appears on the right.

Each video stream has the user's name displayed at the bottom right corner, clearly identifying both participants. The interface also includes control options at the bottom of the screen, allowing users to mute or unmute their microphone, toggle their video display on or off, and decline the video streaming session if necessary.

The responsive design of the video streaming interface ensures that it adapts to fit any screen size, providing an optimized experience for users on various devices, including desktop computers, laptops, tablets, and smartphones.

6.7 Evaluating Crew Active System with Google Lighthouse

The Crew Active system was evaluated using Google Lighthouse, a powerful tool for analyzing the quality of web applications. The assessment focused on key performance indicators, including performance, accessibility, SEO, and PWA capabilities.

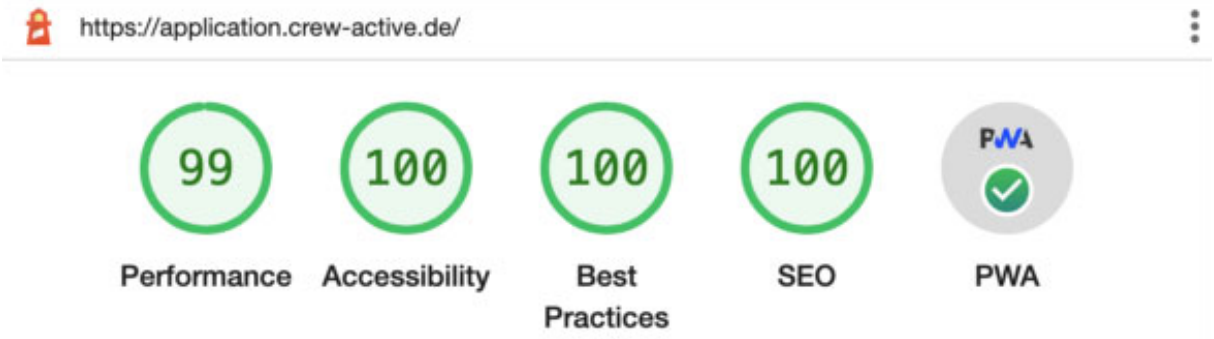

Figure 6.14: Extended Crew Active System Google Lighthouse Evaluation Results

As illustrated in Figure 6.14, the Crew Active system achieved excellent scores in the evaluation, with a 99% performance rating, 100% accessibility, and 100% SEO. Furthermore, the PWA assessment shows a green checkmark, indicating that it is installable and optimized.

The Crew Active system as shown in Figure 6.15 registers a service worker, is configured for a custom splash screen, sets a theme color for the address bar, and includes a <meta name="viewport"> tag with width or initial-scale. Additionally, it provides a valid apple-touch-icon and a manifest with a maskable icon, ensuring compatibility with various devices and platforms.

These results demonstrate the high quality and performance of the Crew Active system, confirming its effectiveness as a web application and its compliance with best practices for user experience, accessibility, and search engine optimization.

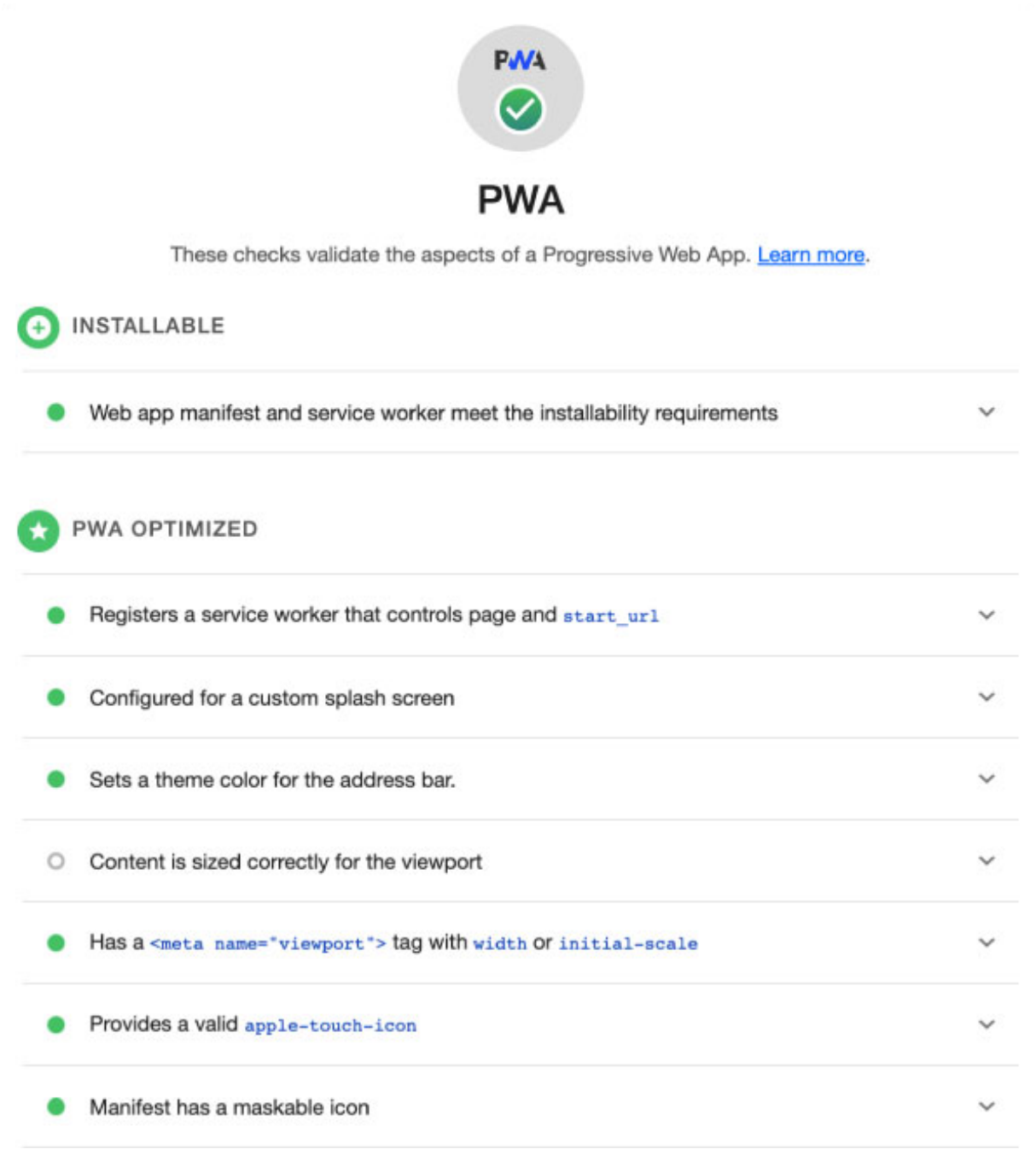

Figure 6.15: Extended Crew Active System Google Lighthouse PWA Evaluation Results

Chapter 7

Conclusion and Future Work

7.1 Summary of the main findings and contributions

The web-based time recording system Crew Active has been redesigned to meet some functional requirements, offline capability, video streaming between users, a user-friendly interface, and the transformation into a progressive web app. These requirements were made possible using key technologies, including MERN stack, PWA architecture, Redux Persist, WebRTC, Socket.IO, Material UI, Framer Motion, and Tailwind CSS.

Using the MERN stack enables Full-Stack JavaScript Development, allowing using a single programming language in front and backend. From a developer's perspective, this approach will enable an easy switch between different parts of the whole application. Nodejs and MongoDB will be in the backend, enabling high scalable and performant, allowing the application to handle a large amount of data and traffic. Express and React have a big community and are widely used, offering robust features and better performance. Material UI is used to have one of the best user experiences for each category of people. Socket.IO is an excellent library used as a signaling channel that can be integrated everywhere on in Web in Android or IOS applications. Framer Motion is used for different animations. Moreover, Tailwind CSS makes the application responsive and handles CSS in a more structured way.

Using the React PWA template will make implementing the PWA architecture in the system manageable, enabling offline capability. Redux-Persist is used to persist all the states of Redux and dispatch the persisted state from IndexedDb to the Redux state. This will allow the user to access the content when the user is offline and make changes that will be synced when the user is online for the first time. At the same time, WebRTC allows peer-to-peer video streaming between users.

Overall the extended Crew Active system provides a practical and cutting-edge time-recording system solution that meets the need of the users. The thesis demonstrates the power of the PWA architecture, WebRTC, and other critical technologies by improving performance and offering new features for the system.

7.2 Limitations

With the new features that have won after fulfilling the functional requirements, the extended Crew Active system also has limitations and future potential for improving the system.

Limitations:

- While the system can be used offline for creating and changing tasks, other features require an internet connection, which can lead to limiting the system's accessibility and functionality in certain situations.
- The system has an administrator view responsible only for desktop and tablet devices but not for mobile devices. This could limit the flexibility and accessibility of the system, especially for administrators who needs to add new workers when offline.
- The system requires significant resources to maintain and develop new features, including time and costs.
- Video streaming functionality is only enabled on web base systems, not in the IOS or Android application. This can limit the system functionality for the user primarily using the mobile application.
- The system is all the time under development. It can make user adoption complex or confusing.

Also that the extended Crew Active time recording system has limitations; addressing this limitation and improving the system functionality will lead to a more effective and impactful solution. The system could also be adapted to meet the needs of other industries or use cases, providing a more versatile solution for organizations.

7.3 Suggestions for Further Improvements

The extended Crew Active system offers a range of features, but there is still enormous potential for enhancements. Implementing additional improvements could help users achieve their goals more efficiently and at lower costs.

Some suggested improvements include:

- Enhanced Offline Functionality: The current offline functionality can be further improved by enabling other operations that typically require an internet connection to be performed offline.
- Video Streaming in Native Applications: Crew Active can be enhanced by integrating video streaming into a native app.
- Improved Security: Implementing two-factor authentication can increase security by preventing unauthorized access to sensitive data.
- Integration of Artificial Intelligence: Incorporating artificial intelligence can enable user recommendations and automate various processes.
- Multilingual Support: Expanding Crew Active to support multiple languages can broaden its appeal to a wider user base.
- Screen Sharing and Multi-Participant Video Conferencing: Enabling screen sharing and video conferences with multiple participants will improve the user experience by allowing simultaneous collaboration with various participants.
- User Feedback Collection and Analysis: Gathering user feedback and experiences will enable better system performance analysis and inform decisions for future improvements.

By exploring these areas, the extended Crew Active system can undergo further enhancements and introduce new features, providing a superior user experience and catering to a broader range of needs.

List of Figures

Bibliography

[1] URL: https://caniuse.com/?search=service%20workers (visited on 05/03/2023).

[2] *A Study of WebRTC Security*. URL: https://webrtc-security.github.io/ (visited on 25/02/2023).

[3] *A Study of WebRTC Security*. URL: https://webrtc-security.github.io/#ref.5.

[4] *Amazon Kinesis Video Streams WebRTC Developer Guide*. URL: https://docs.aws.amazon.com/pdfs/kinesisvideostreams-webrtc-dg/latest/devguide/kinesisvideo-dg.pdf (visited on 08/03/2023).

[5] Oahehc Andrew. *ServiceWorker: Lifecycle, Update, and Notification*. URL: https://medium.com/@oahehc/serviceworker-lifecycle-update-and-notification-439c998e1e59 (visited on 28/02/2023).

[6] Ross Benes. *With new mobile site, Forbes boosted impressions per session by 10 percent*. URL: https://digiday.com/media/new-mobile-site-forbes-boosted-impressions-per-session-10-percent/ (visited on 22/03/2023).

[7] caniuse. *WebRTC Browser Compatibility*. URL: https://caniuse.com/?search=webrtc (visited on 11/02/2023).

[8] Google Search Central. *SEO for Single Page Apps*. URL: https://www.youtube.com/watch?v=l-JWN2x_Na0&t=83s (visited on 08/03/2023).

[9] Cloudflare. *What is Web Application Security?* URL: https://www.cloudflare.com/learning/security/what-is-web-application-security/ (visited on 07/02/2023).

[10] Google Developers. *Offline data*. URL: https://web.dev/learn/pwa/offline-data/ (visited on 06/03/2023).

[11] Google Developers. *Storage for Progressive Web Apps*. URL: https://developers.google.com/web/fundamentals/instant-and-offline/web-storage (visited on 06/03/2023).

[12] Google Developers. *What are Progressive Web Apps?* URL: https://web.dev/what-are-pwas/ (visited on 10/03/2023).

[13] MDN Web Docs. *IndexedDB API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (visited on 06/03/2023).

[14] MDN Web Docs. *Service Worker API - Cache*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Cache (visited on 06/03/2023).

[15] MDN Web Docs. *Web Storage API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API (visited on 06/03/2023).

[16] Facebook. URL: https://github.com/facebook/react (visited on 10/03/2023).

[17] Facebook. *Making a Progressive Web App*. URL: https://create-react-app.dev/docs/making-a-progressive-web-app/ (visited on 09/03/2023).
[18] Facebook. *React Documentation*. URL: https://beta.reactjs.org/learn/your-first-component (visited on 08/02/2023).
[19] Facebook. *React the library for web and native user interfaces*. URL: https://react.dev/ (visited on 10/03/2023).
[20] ForgeRock. *Integrate a single-page app*. URL: https://backstage.forgerock.com/docs/sdks/latest/blog/IntegrateyourSPAwithFIDC.html (visited on 08/03/2023).
[21] Vitaly Fredman. *10 Best Web Development Frameworks to Use in 2023*. URL: https://hackr.io/blog/web-development-frameworks (visited on 07/02/2023).
[22] Vitaly Fredman. *Responsive Web Design: What It Is And How To Use It*. URL: https://www.smashingmagazine.com/2011/01/guidelines-for-responsive-web-design/ (visited on 07/02/2023).
[23] *Getting started with peer connections*. URL: https://webrtc.org/getting-started/peer-connections (visited on 10/03/2023).
[24] Alexander S. Gillis. *Native App*. URL: https://www.techtarget.com/searchsoftwarequality/definition/native-application-native-app (visited on 10/03/2023).
[25] *Handling Service Worker updates – how to keep the app updated and stay sane*. URL: https://whatwebcando.today/articles/handling-service-worker-updates/ (visited on 25/02/2023).
[26] Dave Hudson. *Progressive Web App Splash Screens*. URL: https://medium.com/@applification/progressive-web-app-splash-screens-80340b45d210 (visited on 28/02/2023).
[27] *Introduction to WebRTC protocols*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols#nat (visited on 10/03/2023).
[28] Charithra Kariyawasam. *Introduction to CSRF*. URL: https://medium.com/@charithra/introduction-to-csrf-a329badfca49 (visited on 25/02/2023).
[29] Yuri Luchaninov. *Using Node.js for Backend Web Development in 2023*. URL: https://mobidev.biz/blog/node-js-for-backend-development (visited on 07/02/2023).
[30] Yurii Luchaninov. *WebRTC App Development: Use Cases, Challenges, and the Future*. URL: https://mobidev.biz/blog/webrtc-app-development-challenges-use-cases-future (visited on 14/02/2023).
[31] MDM. *How to make PWAs re-engageable using Notifications and Push*. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Re-engageable_Notifications_Push (visited on 10/03/2023).
[32] Rahul Surendra Mishra. *Progressive WEBAPP : Review*. URL: https://www.irjet.net/archives/V3/i6/IRJET-V3I6568.pdf (visited on 28/02/2023).
[33] Kyle Mistele. *XSS: What it is, how it works, and how to prevent it*. URL: https://medium.com/codelighthouse/xss-what-it-is-how-it-works-and-how-to-prevent-it-454629e3a0da (visited on 25/02/2023).
[34] Monocubed. *List of 10 Best Front end Frameworks to Use For Web Development*. URL: https://www.monocubed.com/blog/best-front-end-frameworks/ (visited on 08/03/2023).

[35] mozilla. *Service worker concepts and usage*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API#service_worker_concepts_and_usage (visited on 28/02/2023).
[36] mozilla. *Using media queries*. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries (visited on 07/02/2023).
[37] Vipul Nema. *Understanding service worker life cycle*. URL: https://medium.com/@vipulnema2610/understanding-service-worker-life-cycle-b6580aa4eb50 (visited on 25/02/2023).
[38] Mozilla Developer Network. *WebRTC API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API (visited on 12/02/2023).
[39] Odunayo Ogungbure. *Progressive Web Apps: Caching Strategies*. URL: https://dev.to/mr_steelze/progressive-web-apps-caching-strategies-mf2 (visited on 28/02/2023).
[40] Addy Osmani. *A Pinterest Progressive Web App Performance Case Study*. URL: https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154 (visited on 22/03/2023).
[41] Addy Osmani. *A Tinder Progressive Web App Performance Case Study*. URL: https://medium.com/@addyosmani/a-tinder-progressive-web-app-performance-case-study-78919d98ece0 (visited on 22/03/2023).
[42] Addy Osmani. *Instant Loading Web Apps with an Application Shell Architecture*. URL: https://developer.chrome.com/blog/app-shell/ (visited on 28/02/2023).
[43] Yuvraj Pandey. *Frontend Performance Optimization with Code Splitting*. URL: https://yuvrajpy.medium.com/frontend-performance-optimization-with-code-splitting-using-react-lazy-suspense-1e0d1a85e32c (visited on 06/03/2023).
[44] Vatsal Patel. *An Ultimate Guide for Mobile App Development*. URL: https://medium.com/nerd-for-tech/an-ultimate-guide-for-mobile-app-development-ff3c17ece87a (visited on 10/03/2023).
[45] Tuomas Pekkanen. *PACKAGING COMPLEX WEB CLIENT IN EASILY EMBEDDABLE SOLUTION*. URL: https://trepo.tuni.fi/bitstream/handle/10024/140638/PekkanenTuomas.pdf (visited on 09/03/2023).
[46] *Progressive Web Apps*. URL: https://web.dev/learn/pwa/progressive-web-apps/ (visited on 01/03/2023).
[47] *Redux Documentation*. URL: https://redux.js.org/introduction/getting-started (visited on 08/02/2023).
[48] *Redux Fundamentals, Part 2: Concepts and Data Flow*. URL: https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow#reducers (visited on 08/02/2023).
[49] Grand View Research. *Web Real-Time Communication Market Size*. URL: https://www.grandviewresearch.com/industry-analysis/web-real-time-communication-market (visited on 11/02/2023).
[50] rt2zz. *redux-persist*. 2015. URL: https://github.com/rt2zz/redux-persist (visited on 09/03/2023).
[51] Jamie Maria Schouren. *Web App Manifests — A Guide to get your website on a user's Home Screen*. URL: https://medium.com/deity-io/web-app-manifests-

a-guide-to-get-your-website-on-a-users-home-screen-6baf108538e7 (visited on 06/03/2023).
[52] Amazon Web Services. *Amazon S3*. URL: https://aws.amazon.com/s3/ (visited on 06/03/2023).
[53] Pawel Skolski. *Single-page application vs. multiple-page application*. URL: https://neoteric.eu/blog/single-page-application-vs-multiple-page-application/ (visited on 07/02/2023).
[54] Tree Web Solutions. *SPAs and Server Side Rendering: A Must, or a Maybe?* URL: https://treewebsolutions.com/articles/spas-and-server-side-rendering-a-must-or-a-maybe-55 (visited on 08/03/2023).
[55] Kavirajan ST. *What is WebRTC and How to Setup STUN/TURN Server for WebRTC Communication?* URL: https://medium.com/av-transcode/what-is-webrtc-and-how-to-setup-stun-turn-server-for-webrtc-communication-63314728b9d0 (visited on 10/03/2023).
[56] stackpath. *WHAT IS A WEB APPLICATION?* URL: https://www.stackpath.com/edge-academy/what-is-a-web-application/ (visited on 07/02/2023).
[57] tailwindCSS. *Responsive Design*. URL: https://tailwindcss.com/docs/responsive-design (visited on 06/03/2023).
[58] *The service worker lifecycle*. URL: https://web.dev/service-worker-lifecycle/ (visited on 25/02/2023).
[59] *The Ultimate Guide To Progressive Web Apps in 2023 – with 50 PWA Examples*. URL: https://www.mobiloud.com/blog/progressive-web-app-examples (visited on 09/03/2023).
[60] thinkwithgoogle.com. *The next billion users: trivago embrace Progressive Web Apps as the future of mobile*. URL: https://www.thinkwithgoogle.com/intl/en-154/marketing-strategies/app-and-mobile/trivago-embrace-progressive-web-apps-as-the-future-of-mobile/ (visited on 10/03/2023).
[61] Joe Tuan. *Progressive Web Apps vs. Native Apps in 2023: Pros And Cons*. URL: https://topflightapps.com/ideas/native-vs-progressive-web-app/ (visited on 16/02/2023).
[62] *Using WebRTC data channels*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Using_data_channels (visited on 08/03/2023).
[63] Anton Venema. *Diagnosing Network Problems with WebRTC Applications*. URL: https://www.liveswitch.io/blog/diagnosing-network-problems-with-webrtc-applications (visited on 10/03/2023).
[64] Eni Veshi. *Digitale Zeiterfassung*. URL: https://www.crew-active.de/ (visited on 09/03/2023).
[65] Eni Veshi. *Zeiterfassung für Mitarbeiter*. URL: https://docs.crew-active.de/ (visited on 09/03/2023).
[66] w3c. *Web Application Manifest*. URL: https://w3c.github.io/manifest/ (visited on 28/02/2023).
[67] *Web App Manifest*. URL: https://www.w3.org/TR/2016/WD-appmanifest-20160315/ (visited on 09/03/2023).
[68] web.dev. *Alibaba*. URL: https://web.dev/alibaba/ (visited on 10/03/2023).

[69] web.dev. *Progressive Web App Capabilities*. URL: https://web.dev/learn/pwa/capabilities/ (visited on 20/02/2023).
[70] web.dev. *PWAs leverage modern web capabilities*. URL: https://web.dev/drive-business-success/ (visited on 10/03/2023).
[71] *WebRTC*. URL: https://webrtc.org/ (visited on 10/03/2023).
[72] *WebRTC*. URL: https://github.com/webrtc (visited on 10/03/2023).
[73] *WebRTC adapter*. URL: https://github.com/webrtcHacks/adapter (visited on 10/03/2023).
[74] *WebRTC TURN server: Everything you need to know*. URL: https://www.100ms.live/blog/webrtc-turn-server (visited on 08/03/2023).
[75] *WebRTC: Real-Time Communication in Browsers*. W3C. URL: https://www.w3.org/TR/webrtc/ (visited on 10/03/2023).
[76] *Weekendesk crée une expérience ultra-fluide en réconciliant site web et app native*. URL: https://www.thinkwithgoogle.com/_qs/documents/8971/Weekendesk_PWA_-_EXTERNAL_CASE_STUDY.pdf (visited on 10/03/2023).
[77] *What is a data channel?* URL: https://developer.mozilla.org/en-US/docs/Games/Techniques/WebRTC_data_channels (visited on 08/03/2023).
[78] *What is WebRTC?* URL: https://bloggeek.me/what-is-webrtc/ (visited on 02/03/2023).
[79] whatpwacando.today. *What PWA Can Do Today*. URL: https://whatpwacando.today/ (visited on 18/02/2023).
[80] Simon Wicki. *PWA: Create a "New Update Available" Notification using Service Workers*. URL: https://medium.com/progressive-web-apps/pwa-create-a-new-update-available-notification-using-service-workers-18be9168d717 (visited on 28/02/2023).
[81] *Workbox*. URL: https://web.dev/learn/pwa/workbox/ (visited on 07/03/2023).
[82] *Workbox usage*. URL: https://almanac.httparchive.org/en/2021/pwa#workbox-usage (visited on 07/03/2023).