Hochschule für Telekommunikation Leipzig (HfTL)

PROFILIERUNG NETZBASIERTE ANWENDUNGEN

PROJEKTDOKUMENTATION

Cache und Push-Notifications in mobilen Webanwendungen

Umsetzung mittels Service Worker Technologie

David Howon (147102)

Michael Müller (147105)

Wintersemester 2016/17

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**
2.1 Serviceworker . . . 2
2.2 Web Push API . . . 2

3 Anforderungen **3**
3.1 funktionale Anforderungen . . . 3
3.2 nicht-funktionale Anforderungen . . . 4

4 Konzeption **6**
4.1 Offlinefähigkeit . . . 6
4.1.1 Caching statischer Ressourcen . . . 6
4.1.2 lokale persistente Speicherung und Synchronisation des Application State . . . 8
4.2 Web Push . . . 9
4.2.1 Ablauf . . . 9
4.3 Architekturbeschreibung . . . 11
4.4 Applicationserver . . . 12
4.4.1 Datenbank . . . 12
4.4.2 REST-API . . . 12
4.5 Datenmodel . . . 13
4.6 Client-Oberfläche . . . 14

5 Implementierung **15**
5.1 Applicationserver . . . 15
5.1.1 Node.js Express-Server . . . 15
5.1.2 Datenbank-Server anbinden . . . 16
5.1.3 REST-Schnittstelle . . . 16
5.1.4 Cross-Origin Resource Sharing . . . 19
5.2 Service Worker . . . 20
5.2.1 Installation . . . 20
5.2.2 Caching der statischen Ressourcen . . . 21
5.2.3 Push-Nachrichten abonnieren . . . 23
5.2.4 Background Sync . . . 26

6 Ausblick **28**

1 Einleitung

Fast jeder Mensch benutzt Sie, installiert Sie und wendet sie an. Die Rede ist von nativen App‘s, hierbei handelt es sich um installierbare und ausführbare Applikationen, die auf Mobilgeräten, Desktop PC‘s oder aber auf Tablets ihren Dienst verrichten und uns das Leben vereinfachen oder aber auch einfach nur Informieren und zu Kommunikation genutzt werden. Eine weitere Technologie die mindestens genauso Anklang findet, sind herkömmliche Webseiten von Anbietern wie z.B. Reiseunternehmen, Firmen mit einer Internetpräsenz und viele weitere.

In den letzten Jahren kam der Begriff der „Progressive Web Apps“ auf, doch was bedeutet dieser und was für Vorteile kann uns diese neue in den Anfängen steckende Technologie bieten? Haben „Progressive Web Apps“ die Möglichkeit, herkömmliche Technologien wie traditionelle native Apps zu verdrängen und was macht Sie aktuell so beliebt?

Eine Progressive Web App verbindet die Vorteile zweier Welten; HTML/CSS-basierte Webseiten und native Apps. Sie ist wie eine normale Webseite in einem Browser aufrufbar und schafft aber ein authentisches App-Erlebnis gegenüber dem User. Das authentische Gefühl, wird dem User durch Techniken, die auch in herkömmlichen, traditionellen Apps zum Tragen kommen, wie etwa Push-Nachrichten, die über auftretende Ereignisse informieren oder schnelle Ladezeiten und generelle Nutzung der App vermittelt, selbst bei schlechter Internetverbindung. Die Offline-Bedienbarkeit und das Echtzeit-Erlebnis ermöglichen eine permanente Verfügbarkeit der Webseite. Im Gegensatz zu traditionellen nativen Apps für Tablet, Smartphone und Desktop PC können progressive Web Apps ohne eine Installation auf den Endgeräten auskommen.

Durch die Implementierung der Technologie „Service Worker“ von großen Browserherstellern wie Google und Firefox, wird es überhaupt erst möglich Push-Nachrichten zu empfangen und zu verarbeiten.

Doch hält diese Technologie, was sie verspricht und mit welchen Techniken kann das umgesetzt werden? Deshalb wurde dieses Projekt ins Leben gerufen, um genau diese neue Technologie, der „Progressiven Web App“ zu untersuchen und am Beispiel einer simplen Anwendung zu verdeutlichen.

Im Rahmen der Profilierung „netzbasierte Anwendungen“ an der Hochschule für Telekommunikation Leipzig (HfTL) betrachtet diese Dokumentation die Möglichkeiten der Offlinefähigkeit und verarbeitung von Pish-Benachrichtigungen in progressiven Web Apps.

2 Grundlagen

2.1 Serviceworker

Ein Service Worker ist eine W3C-Standard-Webtechnik bei der JavaScript-Code im Hintergrund von Web-Browsern ausgeführt wird. Mit Hilfe von Service Worker ist es möglich, essentielle Funktionalitäten wie Caching zur Offline-Verwendbarkeit (z.B. bei Ausfall der Internetverbindung) von Web-Anwendungen, Aktualisierungen von Inhalten im Hintergrund, aber auch die von nativen Apps bekannten Push-Benachrichtigungen(Push-Notifications) zu ermöglichen. Dies findet alles im Hintergrund des Browsers statt und macht somit eine Installation von Software oder Software-Diensten unnötig.

Der Service Worker kann zum einen als Proxy fungieren und zum anderen vom Server gesendete Benachrichtigungen, selbst dann empfangen, wenn gerade keine Webseite der entsprechenden Domain geöffnet ist.

2.2 Web Push API

Bei Web Push handelt es sich um eine Erweiterung des bekannten Service Worker Standards. Solange der Browser geöffnet ist können Benachrichtigungen von Webseiten empfangen werden, selbst wenn der eigentliche Tab nicht geöffnet ist. So kann man zum Beispiel einen E-Mail-Tab schließen und trotzdem über eingehende Mails informiert werden. Da keine zusätzlichen Apps oder Text-Nachrichten für direkte Notifications nötig sind, ergibt sich ein großer Vorteil für Speichernutzung, Performance und Akkulaufzeit von Mobilgeräten.
Web Push benötigt genauso wie die Standortfreigabe oder der Kamerazugriff eine (jederzeit wiederrufbare) Berechtigung, bevor eine Webseite auf Push-Events reagieren und Notifications anzeigen kann.

Durch eine ständige Verbindung zu einem Push Service in unserem Fall „Firebase Cloud Messaging“ , der als zentrale Schaltstelle für Nachrichten fungiert, werden Web-Push-Benachrichtigungen ermöglicht. Ursprünglich betrieb jeder Browser-Anbieter einen eigenen Push-Service zum Schutz der Privatsphäre. Erst kürzlich wurden aber GCM (Google Cloud Messaging Push Service von Google) und Firebase (Mozilla Firefox Push Service) zu Firebase Cloud Messaging zusammengelegt.
Dabei erhält jede Webseite einen anderen, anonymen Web Push Identifier zur Verhinderung von seitenübergreifenden Zuordnungen. Zudem müssen die Nutzerdaten über ein Public-Key-Verfahren verschlüsselt werden. Der Service Worker meldet sich nur beim Push-Dienst an, wenn der User die notwendigen Push-Berechtigungen erteilt hat.

3 Anforderungen

Um die Möglichkeiten der Service Worker Technologie untersuchen zu können, wird eine relativ simple Webanwendung konstruiert. Diese dient als gedankliches Gerüst, um der Technologie einen praktischen Aufhänger zu geben. Es ist nicht Anspruch der Arbeit, eine voll-funktionsfähige Webanwendung zu entwickeln.

Nach erfolgreicher Registrierung und Anmeldung können Aufgaben angelegt, bearbeitet, angezeigt und gelöscht werden. Eine Kontaktliste ermöglicht es, Benutzer, die ebenfalls für die Anwendung registriert sind, zu einer persönlichen Freundesliste hinzuzufügen. Push-Benachrichtigungen informieren Benutzer über erhaltene Freundschaftsanfragen bzw. über Annahme/Verweigerung einer Anfrage.

Erstellte Aufgaben können mit Freunden geteilt werden. Der Besitze einer Aufgabe hat die Möglichkeit, die Aufgabe weiter zu bearbeiten bzw. zu löschen. Alle involvierten Benutzer werden über Änderungen an einer Aufgabe mittels Push-Benachrichtigungen informiert.

Die Website soll dem Benutzer das Look & Feel einer nativen Android-App vermitteln. Die Verwendung soll unabhängig einer aktiven Netzwerkverbindung reibungslos funktionieren. Außerdem soll diese als „Progressive Webanwendung“ entwickelt werden.

3.1 funktionale Anforderungen

Im Rahmen dieser Dokumentation werden unter funktionalen Anforderungen diejenigen verstanden, welche zur direkten Zielerfüllung beitragen. Darunter zählen die Anforderung einer „Single Page Application“, Bereitstellung einer geeigneten Offline-Benutzererfahrung sowie die Unterstützung von Push-Benachrichtigungen in mobile Webanwendungen. Die Schnittstelle zum Applicationserver wird ebenfalls unter den funktionalen Anforderungen geführt.

[FA-1] Single Page (Mobil) Application. Die Webanwendung wird als hybride Single Page Application entwickelt. Der Sitzungszustand („Application State“) wird auf dem Client gespeichert und im Hintergrund mit dem Applicationserver synchronisiert. Sowohl die statischen Ressourcen als auch die Präsentationslogik werden in der „Boot-Phase“ einmal vom Client geladen und Inhalte zur Laufzeit dynamisch angepasst. Während der Navigation wird der Präsentationsfluss im Client nicht angehalten bzw. durch „neu Laden“ unterbrochen.

[FA-2] Offlinefähigkeit. Die Benutzung der Webanwendung soll nicht ausschließlich bei bestehender Internetverbindung, sondern ebenfalls Offline reibungslos möglich sein. Dazu bietet die progressive Webanwendung Mechanismen zum Vorhalten der statischen Ressourcen und des Sitzungszustands der Anwendung („Application State“) im Offlinemodus. Benutzer werden über ggf. eingeschränkte Funktionalitäten informiert, während keine aktive Internetverbindung vorhanden ist.

[FA-3] Push-Benachrichtigungen. Benutzer der Webanwendung werden unabhängig vom verwendeten Endgerät über bestimmte Ereignisse mit Hilfe von Push-Benachrichtigungen informiert. Diese Ereignisse werden vom Applicationserver ausgelöst/verarbeitet und dieser initiiert Push-Benachrichtigungen beim Client.

[FA-4] Schnittstelle für Kommunikation mit Applicationserver. Der API Server unterstützt folgende Anforderungen um die Funktionalitäten einer RESTful-Schnittstelle zu erfüllen:

- Bereitstellung von CRUD[1]-Funktionalität für Entities
- Aufruf von Ressourcen über eindeutige und einfache URLs (z.B. `https://example.de/api/task/` und `https://example.de/api/task/:taskId`)
- Verwendung der standardisierten HTTP-Methoden (GET, POST, PUT und DELETE)
- Rückgabe im JSON-Format
- alle Requests werden auf der Konsole ausgegeben

[FA-5] Gesicherter Zugriff auf API. Der Zugriff auf die API ist nur für authentifizierte Benutzer möglich. Bei Ressourcenanforderung muss ein Token im `Authorization-Header` oder `Request-Body` übergeben werden. Im Fehlerfall sollen aussagekräftige und passende HTTP-Fehler zurückgegeben werden.

3.2 nicht-funktionale Anforderungen

Im Rahmen dieser Dokumentation werden unter nicht-funktionalen Anforderungen diejenigen verstanden, welche nicht zur direkten Zielerfüllung beitragen. Darunter fallen vor allem Anforderungen an die Beispielapplikation. Die folgenden Anforderungen beschreiben im Großen und Ganzen den Funktionsumfang der Client-Applikation. Darüber hinaus werden Ereignisse definiert, die innerhalb der Beispielanwendung, welche die Anzeige von Push-Benachrichtigungen auslösen.

[1] *CRUD: create, read, update, delete*

[NFA-1] Look & Feel einer nativen Android-App. Die Website soll sich optisch an den „Material Design“-Richtlinien für Android-Applikationen orientieren. Auf mobilen Endgeräten soll die Touch-Unterstützung genauso gewährleistet sein, wie die für Smartphones typischen Wisch-Gesten. Das Framework „JQuery-Mobile“ mit zugehörigem UI-Framework „JQuery-Mobile-UI“ kann als JQuery-Erweiterung zu diesem Zweck eingesetzt werden.

[NFA-2] Benutzerauthentifizierung. Benutzer können sich für die Nutzung der Anwendung Registrieren und anschließend Anmelden. Für die Registrierung ist ein eindeutiger Benutzername mit Angabe einer E-Mail Adresse sowie ein Passwort notwendig.

[NFA-3] Kontaktliste. Benutzer können sich untereinander mittels Benutzername bzw. E-Mail Adresse zur persönlichen Kontaktliste hinzufügen. Benutzer werden über Freundschaftsanfragen benachrichtigt und können diese bestätigen oder ablehnen.

[NFA-4] Aufgaben anlegen, bearbeiten und löschen. Ein authentifizierter Benutzer kann Aufgaben anlegen und anschließend Bearbeiten oder Löschen. Eine Aufgabe muss einen Titel besitzen. Optional kann eine Beschreibung und ein Fälligkeitsdatum hinterlegt werden.

[NFA-5] Aufgaben teilen. Aufgaben können mit mehreren Benutzern geteilt werden. Benutzer werden über die Einladung zu einer Aufgabe informiert. Nachdem die Einladung bestätigt wurde, wird ein Benutzer über Änderungen an der Aufgabe benachrichtigt.

[NFA-6] Ereignisse für Benachrichtigungen. Benutzer werden über Freundschaftsanfragen und Änderungen an Aufgaben informiert, in denen sie involviert sind. Benachrichtigungen werden nur angezeigt, wenn die Webanwendung nicht aktiv ist. Die Website gilt als „aktiv“, wenn sie auf dem Bildschirm angezeigt wird.
Folgende Ereignisse lösen eine Benachrichtigung aus:

- Freundschaftsanfrage wurde von einem anderen Benutzer gestellt
- Freundschaftsanfrage wurde durch einen anderen Benutzer bestätigt/abgelehnt
- ein anderer Benutzer hat die eigene Freundschaftsanfrage bestätigt/abgelehnt
- Einladung zu einer Aufgabe durch einen anderen Benutzer
- Bestätigung/Ablehnung durch einen Benutzer auf eine Einladung zu einer Aufgabe
- Änderungen an einer Aufgabe, an welcher der Benutzer beteiligt ist

4 Konzeption

4.1 Offlinefähigkeit

Konventionelle Webanwendungen benötigen eine dauerhafte Verbindung zum einem Webserver, um Ressourcen abzufragen. Selbst bei kurzen Verbindungsabbrüchen ist eine Arbeit mit solchen Anwendung unmöglich und in vielen Szenarien nicht akzeptabel. In den folgenden Abschnitten 4.1.1 und 4.1.2 werden Methoden beschrieben, um dieses Problem mit Hilfe der Service Worker Technologie zu lösen und die Benutzererfahrung, durch Offlinefähigkeit einer Webanwendung, zu verbessern.

Grundsätzlich kommen für das vorliegende Szenario zwei Varianten der Offlinefähigkeit in Frage. Die Variante **Überwindung von kurzen Verbindungsabbrüchen** sorgt dafür, dass die Anwendung trotz kurzfristiger Verbindungsabbrüche weiter reagiert. Die Anwendung kann die Verbindung beispielsweise wiederherstellen, jedoch kommt bei dieser Variante keine lokale Persistenz von Daten beim Client und Synchronisation mit dem Applicationserver zum Einsatz.

Bei **„echter“ Offlinefähigkeit** hingegen ist die Webanwendung auch ohne eine Internetverbindung funktionsfähig. Benutzer können nicht nur Daten anzeigen, sondern auch hinzufügen oder löschen. Dies wird durch eine lokale Zwischenspeicherung des benutzerspezifischen Anwendungsmodels („Application State“) und dessen Synchronisation mit dem Applicationserver im Onlinezustand erreicht. Der Benutzer kann so kontinuierlich mit der Webanwendung arbeiten, ohne darauf achten zu müssen, ob er mit dem Internet verbunden ist oder nicht.
Zur Erfüllung der in Anforderung [FA-3] (vgl. Abschnitt 3.1) beschriebenen Offlinefähigkeit wird diese Variante bevorzugt.

4.1.1 Caching statischer Ressourcen

Während Webanwendungen einen Fehler anzeigen und eine weitere Verwendung unmöglich machen, sobald der Benutzer ohne aktive Internetverbindung versucht zu einer Seite zu navigieren, ist es in nativen Apps möglich sich weiter innerhalb der Anwendung zu bewegen.
Eine progressive Webanwendung muss also die Möglichkeit bieten, zu erkennen, ob eine aktive Internetverbindung vorhanden ist oder nicht und entsprechend darauf reagieren. Hier kommt die Service Worker API ins Spiel. Diese Technologie bietet Entwicklern die Möglichkeit einer optimalen Bereitstellung von Offline-Benutzererfahrung.

Wie in Abschnitt 2.1 beschrieben handelt es sich beim Service Worker um eine Art Proxy zwischen der Webanwendung und dem Browser. Dadurch ist es möglich, Responses von HTTP-Request aufzunehmen, zu analysieren und ggf. anzupassen. Die Erkennung, ob ein Benutzer offline ist, ermöglicht

es unterschiedlich auf einen HTTP-Request zu reagieren. Dies ist eine Schlüsselfunktion, um die Anforderungen an Offlinefähigkeit erfüllen zu können.

Bezeichnung	**Beschreibung**
networkOnly	Ressourcen werden nur aus Netzwerk geholt
cacheOnly	Ressourcen werden immer aus Cache geladen
fastest	Versucht von beiden Quellen zu laden und Antwortet mit schnellerem Response
networkFirst	Versucht zuerst aus dem Netzwerk zu laden und schaut in den Cache, wenn dies fehlschlägt
cacheFirst	Bezieht Ressourcen direkt aus dem Cache, fragt jedoch auch beim Netzwerk nach und aktualisiert bei Erfolg die Ressourcen im Cache

Tabelle 4.1: Übersicht Caching Strategien

Tabelle 4.1 zeigt die fünf grundsätzlich möglichen Strategien für das Caching von statischen Ressourcen, die mit Hilfe des Service Workers umgesetzt werden können. Damit die Benutzung der Anwendung auch ohne aktive Internetverbindung gewährleistet ist, müssen die Ressourcen ebenfalls bereitstehen, wenn das Gerät offline ist. Der HTTP-Response muss also lokal gespeicherte Ressourcen ausliefern, wenn diese nicht über das Netzwerk bezogen werden können.

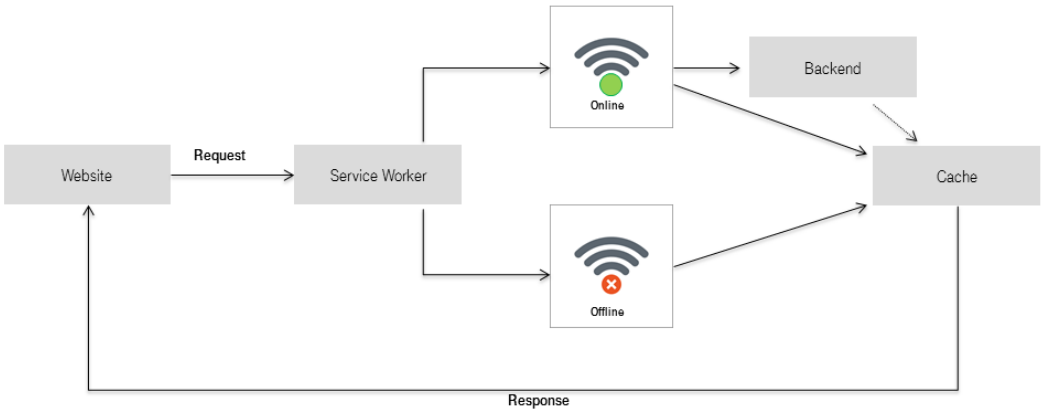

Abbildung 4.1: Caching Strategie `cacheFirst`

Das `cacheFirst`-Verfahren bietet sich für den vorliegenden Anwendungsfall an; die angeforderten Ressourcen werden direkt aus dem Cache geladen und anschließend versucht, diese im Hintergrund mit Ressourcen aus dem Internet zu aktualisieren (vgl. Abbildung 4.1). Dadurch wird die Webseite unabhängig vom Onlinezustand bei Anforderung schnell geladen und dem Benutzer so eine optimale Offline-Benutzererfahrung ermöglicht.

4.1.2 lokale persistente Speicherung und Synchronisation des Application State

Neben der in Abschnitt 4.1.1 beschriebenen Vorhaltung der statischen Ressourcen muss die progressive Webanwendung ebenfalls einen Mechanismus zur Verfügung stellen, um das Datenmodell im Offlinebetrieb oder bei schlechter Verbindung bereitzustellen und es bei aktiver Internetverbindung mit einem entfernten Server synchronisieren zu können.

Genau diese Anforderungen lassen sich mit der Background Sync API erfüllen. Diese reagiert auf Events, die ausgelöst werden, wenn eine aktive Interverbindung hergestellt wird.
Das `sync` Event wird u.a. ausgelöst, wenn die entsprechende Webseite geöffnet wird und wenn eine Netzwerkverbindung wiederhergestellt wird. Damit bietet sich diese Technologie an, um den Application State im Hintergrund zu aktualisieren und im Offlinebetrieb oder bei schlechter Verbindung in einer Warteschlange zu positionieren, die abgearbeitet wird.
Im Fall der schlechten Verbindung würde diese Abarbeitung im Hintergrund stattfinden, auch wenn diese mehr Zeit in Anspruch nimmt. Wenn keine Internetverbindung verfügbar ist, wird die Warteschlange abgearbeitet, sobald eine Verbindung möglich ist. In beiden Fällen kann der Benutzer weiter mit der Webanwendung interagieren.

4.2 Web Push

Die Push-API bietet Webanwendungen die Möglichkeit, von einem Server gesendete Nachrichten zu empfangen, unabhängig davon, ob die Webanwendung im Vordergrund ausgeführt oder aktuell überhaupt geöffnet ist.
Um Push-Nachrichten empfangen zu können, muss für die Webseite ein Service Worker registriert werden. Sobald dieser aktiv ist, können Push-Benachrichtigungen über den Push-Manager des Service Workers abonnieren werden (`PushManager.subscribe()`).

Die resultierende `PushSubscription` enthält alle Informationen, die die Anwendung benötigt, um eine Push-Nachricht zu senden. Neben einem Endpoint ebenfalls den für das Senden von Daten erforderlichen Verschlüsselungsschlüssel.

Der Service Worker wird nach Bedarf gestartet, um eingehende Push-Nachrichten zu behandeln, die an den `ServiceWorkerGlobalScope.onpush()`- Eventhandler übergeben werden. Dies ermöglicht es Webanwendungen, auf empfangene Push-Nachrichten, beispielsweise durch Anzeigen einer Benachrichtigung zu reagieren.
Für die Anzeige von Benachrichtigungen aus dem Service Worker heraus, wird laut Standard die Methode `ServiceWorkerRegistration.showNotification()` bereitgestellt.

Jedes Abonnement für einen Service Worker ist eindeutig. Der Endpoint für das Abonnement ist eine eindeutige URL. Die Kenntnis des Endpoints ist alles, was notwendig ist, um eine Nachricht an die Anwendung zu senden. Die Endpoint-URL muss daher geheim gehalten werden, oder andere Anwendungen könnten Push-Nachrichten an die Anwendung senden.

4.2.1 Ablauf

Abbildung 4.3 zeigt den grundsätzlichen Ablauf von Registrierung des Push-Managers, über die Übertragung der Endpointinformationen bis hin zum Versand von Push-Nachrichten.

Zuerst muss der Nutzer der Webanwendung auf eine Anforderung für Webbenachrichtigungen oder sonstige verwendete Berechtigungen reagieren, indem er die Berechtigungen erteilt (vgl. Abbildung 4.2).

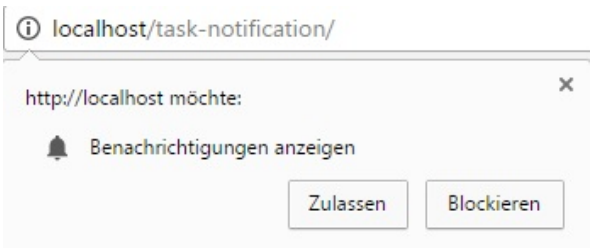

Abbildung 4.2: Chrome - Berechtigungen zum Anzeigen von Benachrichtigungen

Nachdem die Berechtigung erfolgt ist, wird der Service Worker, lokal für die Webanwendung registriert. Danach wird der Push-Messaging-Service, in unserem Fall „Firebase Cloud Messaging“ (kurz FCM) mit der Funktion `PushManager.subscribe()` abonniert (vgl. Abbildung 4.3 Punkt 1).

Mit Hilfe von `PushSubscription.endpoint` kann der mit der Subscription verknüpfte Endpoint abgerufen werden. Die Details werden an den Applicationserver gesendet, so dass er Push-Nachrichten senden kann, wenn dies erforderlich ist (vgl. Abbildung 4.3 Punkt 2). Die Subscription-ID wird aus dem kompletten Endpoint entnommen.

Auf Serverseite wird der Endpoint und alle erforderlichen Details, wie die Sender ID und Geräte ID in der Datenbank gespeichert, so dass sie verfügbar sind, wenn eine Push-Nachricht an einen Benutzer bzw. ein Endgerät gesendet werden soll.

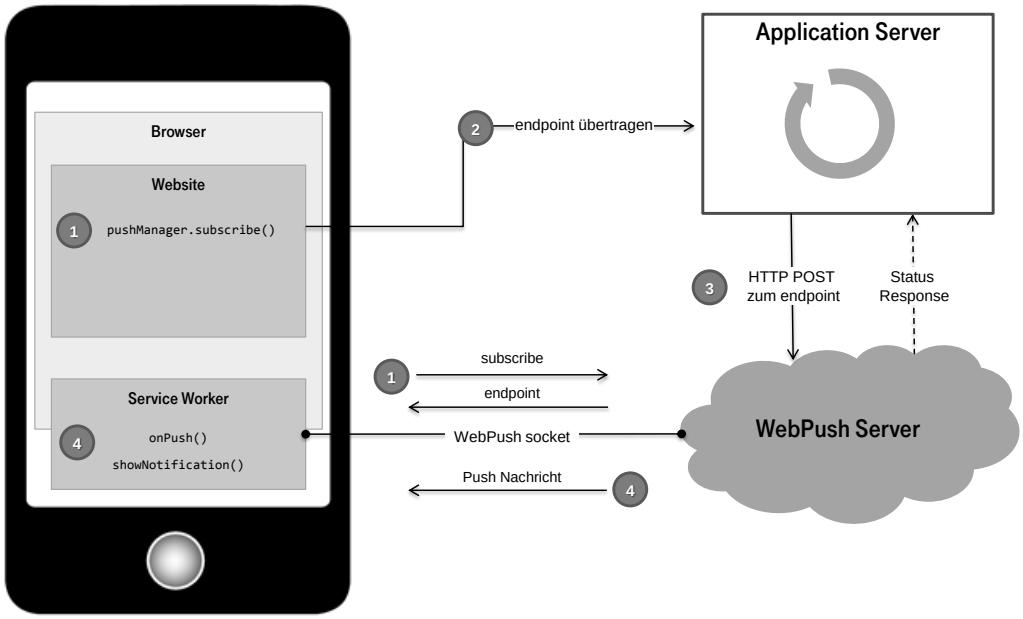

Abbildung 4.3: Push mittels Serviceworker (in Anlehnung an MozillaWiki)
Quelle: `https://wiki.mozilla.org/File:PushNotificationsHighLevel.png`

Um eine Push-Nachricht zu senden, muss ein HTTP-POST an die Endpoint-URL (vgl. Abbildung 4.3 Punkt 3) gesendet werden. Die Anforderung muss einen TTL-Header enthalten, der begrenzt wie lange die Nachricht in der Warteschlange stehen soll, wenn der Benutzer nicht online ist.

Sobald die Push-Nachricht vom Web Push Server erfolgreich versendet wurde, antwortet dieser mit einem Response, welcher eine eindeutige Message ID enthält. Diese referenziert auf eine bestimmte Push-Benachrichtigung.

Den vom Applicationserver zuvor generierten Nutzdaten (Payload) wird diese Message ID zugeordnet und für Clientanfragen vorgehalten.
Sobald eine Push-Nachricht vom Client empfangen wird, löst dies ein `onPush`-Event aus (vgl. Abbildung 4.3 Punkt 4). Der Event-Listener reagiert mit einer direkten Anfrage beim Applicationserver und fragt ggf. vorhandenen Payload für die aktuelle Message ID ab.

Alternativ ist es möglich, den Payload im `body` der Push-Nachricht zu übertragen. Dabei ist es zwingend erforderlich, dass die Nutzdaten verschlüsselt übertragen werden. Ansonsten würde seitens des Push-Messaging-Servers der Payload auf `NULL` gesetzt werden.
Im Rahmen dieses Projekts wurde entschieden, dass keine Nutzdaten über „externe“ Server geleitet werden sollen. Daher kommt diese Variante hier nicht zum Einsatz. Stattdessen wird der Payload aktiv vom Client beim Applicationserver abgefragt, sobald ein Push-Event ausgelöst wird.

4.3 Architekturbeschreibung

Die Anwendung beruht auf dem Client-Server-Prinzip. Dabei stellt der Client lediglich die Oberfläche zur Interaktion mit dem Anwender dar. Außer der notwendigen UI- und Service Worker-Logik ist die gesamte Geschäftslogik auf einen Business-Server (Applicationserver) ausgelagert. Die zentrale Datenbank sowie die statischen Ressourcen zur Darstellung des Client werden ebenfalls vom Applicationserver bereitgestellt. Für die Kommunikation steht eine RESTful-Schnittstelle zur Verfügung.

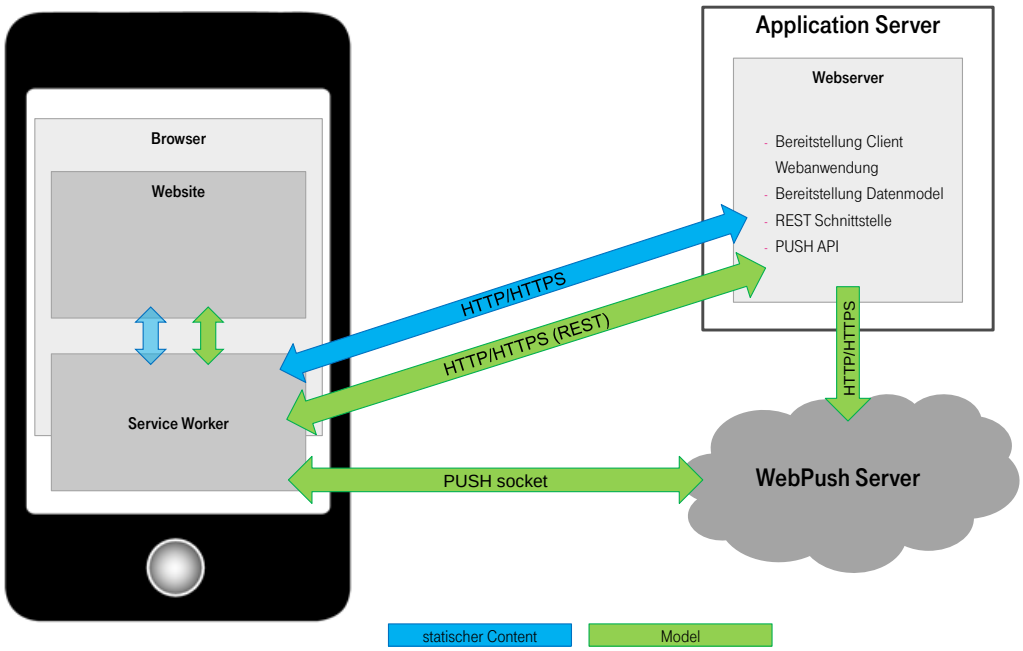

Abbildung 4.4: Archtikturbeschreibung - Umsetzung mit Serviceworker

4.4 Applicationserver

Für die Bereitstellung der Geschäftslogik wird ein eigener Applicationserver benötigt. Als Plattform soll Node.js eingesetzt werden. Dadurch ist es mit überschaubarem Aufwand möglich, einfache Webanwendungen zu erstellen. Die Implementierung einer RESTful-Schnittstelle ist ebenso problemlos möglich wie die Anbindung von ORM-Tools für die Kommunikation mit einer Datenbank.

4.4.1 Datenbank

Der Applicationserver stellt ebenfalls die Datenbank zur Verfügung und Verwaltet deren Zugriffe. Als Datenbank soll eine noSQL-Datenbank-Technologie verwendet werden. Diese ermöglicht eine objektorientierte Speicherung der Daten bei maximaler Flexibilität des Schemas. Node.js unterstützt die Anbindung sowohl von MongoDB als auch CouchDB. In der Implementierung wird MongoDB verwendet werden.

4.4.2 REST-API

Zur Bereitstellung von CRUD-Funktionalitäten über standardisierte HTTP-Methoden (vgl. Abschnitt 3.1, funktionale Anforderung [FA-4]) wird dem Applicationserver eine RESTful-Schnittstelle hinzugefügt. Eine Übersicht über mögliche API-Routen mit entsprechender HTTP-Methode ist in Tabelle 4.2 dargestellt.

Route	**HTTP-Methode**	**Beschreibung**
/api/signup	POST	Registriert einen neuen Benutzer
/api/authenticate	POST	Authentifiziert einen Benutzer
/api/tasks	GET	Gibt alle Aufgaben zurück
/api/tasks	POST	Legt eine neue Aufgabe an
/api/tasks/:taskId	GET	Gibt eine einzelne Aufgabe zurück
/api/tasks/:taskId	PUT	Aktualisiert eine einzelne Aufgabe
/api/tasks/:taskId	DELETE	Löscht eine einzelne Aufgabe
/push/devices	GET	Gibt alle registrierten Geräte zurück
/push/devices	POST	Registriert ein neues Gerät
/push/payload/:messageId	GET	Gibt den Payload für MessageId zurück

Tabelle 4.2: Übersicht API Routen

Authentifizierung und Autorisierung

Für den Zugriff auf die CRUD-Methoden ist eine Benutzerauthentifizierung und Autorisierung notwendig. Registrierte Benutzer authentifizieren sich mittels Benutzername und Passwort über die Route `/api/authenticate` am Applicationserver.
Die notwendigen Parameter müssen im Request-Body übertragen werden. Bei erfolgreicher Authentifizierung antwortet der Server mit einem Token innerhalb des Response-Body. Allen weiteren Requests wird der Security-Token als `Authorization`-Header oder Body-Parameter hinzugefügt.
Wird eine REST-Route ohne Authorization aufgerufen, antwortet der Server mit einem `HTTP 401`-Response und signalisiert damit, dass eine Authentifizierung erforderlich ist.

4.5 Datenmodel

Die Umsetzung der Beispielanwendung benötigt kein komplexes Datenbankmodell. Abbildung 4.5 zeigt den Entwurf des Datenmodells. Dabei wurden nur notwendige Attribute dargestellt, um die Übersichtlichkeit in der Dokumentation gewährleisten zu können.

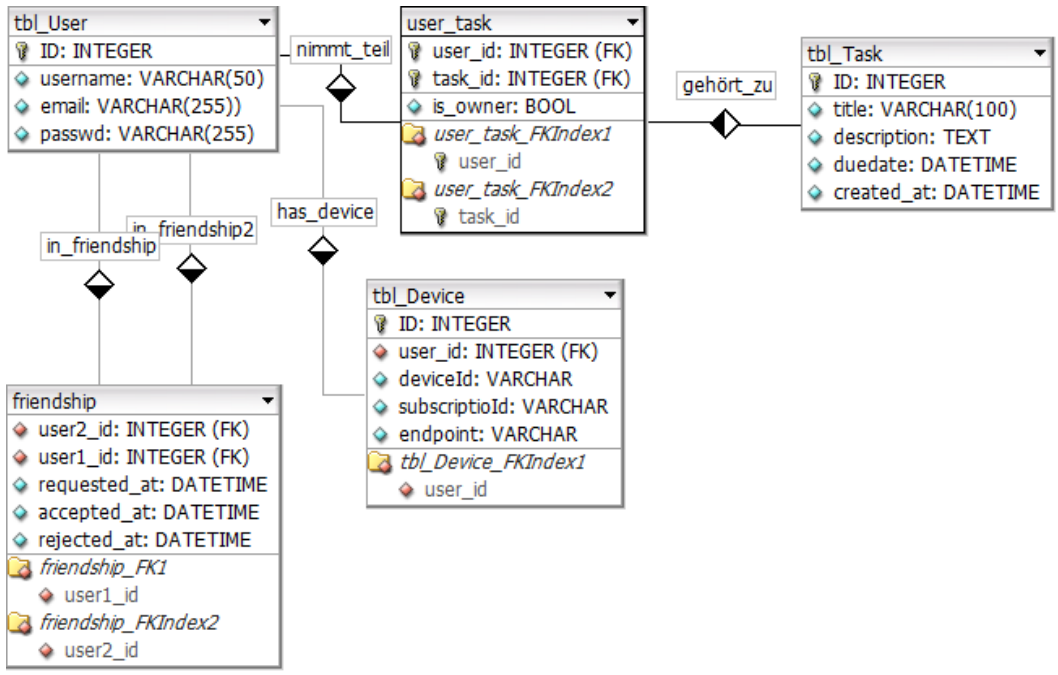

Abbildung 4.5: Datenmodell

Zur Erfüllung der, in Abschnitt 3.2 beschriebenen, nicht-funktionalen Anforderungen ist eine zentrale Benutzer-Entity notwendig. Diese enthält neben den notwendigen Anmeldeinformationen (Benutzername und Passwort) für jeden registrierten Benutzer eine E-Mail Adresse. Diese wird beispielsweise benötigt, um die Registrierung zu bestätigen oder um ein vergessenes Passwort zurückzusetzten.

Jeder Benutzer kann mehrere Aufgaben anlegen. Zu diesen Aufgaben können ein Titel, Beschreibung sowie ein Datum hinterlegt werden. Weiterhin können weitere Benutzer zu einer Aufgabe hinzugefügt werden. Dabei ist ein Benutzer immer „Eigentümer“ während die anderen Benutzer als „Teilnehmer“ agieren.

Zur Abbildung von Freundschaftsbeziehungen ist eine n:m-Beziehung zwischen User und User notwendig. Zu dieser Beziehung werden noch weitere Attribute wie „Anfrage gestellt am“, „bestätigt am“ und „abgelehnt am“ hinzugefügt.

Damit Benutzer unabhängig vom verwendeten Endgerät über Push-Nachrichten benachrichtigt werden können, muss für einen Benutzer mindesten ein Gerät mit zugehöriger Push-Subscription-ID und Endpoint angelegt werden. Somit ist es möglich, einem Benutzer auf verschiedenen Endgeräten die gleichen Push-Benachrichtigungen anzuzeigen.

4.6 Client-Oberfläche

Um das „Look & Feel“ einer nativen Android-Applikation zu erreichen wird nach Anforderung [NFA-1] (vgl. Abschnitt 3.2) das Framework „JQuery-Mobile“ verwendet.
Von Haus aus wird kein „Material Design“ mitgeliefert. Das UI-Framework **nativeDroid2** setzt auf „JQuery-Mobile“ auf und bietet ein „Material Design“-Theme, sowie Hilfsklassen und Funktionen, die es erleichtern ein natives „Look & Feel“ in Webanwendungen zu gewährleisten.

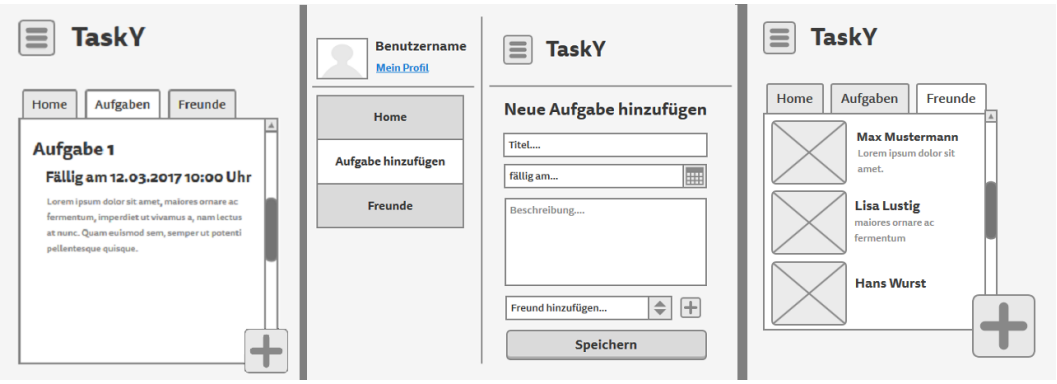

Abbildung 4.6: Wireframe/Mockup-Entwurf grafische Benutzeroberfläche

Neben Ansichten für die Benutzerregistrierung und Anmeldung benötigt die Webseite Views für die Anzeige von Listen, eine Detailansicht für Listenelemente sowie ein Formular zum anlegen von neuen Aufgaben.

Für die Navigation innerhalb der Webseite kommt eine vertikale Navigation zum Einsatz, die wie in nativen Applikationen ggf. ein- bzw. ausgeblendet werden kann. Auf der Startseite ermöglicht eine horizontale Tab-Navigation die Anzeige von verschiedenen Liste. Neben aktuellen Nachrichten wird eine Liste aller Aufgaben, sowie eine Kontaktliste integriert (vgl. Abbildung 4.6).

5 Implementierung

Das folgende Kapitel beschreibt die Umsetzung der im vorherigen Kapitel beschriebenen Konzeption. Dabei wird sich sowohl bei der serverseitigen als auch clientseitigen Implementierung auf die notwendigen Schritte konzentriert, die für die Verwendung der Service Worker Technologie und die Erfüllung der Projektaufgabe erforderlich sind. Die Umsetzung der grafischen Benutzeroberfläche, Installation von MongoDB, sowie die Einrichtung der IndexedDB sind damit nicht Bestandteil dieser Dokumentation.

5.1 Applicationserver

5.1.1 Node.js Express-Server

Neben einer Datenbank bietet der Applicationserver die Funktionalitäten eines Webservers.
Als Backend-Plattform wird Node.js eingesetzt und mit Hilfe von `Node Express` ist in wenigen Schritten ein vollfunktionsfähiger Webserver installiert und eingerichtet.

```
{
  "name": "node-api",
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "bcrypt-nodejs": "0.0.3",
    "body-parser": "~1.15.2",
    "cookie-parser": "~1.4.3",
    "debug": "~2.2.0",
    "express": "~4.14.0",
    "jade": "~1.11.0",
    "jsonwebtoken": "^7.1.9",
    "mongodb": "^2.2.11",
    "mongoose": "~3.6.13",
    "morgan": "~1.7.0",
    "node-gcm": "^0.14.0",
    "passport": "^0.3.2",
    "passport-jwt": "^2.2.0"
  }
}
```

Listing 5.1: package.json - notwendige Node.js Pakete

Die Bereitstellung der notwendigen Funktionalitäten setzt einige `node-packages` voraus, die der `package.json`-Datei hinzugefügt werden (vgl. Listing 5.1).

Wir nutzen „PassportJS“ als Middleware zur Authentifizierung für Node.js und das „JSON Web Token“-Prinzip für die Generierung von Authentifizierungstoken. Weiterhin werden Pakete für die Verschlüsselung von Passwörtern sowie für die Anbindung von MongoDB und `morgan` für das Request-Logging installiert.

Die Umsetzung des Datenbankschemas in MongoDB sowie das objektrelationale Mapping (ORM) erfolgt mittels des `node-package mongoose`. Lisiting 7.2 im Anhang 7.1.1 zeigt beispielhaft die Einrichtung der Benutzer-Entity für die Benutzerauthentifikation.

Eine genaue Beschreibung der Einrichtung einer Node.js-Anwendung ist nicht Bestandteil dieser Dokumentation und soll nicht weiter beleuchtet werden. An dieser Stelle sei an zahlreiche Anleitungen im Internet verwiesen.

5.1.2 Datenbank-Server anbinden

Für die persistente Speicherung der Daten auf Serverseite wird ein MongoDB-Server aufgesetzt. Die genaue Installation und Einrichtung ist nicht Bestandteil dieser Dokumentation.
Nach erfolgreicher Installation muss der Applicationserver so konfiguriert werden, dass er sich mit der Datenbank verbinden und Anfragen stellen kann (vgl. Listing 5.2).

```
1 // app.js
2 mongoose.connect(config.database);
3
4 // config/database.js
5 module.exports = {
6    'secret': 'thisIsNotASecretKeyChangePlease!',
7    'database': 'mongodb://localhost:27017/tasky'
8 };
```

Listing 5.2: Verbindung zur Datenbank konfigurieren

5.1.3 REST-Schnittstelle

Die in Abschnitt 4.4.2 geplante RESTful-Schnittstelle wird im Node.js Express-Server umgesetzt. Jeder Request, der sich direkt auf die Anforderung von benutzerspezifischen Ressourcen bezieht, erfordert einen gültigen Authentifizierungstoken (vgl. Anhang 7.1.2). Es ist möglich aus dem Token den Benutzer abzuleiten und aus der Datenbank abzufragen. Wenn ein gültiger Benutzer gefunden wurde, kann das serialisierte Objekt zum eigentlichen Request hinzugefügt und die Anforderung an die entsprechende Route weitergegeben werden („Router-Middleware“).
Damit kann innerhalb der folgenden Routing-Methoden direkt auf die User-Entity zugegriffen und

damit nur die Daten zu übermitteln werden, die den anfragenden Benutzer zugeordnet sind. Listing 5.3 verdeutlicht dies beispielhaft an den `/api/tasks`-Routen. In Zeile 9 bzw. Zeile 26 wird auf die User-Entity zugegriffen, welche zuvor serverseitig aus dem JWT-Token ausgelesen, aus der Datenbank abgefragt und dem Request hinzugefügt wurde.

```
1  // routes/api.js
2  ...
3  router.route('/tasks')
4     // create a task (accessed from POST /api/tasks)
5     .post(function(req, res){
6        // create a new instance of Task model
7        var task = new Task();
8        // get user from request
9        var user = req.user;
10       // set description from the request
11       task.description = req.body.description;
12       // set user
13       task.user = user._id;
14
15       // save task and check for error
16       task.save(function(err){
17          if(err)
18             res.send(err);
19          res.json({ message: 'Task created!'});
20       });
21    })
22
23    // get all tasks (accessed from GET /api/tasks)
24    .get(function(req, res){
25       var user = req.user;
26       Task.find({ user: user._id }, function (err, tasks) {
27          if(err)
28             res.send(err);
29          res.json(tasks);
30       });
31    });
32 ...
```

Listing 5.3: Verbindung zur Datenbank konfigurieren

Test der REST-API

Um die Funktionalitäten der REST-Schnittstelle zu überprüfen und zu untersuchen wird Postman[1] verwendet. Dieses Tool ermöglicht es in einer grafischen Oberfläche Requests an eine URL zusammenzustellen und die entsprechenden Responses auszuwerten.

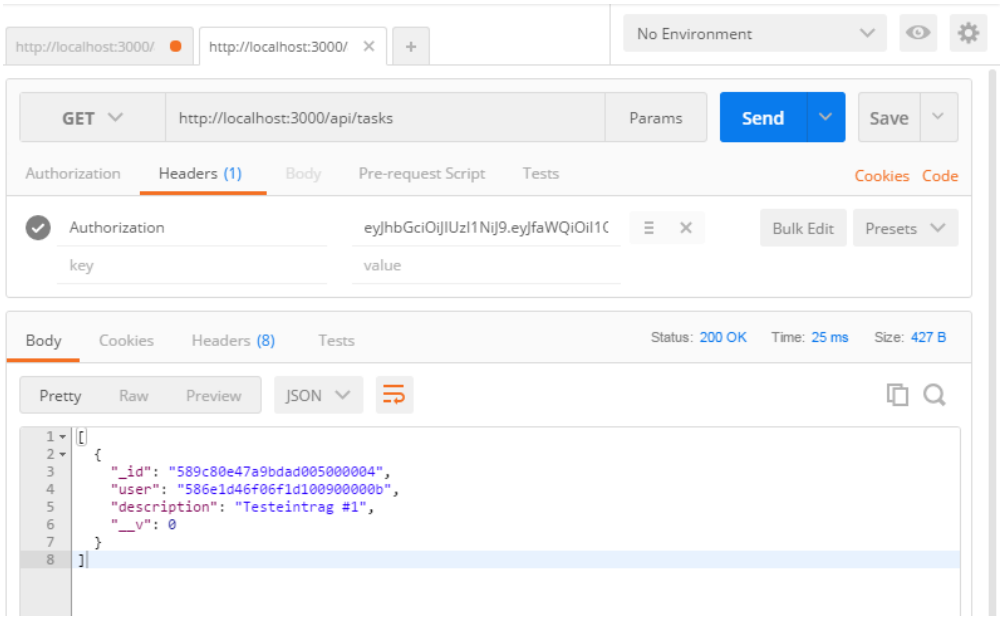

Abbildung 5.1: Postman: „Auflistung aller Aufgaben eines authentifizierten Benutzers“

[1] *https://www.getpostman.com/*

5.1.4 Cross-Origin Resource Sharing

Wenn eine Webseite von einer anderen Domain, als die des Servers, Requests stellt, unterbindet normalerweise die Same-Origin-Policy (SOP) solche Zugriffe. Mittels Cross-Origin Resource Sharing (CORS) ist es möglich, für bestimmte Domains derartige Anfragen an einen Server zu erlauben.

Wir nutzen eine weitere Router-Middleware, um allen Responses unseres Applicationsservers die in Listing 5.4 aufgeführten `Access-Control`-Header hinzuzufügen. Für die Entwicklungsumgebung ist es vertretbar, sämtlichen Hosts (`(*)`)den Zugriff zu erlauben. In einer Produktivumgebung würde man ausschließlich die Client-Webserver-Domain berechtigen.

```
1 router.use(function(req, res, next) {
2    res.header("Access-Control-Allow-Origin", "*");
3    res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With,
       Content-Type, Accept");
4    next(); // go to the next routes and don't stop here
5 });
```

Listing 5.4: Router-Middleware um jedem Response die `Access-Control`-Header hinzuzufügen

5.2 Service Worker

An dieser Stelle sei besonders erwähnt, dass die Service Worker API eine Verbindung über `https` erfordert. Für diese Verbindung muss ebenfalls ein gültiges Serverzertifikat vorliegen. Einzig in der Entwicklung im Umfeld von `localhost` ist eine unverschlüsselte Kommunikation über `http` möglich.

5.2.1 Installation

Um einen Service Worker zu installieren, muss dieser zuerst für die entsprechende Webseite registriert werden. Listing 5.5 zeigt, wie der Service Worker registriert wird.
Zuerst wird überprüft, ob der aktuelle Browser die Service Worker API unterstützt. Um den aktuellen Zustand des Service Workers beim Laden der Seite nachvollziehen zu können, wird dieser in die Debugging-Konsole des Browsers geschrieben.
Die `serviceWorker.register()`-Methode erwartet als Parameter die Service Worker Konfigurationsdatei. Diese enthält `Listener`, die auf verschiedene `Events` reagieren und die eigentliche Funktionalitäten bereitstellen. Weiterhin kann dem Service Worker ein `scope` übergeben werden. Dieser definiert den Context, für den die Service Worker Registration gültig ist.
Es ist also grundsätzlich möglich, mehrere Service Worker für eine Webanwendung zu registrieren. Jeder `scope` benötigt dabei eine eigene Konfigurationsdatei (diese muss sich direkt unter dem `scope`-Pfad befinden). Wird kein `scope` explizit angegeben, wird dieser aus dem Pfad der Konfigurationsdatei abgeleitet.

```
1  // js/app.js
2  if ('serviceWorker' in navigator)
3  {
4     navigator.serviceWorker.register('sw.js').then(function(reg)
5     {
6        if(reg.installing)
7           console.log('Service worker installing');
8        else if(reg.waiting)
9           console.log('Service worker installed');
10       else if(reg.active)
11          console.log('Service worker active');
12    }).catch(function(error)
13    {
14       // registration failed
15       console.log('Registration failed with ' + error);
16    });
17 }
```

Listing 5.5: Einrichtung Service Worker

5.2.2 Caching der statischen Ressourcen

Innerhalb der Service Worker Konfigurationsdatei wird unter anderem festgelegt, welche Ressourcen zwischengespeichert werden sollen, welche Caching-Strategie vom Service Worker verfolgt wird und wie dieser Requests und eingehenden Responses verarbeiten soll.

Listing 5.6 zeigt, wie die Ressourcen festzulegen sind, die im Cache vorgehalten werden. Wenn der Service Worker erfolgreich registriert wurde, wird das `install`-Event ausgelöst. Für die Übersicht wurde die Liste der Ressourcen des Cache in das Array `urlToCache` ausgelagert. Dieses ist im Anhang 7.2.1 vollständig aufgeführt.

```
1  // sw.js
2  this.addEventListener('install', function(event) {
3     console.log('The service worker is being installed.');
4     event.waitUntil( caches.open(CACHE_NAME)
5        .then(function(cache) {
6           return cache.addAll(urlsToCache);
7        })
8        .then(function(){
9           return self.skipWaiting();
10       })
11    );
12 });
```

Listing 5.6: Ressourcen festlegen, die im Zwischenspeicher vorgehalten werden sollen

Die Integration des JavaScript-`Promise`-Konzepts ermöglicht es dem Service Worker im Hintergrund zu arbeiten und Anfragen asynchron zu bearbeiten. Es wird oft mit einer Verkettung von `Promises` gearbeitet, um auf den Abschluss einer Operation zu warten und anschließend weitere Operationen durchzuführen. Dabei kommen ebenfalls `callback`-Funktionen zum Einsatz, die das Ergebnis einer vorangegangenen Operation enthalten und selbst wieder einen `Promise` darstellen.

Nachdem der Service Worker erfolgreich registriert und die statischen Ressourcen für das Caching eingerichtet sind, wird die Caching-Strategie aus Abschnitt 4.1.1 umgesetzt. Wenn ein Request von der Webseite gestellt wird, empfängt der Service Worker `fetch`-Events. Listing 5.7 zeigt wie die Caching-Strategie durch Verarbeitung möglicher Responses umgesetzt werden kann.

```
// sw.js
this.addEventListener('fetch', function(event) {
   console.log('The service worker is serving the asset.');

   event.respondWith(
     // try to find cached resource
     caches.match(event.request).catch(function() {
       // fetch network request
       return fetch(event.request);
     }), function() { // Not fired due to the catch
       // fetch network request
       return fetch(event.request).then(function(response) {
         // save resource to cache
         return caches.open(CACHE_NAME).then(function(cache) {
           cache.put(event.request, response.clone());
           return response;
         });
       });
     })
   );
});
```

Listing 5.7: Verarbeitung empfangener Requests und Auswertung möglicher Responses im Service Worker

5.2.3 Push-Nachrichten abonnieren

Damit eine Webanwendung Push-Nachrichten erhalten und verarbeiten kann, benötigt sie einen aktiven Service-Worker. Dieser kann sich mittels `PushManager.subscribe()` für Push-Nachrichten anmelden. Der in Abschnitt 4.3 beschriebene Ablauf wurde wie geplant umgesetzt.

```
// resources/js/app.js
navigator.serviceWorker.ready.then(function(serviceWorkerRegistration) {
   serviceWorkerRegistration.pushManager.getSubscription().then(
      function(pushSubscription) {
         if(pushSubscription)
         {
            console.log("Push Subscription exists");
            // send subscription to application server
            sendSub(pushSubscription);
         }
         else
         {
            console.log("Push Subscription not existing");
            // create subscription
            subscribePush();
         }
      }.bind(this)).catch(function(e) {
      console.error('Error getting subscription', e);
   });
});
```

Listing 5.8: Service Worker: Anmeldung am Push-Server

Nachdem geprüft wurde, ob für den aktiven Service Worker bereits eine `PushSubscription` vorhanden ist (siehe Listing 5.8), wird entweder die `PushSubscription` an den Applicationserver übertragen (`sendSub()`-Funktion) oder eine Anmeldung beim Push-Service innerhalb der Methode `subscribePush()`) initiiert. Nach erfolgreicher Anmeldung mittels `PushManager.subscribe()` wird die zurückgegebene `PushSubscription` mit Hilfe der Funktion `sendSub()` an den Applicationserver übertragen. Im Anhang 7.3 sind die Funktionen für die Anmeldung, Abmeldung und Übertragung der `PushSubscription` aufgeführt.

Endpoint auf Applicationserver speichern

Die im Anhang 7.3 aufgeführte `sendSub()`-Funktion ist dafür verantwortlich, die Endpoint-Informationen auf den Applicationserver zu übertragen.
Neben dem Endpoint und der Subscription-ID wird eine eindeutige `DeviceID` sowie ein optionaler Gerätename übertragen.

Der Applicationserver speichert diese Informationen in der Datenbank bzw. aktualisiert einen bereits vorhandenen `Device`-Eintrag für den entsprechenden Benutzer und der `DeviceID`. Die `DeviceID` wird mit Hilfe der `Fingerprint2`-Bibliothek[2] generiert und ist für jedes Gerät und Browser eindeutig.

```
1  // CHECK if unique device id exists
2  $(document).ready(function(){
3
4     // store device id
5     var deviceId = localStorage.getItem("deviceId");
6     if(deviceId === null)
7     {
8        new Fingerprint2().get(function(result, components){
9           localStorage.setItem("deviceId", result);
10          console.log("Set deviceId: " + result); //a hash, representing your
                device fingerprint
11       });
12    }
13    console.log("DeviceID: " + deviceId);
14 });
```

Listing 5.9: Bei Seitenstart prüfen, ob eine DeviceID vorhanden ist und ggf. anlegen

Verarbeitung von `Push`-Events

Wenn der Service Worker eine Push-Nachricht empfängt, wird das Event `push` ausgelöst. In der Service Worker Konfigurationsdatei wird dafür ein `EventListener` registriert. Listing 5.10 zeigt, wie bei eintreffenden Push-Benachrichtigungen beim Applicationserver der zugehörige Payload angefragt wird.
Wenn der empfangene Payload den Type `update` trägt, bedeutet dies, dass sich im `body` Aktualisierungen des Models verbergen. Daraufhin wird die Funktion `updateIndexedDb()` aufgerufen. Als Parameter erwartet diese ein mehrdimensionales Array aus serialisierten Objekten. Diese sind mit einem Timestamp versehen und werden ggf. in der lokalen Datenbank aktualisiert.

[2] *https://github.com/Valve/fingerprintjs2*

```
// sw.js
self.addEventListener('push', function(event) {
   console.log('Push message', event);

   event.waitUntil(getEndpoint().then(function(endpoint){
      var subId = endpoint.split("/").pop();
      var request = new Request(PUSH_URL + "/payload/"+subId, {
         method: 'GET',
         mode: 'cors',
         redirect: 'follow'
      });
      return fetch(request);
   }).then(function(res) {
      res.json().then(function(data){
         if(data.type == "update")
         {
            updateIndexedDb(data.body);
         }
         else
         {
            // Show notification
            self.registration.showNotification(data.title, {
               'body': data.body,
               'icon': data.icon
            });
         }
         console.log(data);
      })
   });
});
```

Listing 5.10: Service Worker `push`-EventListener

In dem Fall, dass der Payload nicht den Typ `update` trägt, wird davon ausgegangen, dass eine `Push-Notification` angezeigt werden soll. Der Payload enthält die notwendigen Attribute, um eine Benachrichtigung anzuzeigen (vgl. Abbildung 5.2). Benachrichtigungen werden sowohl auf mobilen Endgeräten als auch auf Desktop-PC's angezeigt.

Abbildung 5.2: Beispiel Anzeige einer Desktop-Benachrichtigung

5.2.4 Background Sync

Ähnlich der Push-Benachrichtigung verwendet Background Sync den Service Worker als Event-Target. Dadurch können Synchronisationen durchgeführt werden, obwohl die betroffene Webseite nicht geöffnet ist oder gerade keine aktive Internetverbindung besteht. Um diese Funktionalität verwenden zu können, muss ein `sync` beim Service Worker registriert werden.

```
// js/app.js
if ('serviceWorker' in navigator && 'SyncManager' in window)
{
  navigator.serviceWorker.ready.then(function(reg) {
    return reg.sync.register('appStateSync');
  }).catch(function() {
    // system was unable to register for a sync,
    // this could be an OS-level restriction
    postDataToServer();
  });
}
else
{
  // serviceworker/sync not supported
  postDataToServer();
}
```

Listing 5.11: Registrierung Background Sync

Da nicht alle Browser die Background-Sync-Funktionalitäten unterstützen, jedoch Service Worker teilweise integriert haben, bietet es sich an eine erweiterte Prüfung durchzuführen. Listing 5.11 zeigt, wie neben der Service Worker Unterstützung auf Background Sync Funktionalität geprüft wird.
Die Funktion `postDataToServer()` bildet den Fallback, falls Background Sync nicht unterstützt wird, und überträgt die Zustandsdaten über „klassische“ Verfahren (vor Background Sync) mittels AJAX-Requests.

```
// sw.js
this.addEventListener('sync', function(event)
{
  if (event.tag == 'appStateSync')
    event.waitUntil(syncLocalDatabase());
});
```

Listing 5.12: `sync`-EventListener in der Service Worker Konfigurationsdatei

Die Funktion `syncLocalDatabase()` synchronisiert die locale IndexedDB mit der Datenbank des Applicationserver. Dabei werden ausschließlich Daten lokal im Client gespeichert, die für den Benutzer relevant sind. Die Funktion gibt einen `Promise` zurück, so dass die Synchronisation asynchron

im Hintergrund ausgeführt werden kann.
Bei jedem Sync wird der aktuelle Timestamp in einem gesonderten `STORE` (dieser enthält ebenfalls weitere Konfigurationsparameter) gespeichert und nur diejenigen Daten vom Applicationserver angefordert, die sich seit der letzten Aktualisierung geändert haben. Anschließend werden diese Objekte durchlaufen und der Timestamp der letzten Änderung mit der letzten Änderung des lokal hinterlegten Objekts verglichen und ggf. wird der Eintrag aktualisiert.

Schlägt die Synchronisation fehl, wird diese automatisch vom Background Sync Dienst wiederholt (durch erneuten auslösen eines `sync`-Events).

6 Ausblick

Offlinefähigkeit ist ein sehr interessantes Feature, das die Benutzererfahrung mit mobilen Anwendungen erheblich verbessern kann. Wenn der Anwender ständigen Verbindungsabbrüchen ausgesetzt ist, scheint diese Funktionalität ein unabdingbares Element der modernen Entwicklung von mobilen Apps zu sein.

Für die Art der Umsetzung muss man sich entsprechend Gedanken machen, ob die einfachste Variante reicht oder ob das Businessszenario die echte Offlinefähigkeit notwendig macht. Der Fokus muss hierbei auf der Erkennung der Bedürfnisse des Kunden liegen, um die passende Variante zu wählen.

Die Service Worker API zielt genau auf die Lücke, die zwischen mobile Web Apps und nativen Apps existiert. Besonders die Bereitstellung von Konzepten und Methoden zur Unterstützung einer Offlinefähigkeit, Abarbeitung von Hintergrundaktivitäten und die Möglichkeit der Verwendung Push-Nachrichten im inaktiven Modus stellen einen großen Schritt bei der Annäherung von mobile Webseiten an native Apps dar.

Es ist zu erwarten, dass sich die Service Worker Technologie weiter durchsetzen wird und damit weiter verbreitet. Besonders die Unterstützung mobiler Browser wird dabei im Mittelpunkt stehen.

Abbildungsverzeichnis

4.1 Caching Strategie `cacheFirst` 7
4.2 Chrome - Berechtigungen zum Anzeigen von Benachrichtigungen 9
4.3 Push mittels Serviceworker (in Anlehnung an MozillaWiki) 10
4.4 Archtikturbeschreibung - Umsetzung mit Serviceworker 11
4.5 Datenmodell 13
4.6 Wireframe/Mockup-Entwurf grafische Benutzeroberfläche 14

5.1 Postman: „Auflistung aller Aufgaben eines authentifizierten Benutzers“ 18
5.2 Beispiel Anzeige einer Desktop-Benachrichtigung 25

7 Anhang

7.1 Applicationserver

7.1.1 mongoos-Schema User-Entity

```
/**
 * USER Entity
 */

// load ORM
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var bcrypt = require('bcrypt-nodejs');

// create user schema
var UserSchema = new Schema({
   name: String,
   username: {
      type: String,
      unique: true,
      required: true
   },
   email: {
      type:String,
      unique: true,
      required: true
   },
   password: {
      type: String,
      required: true
   },
   admin: Boolean,
   created_at: Date,
   updated_at: Date
});

// on every save set updated_at and salt password
UserSchema.pre('save', function (next) {
   var user = this;
   var currentDate = new Date();

   // set updated_at date
   user.updated_at = currentDate;
   // if created_at doesn't exist, add to field
```

```
40    if(!user.created_at)
41       user.created_at = currentDate;
42
43    // save salted password to database if modified
44    if (this.isModified('password') || this.isNew) {
45       bcrypt.genSalt(10, function (err, salt) {
46          if (err) {
47             return next(err);
48          }
49          bcrypt.hash(user.password, salt, null, function (err, hash) {
50             if (err) {
51                return next(err);
52             }
53             user.password = hash;
54             next();
55          });
56       });
57    } else {
58       return next();
59    }
60 });
61
62 // compare salted passwords
63 UserSchema.methods.comparePassword = function (passwd, cb) {
64    bcrypt.compare(passwd, this.password, function (err, isMatch) {
65       if (err) {
66          return cb(err);
67       }
68       cb(null, isMatch);
69    });
70 };
71
72 module.exports = mongoose.model('User', UserSchema);
```

Listing 7.1: Mongoose Schema der User-Entity

7.1.2 Router-Middleware zur absicherung der API

```
// routes/api.js

/**
 * SECURITY: middleware to protect API
 *********************************************/
getToken = function (headers) {
   if (headers && headers.authorization) {
      return headers.authorization;
      var parted = headers.authorization.split(' ');
      if (parted.length === 2) {
         return parted[1];
      } else {
         return null;
      }
   } else {
      return null;
   }
};

router.use(function(req, res, next) {
   console.log(req.headers);
   // check header or url parameters or post parameters for token
   var token = req.body.token || req.query.token ||
       req.headers['x-access-token'] || getToken(req.headers);

   // decode token
   if (token) {
      // verifies secret and checks exp
      jwt.verify(token, config.secret, function(err, decoded) {
         if (err) {
            return res.status(403).json({ success: false, message: 'Failed to
                authenticate token.' });
         } else {
            // if everything is good, save to request for use in other routes
            User.findOne({
               username: decoded.username
            }, function(err, user) {
               if (err) throw err;

               if (!user) {
                  return res.status(403).send({success: false, message:
                      'Authentication failed. User not found.'});
               } else {
                  req.user = user;
                  next();
               }
            });
         }
```

```
46          });
47      } else {
48          // if there is no token return an error
49          return res.status(403).send({
50              success: false,
51              message: 'No token provided.'
52          });
53      }
54  });
```

Listing 7.2: Mongoose Schema der User-Entity

7.2 Service Worker Konfiguration

7.2.1 Ressourcen für Caching festlegen

```
var urlsToCache = [
   './vendor/natividroid2/css/natividroid2.color.blue-grey.css',
   './vendor/natividroid2/css/natividroid2.color.teal.css',
   './vendor/natividroid2/css/flexboxgrid.min.css',
   './vendor/natividroid2/css/material-design-iconic-font.min.css',
   './vendor/natividroid2/fonts/Material-Design-Iconic-Font.eot',
   './vendor/natividroid2/fonts/Material-Design-Iconic-Font.svg',
   './vendor/natividroid2/fonts/Material-Design-Iconic-Font.ttf',
   './vendor/natividroid2/fonts/Material-Design-Iconic-Font.woff',
   './vendor/natividroid2/js/natividroid2.js',
   './vendor/fingerprint2js/fingerprint2.js',

   './vendor/font-awesome/css/font-awesome.min.css',
   './vendor/font-awesome/fonts/FontAwesome.otf',
   './vendor/font-awesome/fonts/fontawesome-webfont.eot',
   './vendor/font-awesome/fonts/fontawesome-webfont.svg',
   './vendor/font-awesome/fonts/fontawesome-webfont.ttf?v=4.6.3',
   './vendor/font-awesome/fonts/fontawesome-webfont.woff?v=4.6.3',
   './vendor/font-awesome/fonts/fontawesome-webfont.woff2?v=4.6.3',

   './vendor/jquery/jquery-3.1.1.min.js',
   './vendor/jquery/jquery-migrate-3.0.0.js',
   './vendor/jquery-mobile/jquery.mobile-1.4.5.min.js',
   './vendor/jquery-mobile/jquery.mobile-1.4.5.min.css',
   './vendor/jquery-mobile/images/ajax-loader.gif',
   './vendor/jquery-ui/jquery-ui.min.js',
   './vendor/jquery-ui/jquery-ui.min.css',
   './vendor/jquery-validate/jquery.validate.min.js',

   './vendor/waves/waves.min.js',
   './vendor/waves/waves.min.js.map',
   './vendor/waves/waves.min.css',
   './vendor/wow/animate.css',
   './vendor/wow/wow.min.js',

   './vendor/idb/',
   './vendor/idb/lib/',
   './vendor/idb/lib/idb.js',

   './config/nd2settings.js',
   './fragments/bottom.sheet.html',
   './fragments/panel.left.html',
   './fragments/page.home.html',
   './fragments/page.login.html',
   './fragments/page.register.html',
```

```
46    './fragments/page.task.add.html',
47
48    './resources/css/style.css',
49    './resources/fonts/Roboto-Regular.ttf',
50    './resources/img/2.jpg',
51    './resources/img/8.jpg',
52    './resources/img/9.jpg',
53    './resources/img/10.jpg',
54    './resources/img/examples/card_bg_1.jpg',
55    './resources/img/examples/card_bg_2.jpg',
56    './resources/img/examples/card_bg_3.jpg',
57    './resources/img/examples/card_thumb_1.jpg',
58    './resources/img/examples/card_thumb_2.jpg',
59    './resources/img/examples/card_thumb_3.jpg',
60
61    './resources/js/app.js',
62    './resources/js/pushFunctions.js',
63    './resources/js/validation.js',
64    './resources/js/home.js',
65    './manifest.json',
66    './index.php',
67    './'
68 ];
```

Listing 7.3: Service Worker Konfiguration - Ressourcen für Caching

7.3 PushFunctions.js

```
// resources/js/pushFunctions.js

var PUSH_URL = "http://localhost:3000/push";

/**
 * Subscribe push
 *
 * - creates push manager subscription
 * - sends subscription to application server
 */
function subscribePush() {
   navigator.serviceWorker.ready.then(function(serviceWorkerRegistration) {
     serviceWorkerRegistration.pushManager.subscribe({userVisibleOnly: true})
       .then(function(pushSubscription) {
          //Store this subscription on application server
            sendSub(pushSubscription);
          return true;
       })
       .catch(function(e) {
         console.error('Unable to register push subscription', e);
         return false;
       });
   });
  }

/**
 * unsubscribe push
 *
 * - unsubscribe push manager
 * - remove subscription from application server
 */
function unsubscribePush() {
   console.log('unsubscribing...');
   navigator.serviceWorker.ready.then(function(serviceWorkerRegistration) {

   serviceWorkerRegistration.pushManager.getSubscription()
   .then(
     function(pushSubscription) {
     // We have a subscription, so remove it from applications server...
     cancelSub(pushSubscription);
     //... and unsubscribe it
     pushSubscription.unsubscribe().then(function() {}).catch(function(e) {
       console.log('Error unsubscribing: ', e);
     });
      })
   .catch(function(e) {
       console.error('Error unsubscribing.', e);
     });
```

```
49    });
50  }
51
52  /**
53   * send Subscription to application server
54   */
55  function sendSub(pushSubscription) {
56    var deviceId = localStorage.getItem('deviceId');
57    var deviceName = "Hier wird iwann der Username stehen";
58      var endpoint = pushSubscription.endpoint;
59      var subId = endpoint.split("/").pop();
60
61      localStorage.setItem('gcm-regid', subId);
62
63      var authToken = localStorage.getItem("auth-token");
64      fetch(PUSH_URL + "/devices/", {
65          mode: 'cors',
66          method: 'POST',
67          headers: {
68              "Content-Type": "application/x-www-form-urlencoded",
69        },
70          body:
                "deviceName="+deviceName+"&deviceId="+deviceId+"&registrationId="+subId+"&endpoint="+endpo
71      })
72      .then(function(res) {
73        res.json().then(function(data) {
74              // Log the data for illustration
75              console.log(data);
76          });
77    });
78  }
```

Listing 7.4: PushFunctions.js - Verarbeitung der PushSubscriptions