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

Hochschule für Telekommunikation Leipzig
University of Applied Sciences

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Ziele

Diese Dokumentation entstand im Rahmen der Profilierung „Netzbasierte Anwendungen“ im Wintersemester 2016/17 an der Hochschule für Telekommunikation Leipzig (HfTL).

Projektbericht - Bestandteile

- Motivation und Ziele
- Grundlagen
- Anforderungen
- Konzeption (MVC, Methodik, + Alternativen)
- Implementierung
- Zusammenfassung und Ausblick

... Einleitung moderne webtechnologien –> webapps statt nativen Apps ...
... Beschreibung der Aufgabe/des Problems ...

... Versuch der Lösungsfindung/Kurzbeschreibung Projekt ...

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 als Proxy fungieren und zum anderen vom Server gesendete Benachrichtigungen, selbst dann empfangen, wenn gerade keine Web-Page der entsprechenden Domain / Web-App 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 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

3.1 allgemeine Beschreibung der Applikation

Nach erfolgreicher Registrierung und Anmeldung kann der Benutzer Aufgaben anlegen, bearbeiten, anzeigen und löschen. Weiterhin gibt es eine Kontaktliste, in welcher alle Kontakte angezeigt werden, die ebenfalls für die Anwendung registriert sind und zu persönlichen Kontakten hinzugefügt wurden. Aufgaben können mit persönlichen Kontakten geteilt werden. Ebenso ist es möglich Gruppen anzulegen, dieser Kontakte hinzuzufügen und Aufgabe mit der Gruppe zu teilen.

Über Änderungen an Gruppen oder Aufgaben wird der Benutzer über PUSH-Benachrichtigungen informiert. Wenn einer Aufgabe ein Benachrichtigungszeitpunkt angegeben wurde, wird ebenfalls eine PUSH-Notification angezeigt sobald die Aufgabe terminiert.

3.2 funktionale Anforderungen

Im Rahmen dieser Dokumentation werden unter funktionalen Anforderungen diejenigen verstanden, welche zur direkten Zielerfüllung beitragen (vgl. 1.1).

weiter ausführen...

[FA-1] Single Page Application

[FA-2] Offlinefähigkeit Die Benutzung der Webanwendung soll nicht ausschließlich bei bestehender Internetverbindung, sondern ebenfalls Offline reibungslos möglich sein. Dazu bietet die hybride Webanwendung Mechanismen zum Vorhalten der persistenten Daten und des nutzerspezifischen Datenmodells im Offlinezustand. 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 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

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

3.3 nicht-funktionale Anforderungen

Im Rahmen dieser Dokumentation werden unter nicht-funktionalen Anforderungen diejenigen verstanden, welche nicht zur direkten Zielerfüllung beitragen (vgl. 1.1).

weiter ausführen...

[NFA-1] 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-2] 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-3] Aufgaben anlegen, bearbeiten und löschen. Ein authentifizierte Benutzer können Aufgaben anlegen und anschließend Bearbeiten oder Löschen. Eine Aufgabe muss einen Titel besitzen. Optional kann eine Beschreibung und ein Fälligkeitsdatum hinterlegt werden.

[NFA-4] Aufgaben teilen. Aufgaben können mit mehreren Benutzer 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-8] Gesicherter Zugriff auf API. Der Zugriff auf die API ist nur für authentifizierte Benutzer möglich. Für die Authentifizierung wird das Konzept Token verwendet.

[NFA-9] Ereignisse für Benachrichtigungen. Benutzer, die in einer Aufgabe involviert sind, erhalten Benachrichtigungen über Änderungen an Aufgaben. Wenn für eine Aufgabe eine Fälligkeit mit Benachrichtigung hinterlegt wurde, wird der Benutzer zum entsprechenden Zeitpunkt informiert.
Wird ein Benutzer in eine Gruppe eingeladen bzw. wird einer Gruppe eine Aufgaben hinzugefügt bzw. bearbeitet werden alle Gruppenmitglieder entsprechend Benachrichtigt.

- 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 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 und Synchronisation mit dem Server 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 3.2) 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 hybride 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.

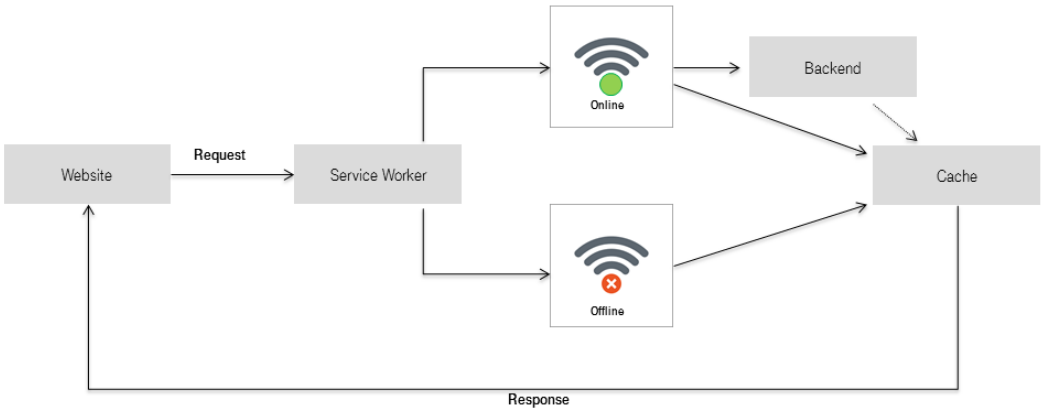

Abbildung 4.1: Caching Strategie `cacheFirst`

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 Seite unabhängig vom Onlinezustand bei Anforderung schnell geladen und dem Benutzer eine optimale Offline-Benutzererfahrung ermöglicht.

4.1.2 lokale peristente Speicherung und Synchronisation des Application State

Neben der in Abschnitt 4.1.1 beschriebenen Vorhaltung der statischen Ressourcen muss die hybride Webanwendung ebenfalls einen Mechanismus zur Verfügung stellen, um das Datenmodell im Offlinebetrieb bereitzustellen.

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 oder sogar aktuell geladen ist.
Damit eine App, Push-Nachrichten empfangen kann, muss sie einen aktiven Service-Worker haben. Wenn der Service-Worker aktiv ist, kann er Push-Benachrichtigungen über seinen internen Push-Manager (`PushManager.subscribe()`) abonnieren.

Die resultierende `PushSubscription` enthält alle Informationen, die die Anwendung benötigt, um eine Push-Nachricht zu senden: einen Endpoint und 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 für 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 Web-App auf eine Anforderung für Webbenachrichtigungen oder sonstige verwendete Berechtigungen reagieren, indem er der App die Berechtigungen erteilt. 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.

Bild hinzufügen

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. Die Subscription-ID wird aus dem kompletten Endpoint entnommen.

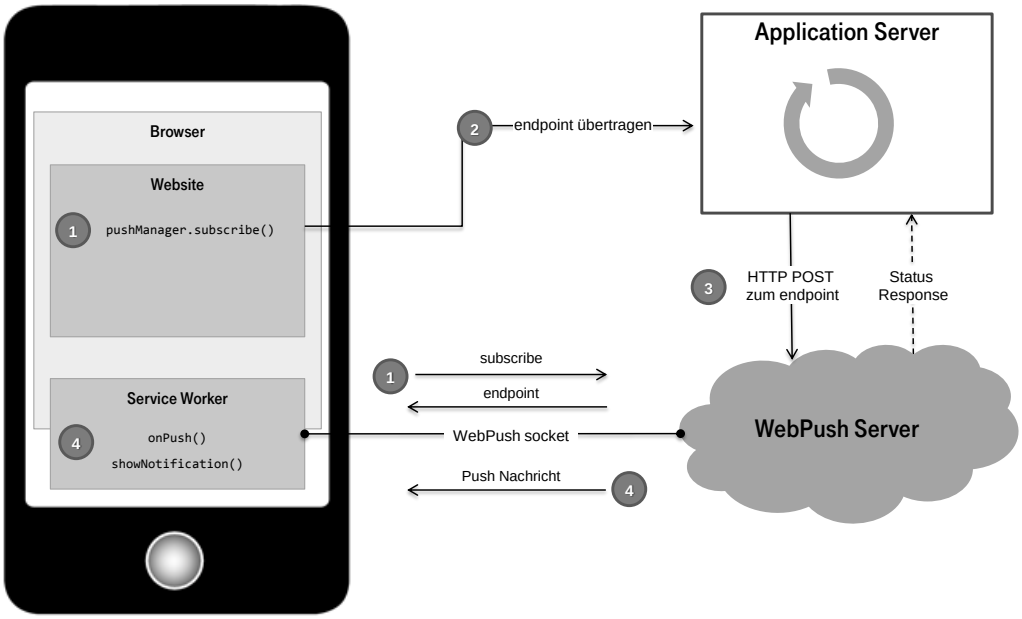

Abbildung 4.2: Push mittels Serviceworker (in Anlehnung an MozillaWiki [1])
Quelle: `https://wiki.mozilla.org/File:PushNotificationsHighLevel.png`

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.

Um eine Push-Nachricht zu senden, muss ein HTTP-POST an die Endpoint-URL 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.

Um Nutzdaten in die Anfrage einzubinden, müssen diese verschlüsselt werden (mit dem öffentlichen Schlüssel des Clients (public key)). Da wir uns gegen eine Nutzdatensendung über den Browseranbieter entschieden haben, entfällt bei uns dieser Schritt.

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 auf für Clientanfragen vorgehalten.
Sobald eine Push-Nachricht vom Client empfangen wird, löst dies ein onPush-Event aus. Der Event-Listener reagiert mit einer direkten Anfrage beim Applicationserver und fragt ggf. vorhandenen Payload für die aktuelle Message ID ab.

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 Serviceworker-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.

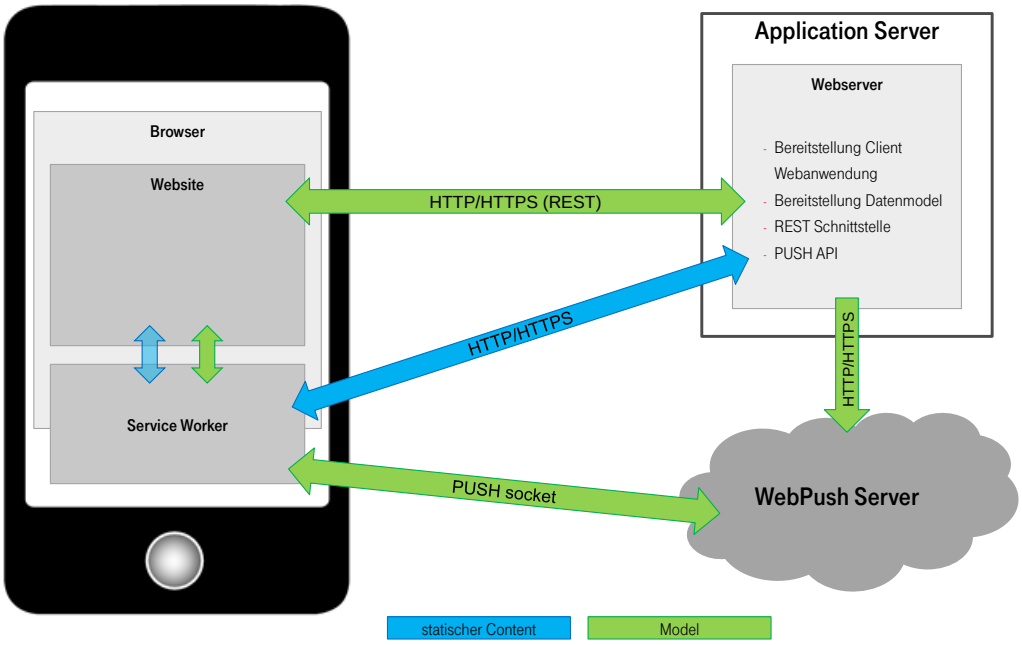

Abbildung 4.3: Archtikturbeschreibung - Umsetzung mit Serviceworker

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 3.2, 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.

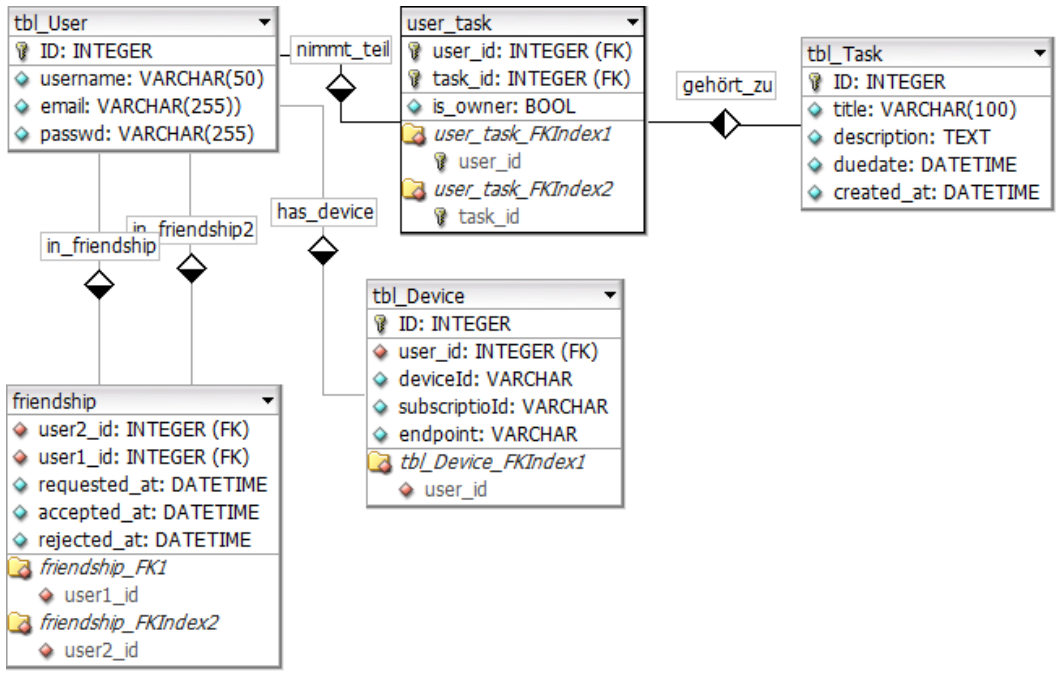

Abbildung 4.4: Datenmodell

Zur Erfüllung der, in Abschnitt 3.3 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

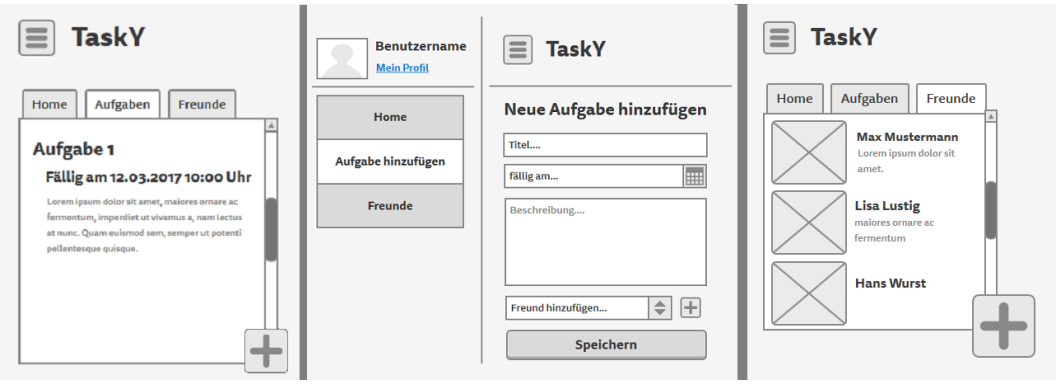

Abbildung 4.5: Wireframe/Mockup-Entwurf grafische Benutzeroberfläche

Beschreibung des UI ...

Um das „Look and Feel“ einer nativen App zu erreichen wird das UI-Framework **nativeDroid2** verwendet.

5 Implementierung

5.1 Applicationserver

5.1.1 Datenbank

5.1.2 REST-Schnittstelle

5.2 Serviceworker

5.2.1 Caching der statischen Ressourcen

5.2.2 Push-Notification

```
// /js/app.js
if ('serviceWorker' in navigator)
{
    navigator.serviceWorker.register('sw.js').then(function(reg) {

        if(reg.installing)
        {
            console.log('Service worker installing');
        }
        else if(reg.waiting)
        {
            console.log('Service worker installed');
        }
        else if(reg.active)
        {
            console.log('Service worker active');
        }

    }).catch(function(error)
    {
        // registration failed
        console.log('Registration failed with ' + error);
    });
}

\caption{Einrichtung Serice Worker}
```

6 Zusammenfassung und Ausblick

... Was kann nicht geleistet werden? ...

... Was ist eventuell zukünftig möglich ? ...

6.1 Fazit

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.

Literaturverzeichnis

[1] Mozilla: *Firefox/Push Notifications - MozillaWiki*. `https://wiki.mozilla.org/Firefox/Push_Notifications`. Version: 08.01.2017

Abbildungsverzeichnis

7 Anhang

7.1 API Beschreibung

Löscht eine einzelne Aufgabe

Tabelle 7.1: Übersicht API Routen

/api/signup

Request:	**Response:**
• `username`: (String) gewünschter Benutzername • `password`: (String) Passwort • `email`: (String) E-Mail Adresse	• `username`: (String) gewünschter Benutzername • `password`: (String) Passwort • `email`: (String) E-Mail Adresse

Benutzer anlegen	
URL	`/api/signup`
Methode	`GET`
Request-Parameter	Required: • `username`: (String) gewünschter Benutzername • `password`: (String) Passwort • `email`: (String) E-Mail Adresse
Success-Response	• Code: 200 • Content: `success: true, message: 'Successful created new user.'`
Legt einen neuen Benutzer an.	
Request `username`: (String) gewünschter Benutzername `password`: (String) Passwort `email`: (String) E-Mail Adresse	

- `username`: (String)
  gewünschter Benutzername
- `password`: (String)
  Passwort
- `email`: (String)
  E-Mail Adresse

Response

- 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)

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