

# RockIt



## Rich Media Applications

---

Erstellung einer HTML5-/CSS-/JavaScript-/Type-Script-App unter Verwendung von Cordova.

1. Autor

Julian Czech  
727545

[julian.czech@hs-osnabrueck.de](mailto:julian.czech@hs-osnabrueck.de)

2. Autor

Julia Schweigert  
731425

[julia.schweigert@hs-osnabrueck.de](mailto:julia.schweigert@hs-osnabrueck.de)

Prüfer

Dipl.-Inf. Björn  
Plutka

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung "RockIt"</b>	<b>3</b>
<b>2</b>	<b>Zielgruppe</b>	<b>4</b>
<b>3</b>	<b>Anleitung</b>	<b>5</b>
3.1	Registrierung	5
3.2	Gewohnheiten (habits)	6
3.3	Ziele (goals)	7
3.4	Challenges starten	8
<b>4</b>	<b>Technische Umsetzung</b>	<b>9</b>
4.1	SQLite	10
4.2	Challenges	13
<b>5</b>	<b>Verbesserungsvorschläge</b>	<b>16</b>
<b>6</b>	<b>Fazit</b>	<b>17</b>
<b>7</b>	<b>Eigenständigkeitserklärung</b>	<b>18</b>

# 1. Einleitung – “RockIt”

Schlechte Gewohnheiten verfolgen uns jeden Tag. Jeder hat nicht nur einmal versucht, sich neue gute Gewohnheiten anzueignen, um seinen **Lebensstil zu verbessern**, sei es durch mehr Sport, Lesen oder früher schlafen zu gehen. Meistens scheitern diese Vorhaben daran, dass man keine Kontrolle hat, die Motivation verliert, oder seine Vorhaben vergisst. Deshalb ist es vorteilhaft, diese **schriftlich festzuhalten**, oder das Vorhaben mit einem Partner gemeinsam zu starten. Letzteres hat einen positiven Effekt, es steigert nämlich die **Motivation**.

Durch einen weiteren Teilnehmer unterliegt man einerseits einer ständigen Kontrolle und andererseits entsteht ein gewisser **Wettbewerb**. Keiner will als erstes aufgeben und so stachelt man sich gegenseitig an. Genau dieser Effekt wurde in der App “**RockIt**” umgesetzt.

Da jeder Mensch seine eigenen Ziele verfolgt, findet man nur schwer einen Partner, mit dem man zusammen ein gemeinsames Vorhaben anstreben kann. “RockIt” bietet den Nutzern die Möglichkeit nicht nur seine eigenen Gewohnheiten zu verwalten und den **Fortschritt der letzten Wochen** zu sehen, sondern



Challenges  
eingehen




Wettbewerb

auch den Fortschritt der Freunde und das unabhängig davon, wer sich welche Vorhaben gesetzt hat.

Im Allgemeinen handelt es sich bei der App um eine **Lifestyle-App**, welche in erster Linie den Benutzer dabei unterstützt, sich neue Gewohnheiten anzueignen.

Die App heißt “**RockIt**” und betont damit den einzigartigen motivierenden Aspekt der App, welcher diese gegenüber anderen Lifestyle Apps herausstechen lässt. “**RockIt**” soll den Nutzer motivieren, sich nicht nur langfristige Gewohnheiten anzueignen, sondern auch einmalige Ziele zu setzen oder Freunde zu eintägigen Challenges herauszufordern.

Die **täglichen Challenges** sind eine weitere Besonderheit dieser App, welche hauptsächlich für die Dokumentation und Verwaltung der eigenen Vorhaben gedacht ist. Durch das Feature der Challenges jedoch wird die App zu einem **unterhaltsamen Begleiter**. Die App bietet eine Vielzahl an möglichen Challenges, die zusammen mit einem weiteren Kontakt durchgeführt werden können. Falls man also Langeweile hat, hat man immer die Möglichkeit sich für den heutigen Tag einer Herausforderung zu stellen und zu sehen, was der Tag noch alles bringen kann.



Gewohnheiten  
aneignen



Ziele einhalten

## 2. Zielgruppe



Suche  
nach der  
**Motivation**



Spaß mit  
**Freunden**



Eigene  
**Kontrolle**

Die Zielgruppe der App sind Personen, die sich neue Gewohnheiten aneignen, oder alte abgewöhnen wollen, aber an der Umsetzung scheitern. Ob fehlende Motivation oder fehlende Kontrolle, diese App unterstützt den Nutzer beim Erreichen der Vorhaben. Das Design ist an Jugendliche und **junge Erwachsene** zwischen 15 und 35 Jahren gerichtet.

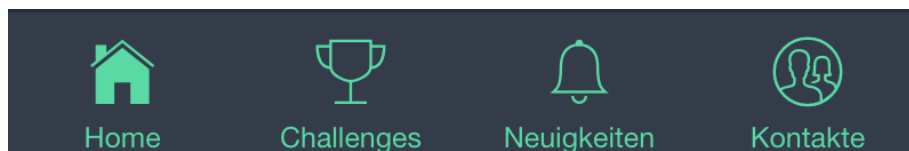
Die Anwender nutzen die App als zur **Kontrolle und Dokumentation der Gewohnheiten und Ziele**. Da die Anwender dazu bestrebt sind, sich selbst zu kontrollieren, geben sie auch nur wahre Begebenheiten an. Die App kann keiner gebrauchen, der sich selbst anlügt und zum Beispiel Ziele als erledigt vermerkt, die gar nicht erreicht wurden. Dafür müsste man die App nicht nutzen. Deshalb ist diese an die Personen ausgerichtet, die nach einer einfachen und **kompakten Lösung eines Gewohnheits-Trackers** suchen.

## 3. Anleitung

Die Verwendung der App wurde so umgesetzt, dass sie intuitiv bedienbar ist. Die Tab-Leiste zeigt vier Icons zur Auswahl an, welche selbsterklärend sind.

Als ersten Button findet man den Home-Button, welcher die **Startseite** und Übersicht über alle Gewohnheiten und Ziele aufzeigt. Über diese Startseite gelangt man auch zu den **Einstellungen**, welche oben rechts beim Zahnrad zu finden sind.

Der zweite Button führt einen zu den täglichen **Challenges**. Dort ist eine Liste von den laufenden Challenges und versendeten Anfragen zu finden. Außerdem kann man auf dieser Seite neue Challenges generieren lassen, was später noch näher beschrieben wird. Weiterhin gibt es eine Seite mit **Neuigkeiten** und eine Übersicht der **Kontakte**.



### 3.1 Registrierung

Der Nutzer muss sich bei **“RockIt”** registrieren. Als Identifikation und für das spätere Anmelden in der App wird eine E-Mail Adresse und ein mindestens 6 Zeichen langes Passwort benötigt. Wenn man die App startet, so kommt man auf die **Anmeldeseite**. Falls man noch keinen Account hat, kann man sich über den Button **Registrieren** einen Account anlegen.

Sobald man **Username, Vorname, Nachname, E-Mail Adresse** und **Passwort** ausgefüllt hat, kann man auf “Registrieren” klicken und wird automatisch eingeloggt.

Falls ein Profil vorhanden ist, kann man sich auf der Anmeldeseite über die E-Mail Adresse und das passende Passwort einloggen.

Das Bild zeigt einen Screenshot der Registrierungsform in der App. Die Form ist auf einem dunklen Hintergrund mit hellen Textfeldern dargestellt. Oben links befindet sich ein zurück-Icon und der Text 'Back'. Oben rechts steht 'REGISTRIEREN'. Die Form enthält folgende Felder: 'Username', 'Vorname', 'Nachname', 'E-Mail Adresse' und 'Passwort'. Am unteren Rand befindet sich ein breiter, hellblauer Button mit der Aufschrift 'Registrieren'.

## 3.2 Gewohnheiten (habits)



**Gewohnheiten** können individuell unter Angabe von Titel, Beschreibung und wöchentlicher Häufigkeit angelegt werden. Die Angabe einer Beschreibung ist jedoch optional.

Eine detaillierte Ansicht einer Gewohnheit informiert den Nutzer sowohl über den **Fortschritt der aktuellen als auch über den der letzten fünf Wochen**. Es wird außerdem ein Fortschritt bzgl. der aktuellen Woche berechnet, der sich auf alle angelegten Gewohnheiten bezieht.

Eine Gewohnheit muss pro Woche so oft als erledigt vermerkt werden, dass die Summe der angegebenen wöchentlichen Häufigkeit entspricht. Dieser Fortschritt wird in der Detailansicht einer Gewohnheit als Doughnut-Diagramm veranschaulicht. Die **Statistik** der letzten fünf Wochen wird ebenfalls in der Detailansicht als Balkendiagramm visualisiert.

Auf der Startseite informiert eine Anzeige den Nutzer darüber, wie viel **Prozent der wöchentlichen Absolvierungen aller Gewohnheiten** bereits erreicht wurden.

Da eine Gewohnheit zu mehr als 100% in einer Woche erreicht werden kann, z.B. wenn man viermal statt dreimal Joggen geht, wird dies dem Nutzer positiv angerechnet und gleicht ein "Nicht-Absolvieren" einer anderen Gewohnheit aus.



## 3.3 Ziele (goals)

**Ziele** werden unter Angabe von Titel, Beschreibung und eines Fälligkeitsdatums individuell angelegt und werden nur einmalig als erledigt vermerkt.

Wurde das Ziel bis zum Fälligkeitsdatum nicht erreicht, gilt es als fehlgeschlagen und kann nicht mehr als erreicht vermerkt werden. Wie auch bei den Gewohnheiten ist die Angabe einer Beschreibung optional, alle anderen Felder sind Pflicht.

Auf der Startseite, die eine Übersicht über alle Gewohnheiten und Ziele zeigt, werden die Ziele **nach Priorität absteigend angezeigt**. D.h. ganz oben stehen die Ziele mit hoher Priorität, ganz unten die mit niedriger Priorität.

Ist ein Ziel abgelaufen, kann es nicht mehr als erledigt vermerkt werden.

Ein **traurig blickendes Emoji-Icon** in der detaillierten Ansicht eines Ziels informiert den Nutzer darüber, dass es nicht erreicht wurde.

Wird ein Ziel vor Ablauf der Deadline als erledigt vermerkt, informiert hingegen ein **lächelndes Emoji-Icon** über das Erreichen des Ziels.



## 3.4 Challenges starten

**Challenges** werden allein oder gegen einen Freund ausgetragen. Was bei einer Challenge getan werden muss, wird zufällig aus einem Pool durch die App bereitgestellter Challenges ausgewählt.

Eine Challenge gilt als gewonnen, sobald ein Nutzer diese **innerhalb von 24 Stunden** als erledigt vermerkt. Gelingt dies einem Nutzer nicht, ist die Challenge für ihn verloren.

Diese Funktionalität dient vor allem dazu, den Nutzer auf eigenen Wunsch im Alltag herauszufordern und seine Selbstdisziplin zu testen.

Um eine Challenge anzulegen, muss man zuerst über die Tableiste auf die Gesamtübersicht der Challenges

gehen. Danach klickt man auf **Zufallschallenge**, um die heutige Herausforderung zu bekommen. Man hat nun die Möglichkeit die Challenge anzunehmen, oder zu verwerfen. Im Falle der Annahme, kann man als nächstes entscheiden, ob man bei dieser Challenge **jemanden herausfordern** will. Dabei muss man wissen, dass man mit einer weiteren Person zur gleichen Zeit nur eine Challenge haben kann.

Nachdem man entschieden hat, ob die jeweilige Challenge alleine oder zusammen mit jemanden anderen absolviert wird, bestätigt man die Auswahl und kann die Challenge bei der erwähnten Übersicht wiederfinden.





## 4. Technische Umsetzung



Die App ist sowohl für iOS als auch für Android verfügbar und wurde mithilfe des **Ionic 2 Frameworks** erstellt. Zudem wurden diverse API, Telefondaten und externe Funktionalitäten hinzugezogen und integriert. Hierzu gehören u.a. die Datenverwaltung für Challenges und Freunden über **Google Firebase**, die lokale Datenverwaltung persönlicher Daten über **SQLite**, die **Kamera** zum Erstellen eigener Profilfotos, die **Kontakte** zum Suchen von Freunden und die Visualisierung eigener Fortschritte mittels Diagrammen von **Chart.js**.



## 4.1 SQLite

Als beispielhafter technischer Aspekt, der elementar für die Umsetzung der Applikation ist, sei **SQLite** zu nennen. Mithilfe dieser Komponente wird die Datenhaltung persönlicher Daten ermöglicht. Es wurde bewusst darauf verzichtet, auch persönliche Daten in der **Firestore** zu sichern, um die Anonymität des Nutzers weitestgehend zu bewahren. Da die Applikation nach dem Anspruch einer Veröffentlichung entwickelt wurde, war diese Entscheidung insofern wichtig, dass eine Speicherung von sensiblen Daten auf einem fremden Server nicht im Interesse des Anwenders liegen dürfte.

Deshalb wird bei erstem Nutzen der Anwendung eine **SQLite**-Datenbank lokal auf dem Gerät gespeichert. Der

angeboten. Da diese Funktionalitäten nahezu überall in der App benötigt werden, ist **SQLite** neben der **Google Firestore** eines ihrer Kernelemente zur technischen Umsetzung.

Die Datenhaltung funktioniert nach einem Prinzip, das im Laufe der Entwicklung immer wieder angepasst wurde. Die Ziele werden nicht in tabellarischer Form, sondern im **Storage** gesichert. Das Speichern von Zielen in tabellarischer Form ist ungeeignet, da angelegte Ziele sich nicht verändern bzw. es keinen statistischen Nutzen im Bezug auf ein einzelnes Ziel gibt, wodurch quasi nur eine Zeile ausreichen würde. Deshalb werden alle Ziele als Array im **Storage** abgelegt, als Key dient auch hier eine

```
public openDatabase(){
    return this.db.openDatabase({
        name: this.authData.getUid()+'data.db',
        location: 'default',
    }).then(() => {
        this.init();
    });
}
```

← Code Snippet 1: Datenbank öffnen und Initialisieren starten

Name dieser Datenbank setzt sich aus dem User-Key und dem Wort **data** zusammen. Auf diese Weise können auch unterschiedliche Nutzer die App auf einem Gerät nutzen.

Die Datenbank wird bei jedem Login mit **openDatabase** geöffnet oder, falls es sich um das erstmalige Login handelt, neu erzeugt. Alle Funktionalitäten, die sich auf die persönlichen Daten, die **SQLite**-Datenbank oder den **Storage** beziehen, werden von der Datei **sqlite.ts**

Kombination aus dem User-Key und dem Wort **goals**. Jedes Mal, wenn sich der Zustand eines Ziels verändert, wird das Array im **Storage** aktualisiert. Analog hierzu verhält es sich bei allen anderen Informationen, die im **Storage** oder der Datenbank gesichert werden. Auch statistische Daten wie die Anzahl erreichter und nicht erreichter Ziele sowie die Anzahl gewonnener und nicht gewonnener Challenges, aber auch das beim letzten Benutzen der App eingestellte **theme** werden im **Storage** gesichert.

Obwohl nicht alle dieser Daten in der finalen Version der App verwendet werden, wurden die Funktionalitäten bewusst nicht aus dem Code entfernt. Des Weiteren wird bei jedem Start der App die aktuelle Woche im **Storage** gesichert. Die gespeicherte Woche dient dazu, zu überprüfen, wie viele Wochen seit dem letzten Start der App verstrichen sind.

Das Sichern der Gewohnheiten verteilt sich über den **Storage** und Tabellen. Im **Storage** wird ähnlich zu den Zielen ein Array aller Gewohnheiten abgelegt. Jede Ge-

aus **Storage** und Datenbank rekonstruiert und über die Laufzeit hinweg als Objektvariablen der **SQLiteData**-Klasse temporär abgelegt werden. Dieses temporäre Ablegen dient zum einen dazu, dass nicht bei jedem Gebrauch dieser Informationen **Storage** und Datenbank befragt werden müssen, zum anderen dient es zur einfacheren Implementierung und Lesbarkeit.

Der Ablauf der Rekonstruktion beginnt mit dem Aufruf der **init**-Methode in **sqlite.ts**. Wichtig hierbei ist die Sequentialisierung der teilweise asynchron ablaufenden

```
public addHabit(title: string, description: string, timesToDo: number) {
    let name = title.replace(" ", "_")

    let sql1 = 'CREATE TABLE '+name+' (week INTEGER PRIMARY KEY AUTOINCREMENT,
timesToDo NUMBER, timesDone NUMBER)';

    let sql2 = 'INSERT INTO '+name+' (timesDone, timesToDo) VALUES
('+0+', '+timesToDo+')';

    [...]
```

← Code Snippet 2: Ausschnitt aus addHabit (Gewohnheit hinzufügen)

wohnheit besteht in diesem Array aus Name und Beschreibung, also aus Informationen, deren Veränderung nicht statistisch festgehalten werden muss. Zudem kann mithilfe der Namen der in diesem Array gesicherten Namen der Gewohnheiten eine entsprechend detaillierte Tabelle zu einer Gewohnheit wiedergefunden werden. Die statistisch wertvollen Informationen zu einer Gewohnheit, **timesToDo**, **timesDone** und **week**, werden in tabellarischer Form über den gesamten Zeitraum der Nutzung der App dokumentiert. Als Name einer Tabelle dient der eindeutige Name, als Primary Key dient die Woche.

Ein entscheidender Zeitpunkt zur Laufzeit einer App ist der Start, bei dem alle verwendeten Informationen

Prozesse, was eines der Hauptprobleme bei der Implementierung darstellt. Asynchron ablaufende Prozesse und die dadurch gewonnenen Informationen stehen teilweise nämlich in Abhängigkeit zueinander. Um dies zu ermöglichen wird häufig auf die durch Methoden zurückgegebenen Promises zurückgegriffen, auf die mit **.then** reagiert wird. Der Prozess der Datenrekonstruktion und Initialisierung lässt sich in Worten folgendermaßen beschreiben:

Bei App-Start wird als erstes die im **Storage** gespeicherte Woche angefordert und neu gesetzt, falls es sich um den ersten App-Start handelt. Anschließend finden mehrere Prozesse gleichzeitig statt. Hierzu gehören das Beziehen der Anzahl erreichter und nicht erreichter

Ziele sowie der Anzahl gewonnener und nicht gewonnener Ziele und das Laden der Gewohnheiten und Ziele, jeweils als Array. Diese drei Teilabläufe stehen in keiner Abhängigkeit zueinander.

Da das Laden der detaillierten Informationen zu den jeweiligen Gewohnheiten abhängig vom zuvor abgeschlossenen Laden des Gewohnheiten-Arrays aus dem **Storage** ist, muss dies zwangsläufig danach stattfinden. Dies wird dadurch ermöglicht, dass auf das **Promise**, das von der den **Storage** abfragenden **get**-Methode zurückgegeben wird, reagiert wird, indem die detaillierten Informationen aus der Datenbank geladen werden. Letzteres erfolgt durch den Aufruf von **updateWeekly**.

Diese Methode bezieht erneut die Woche aus dem **Storage** – da zuvor nur geprüft wurde, ob sie überhaupt im **Storage** existiert – und prüft, wie viele Wochen seit dem letzten Start vergangen sind.

Anschließend werden mit **addRowsEach** der Summe der vergangenen Wochen entsprechend viele Zeilen zu jeder Gewohnheit in der Datenbank hinzugefügt. In die Spalte **timesDone** jeder neu erzeugten Zeile wird dabei jeweils 0 eingetragen, da man eine Gewohnheit in den verstrichenen Wochen zwischen zwei Benutzungen der App nicht als erledigt vermerkt hat. Die letzte Zeile einer Tabelle repräsentiert die aktuelle Woche, die z.B. beim Erledigen einer Gewohnheit aktualisiert wird.

```
private init(){
  this.storage.get(this.authData.getUid()+ 'week')
    .then(response => {
      if(response == null){
        this.storage.set(this.authData.getUid()+ 'week', this.getWeek(new Date()));
      }
      this.getStatisticsFromStorage();
      this.initGoals()
      this.initHabits()
        .then(() => {
          this.updateWeekly()
        });
    });
}
```

↑ Code Snippet 3: Initialisieren aller persönlichen Daten

```
this.db.executeSql('SELECT timesToDo FROM '+name+' WHERE week=(SELECT MAX (week) FROM '+name+');', [])
```

↑ Code Snippet 4: Abfrage von timesToDo der aktuellen Woche einer Gewohnheit

## 4.2 Challenges

Die **Challenges** nehmen bei der Anwendung eine wichtige Rolle ein. Sie sind eine der Besonderheiten der App, welche den User im Alltag herausfordert und die Interaktion mit Freunden erlaubt.

Sobald eine Challenge gestartet wurde, hat der Nutzer 24 Stunden Zeit, um diese zu absolvieren und als abgeschlossen zu markieren, um dadurch die zur Motivation angezeigte Anzahl gewonnener Challenges zu erhöhen. Die Anzahl der gewonnenen Challenges wird im Storage gespeichert. Bei erfolgreichem Beenden der Challenge wird die Funktion **challengeWon()** aus **sqlite.ts** auf-

gerufen und der entsprechende Wert um eins erhöht.

Eine Challenge kann auf zwei verschiedene Weisen ablaufen. Entweder der Nutzer tritt diese alleine an oder zusammen mit einem weiteren Teilnehmer aus der Kontaktliste. Im ersten Fall werden es zwei neue Objekte, die in Firebase erstellt – ein Teilnehmerobjekt in **participants** und ein Challenge-Objekt in **challenges**.

```
/**
 * @desc Erhöht Anzahl gewonnener Challenges
 * @param keine
 * @return keine
 */
public challengeWon(){
  this.storage.set(this.authData.getUid()+'challengesWon', (++this.challengesWon))
    .then(response => {
      this.updateChallengesWon().then(response => {},
        err => {
          console.error("ERROR: getChallengesWon() failed");
        });
    }, err => {
      console.error("ERROR: Couldn't update storage in challengeWon()");
    });
}
```

← Code Snippet 5: Funktion aus **sqlite.ts**

```
"participants" : {
  "wxIN4gGy1vYfDZwtwNqB5PAEgd73" : {
    "-KqNbYdQU6qSBr1d2nUI" : {
      "accepted" : true,
      "participant" : "-1"
    }
  }
},
"challenges" : {
  "-KqNbYdQU6qSBr1d2nUI" : {
    "act" : "act6",
    "done1" : 1,
    "done2" : 1,
    "hour" : 14,
    "minutes" : 28,
    "participantAccept" : true,
    "timestamp" : 1501504092830,
    "user1" : "wxIN4gGy1vYfDZwtwNqB5PAEgd73",
    "user2" : "-1"
  }
}
```

← Code Snippet 6: Teil der JSON-Datei aus Firebase

Das Teilnehmer-Objekt ist durch den **User-Key** des jeweiligen Nutzers zu erreichen. So hat jeder Nutzer eine Liste mit laufenden Challenges. Die Unterobjekte haben wiederum den gleichen Schlüssel wie die Challenges und beinhalten einen Vermerk, ob diese von einem zweiten Teilnehmer angenommen wurde. Da bei diesem Beispiel der User eine Challenge alleine antritt, wird **accepted** auf **true** gesetzt und der **participant** mit **-1** versehen. Der Wert **-1** kennzeichnet hierbei, dass der User der alleinige Teilnehmer der Challenge ist. Dieser Wert wird auch im Challenge-Objekt übernommen, wobei der **user2** mit einer **-1** gekennzeichnet wird.

Wenn der Nutzer eine Challenge bestätigt, wird das aktuelle Datum zu dem Zeitpunkt in der Datenbank abgelegt. Der **Timer** der Challenges ist unter der jeweiligen Detailansicht zu sehen. Da das Intervall des Timers auch korrekt laufen muss, wenn die App geschlossen ist und wieder aufgemacht wird, wird die Restzeit über die angelegte Klasse **ChallengeTimer** berechnet. **ChallengeTimer** ist eine Hilfsklasse und berechnet anhand des aktuellen Datums und des gespeicherten Startdatums aus der Datenbank die verbleibende Zeit, worauf ein Intervall gesetzt wird, um den **Timer** ablaufen zu lassen.

```
export class ChallengeTimer {

    public hours: number = 0;
    public minutes: number = 0;
    private startDate: Date;
    private challengeKey: string;
    private ChallengeProvider: ChallengeData;

    constructor(startDate: Date, ChallengeProvider: ChallengeData, challengeKey: string) {
        this.startDate = startDate;
        this.ChallengeProvider = ChallengeProvider;
        this.challengeKey = challengeKey;
    }
    ...
}
```

↑ Code Snippet 7: ChallengeTimer aus timer.ts

Die Klasse **Challenge-Timer** hat zwei öffentliche Variablen – **hours** und **minutes**.

Diese werden bei der **View** der Detailansicht verwendet, um die Uhrzeit in der App anzuzeigen.

```
<div *ngIf="challengeTimer.minutes < 10">
    <h2>{{challengeTimer.hours}}:0{{challengeTimer.minutes}}</h2>
</div>
<div *ngIf="challengeTimer.minutes >= 10">
    <h2>{{challengeTimer.hours}}:{{challengeTimer.minutes}}</h2>
</div>
```

↑ Code Snippet 8: challenge-detail.html

```
export class ChallengeTimer {

    public hours: number = 0;
    public minutes: number = 0;
    private startDate: Date;
    private challengeKey: string;
    private ChallengeProvider: ChallengeData;

    constructor(startDate: Date, ChallengeProvider: ChallengeData, challengeKey: string) {
        this.startDate = startDate;
        this.ChallengeProvider = ChallengeProvider;
        this.challengeKey = challengeKey;
    }
    ...
}
```

Die Hauptfunktion der Klasse **Challenge-Timer** ist **startTimer()**, welche in **challenge-detail.ts** beim Betreten der Detailansicht aufgerufen wird.

← Code Snippet 9: **startTimer()** aus **timer.ts**

Als erstes wird die Differenz vom aktuellen Datum und dem Startdatum mithilfe von **getTime()** in Millisekunden gespeichert. Diese werden in Minuten umgerechnet, um anschließend mit der Funktion **Math.floor(-tempMinutes/60)** die Stundenanzahl zu berechnen.

Dabei wurde **Math.floor** verwendet, damit das Ergebnis abgerundet wird, da alles, was über eine vollen Stunde geht, später als Minuten dargestellt wird.

den **public**-Variablen **minutes** und **hours** diese Werte zu vergeben.

Nach der Umwandlung und Berechnung wird ein Intervall gestartet, welches nach einer Minute die Variable **minutes** um eins dekrementiert. Falls die Minutenanzahl aber den Wert 0 hat, so wird dieser auf den 59 gesetzt und die Stundenanzahl um eins verkleinert.

```
...
timer = setInterval(() => {
    that.minutes--;
    if (that.minutes < 0) {
        that.minutes = 59;
        that.hours--;
        //Timer Ende bei hours < 0 und minutes < 0
        if (that.hours < 0) {
            that.hours = 0;
            that.minutes = 0;
            this.ChallengeProvider.updateChallengeCancel(that.challengeKey);
            clearInterval(timer);
        }
    }
}, 60000);
...
```

← Code Snippet 10: Forts. **startTimer()** aus **timer.ts**

Der Rest, der übrig bleibt und nicht mehr zu den Stunden gezählt wird, ist die Minutenanzahl, welche über die **Modulo-Operation** errechnet wird.

Nachdem die Darstellung der Zeit zu Stunden und Minuten umgewandelt wurde, wird diese von 24 Stunden subtrahiert, um die verbleibende Zeit zu erhalten und

Das Intervall läuft ab, wenn sowohl die Stunden als auch die Minuten gleich Null sind. Dabei wird die Funktion **updateChallengeCancel(key: string)** im Challenge-Provider (**challenge-data.ts**) aufgerufen, um die Challenge zu beenden, da diese nicht in binnen 24 Stunden abgeschlossen wurde.

## 5. Verbesserungsvorschläge

Eine App ist nie fertig. Auch nach Fertigstellung dieser ersten Version gibt es noch immer Stellen, die man hätte besser umsetzen können. Ein Beispiel hierfür ist die Verwendung von Arrays für Ziele und Gewohnheiten, statt derer man besser Maps mit Key-Value-Paaren hätte verwenden können, um ein ständiges Iterieren zu vermeiden. Im Zusammenhang mit dieser App ist die Relevanz jedoch eher gering, da es sich nie um besonders große Datenmengen handelt.

Ein zweiter Punkt ist die **Synchronisation** der asynchron ablaufenden Prozesse, die ggfs. hätte geschickter umgesetzt werden können. Da viele Methodenaufrufe eben asynchron ablaufen und zudem nur Promises als Rückgabe zulassen, ist man gezwungen, Probleme auf andere Art und Weise zu lösen, als man es intuitiv getan hätte. Dies jedoch ist den Eigenschaften von TypeScript zuzuordnen, mit denen man sich erstmalig und intensiv unter gewissem Zeitdruck auseinandersetzen musste. Umso größer ist aber die Erfahrung für spätere

Projekte. Aus zeitlichen Gründen wurde außerdem darauf verzichtet, Informationen zu Gewohnheiten und Zielen **bearbeiten** zu können, wobei dieser Option von uns keine große Relevanz beigemessen wurde, da z.B. die deadline von Zielen ohnehin nicht bearbeitet werden soll (Deadline ist Deadline) und zum anderen auch die **timesToDo** einer Gewohnheit zumindest nicht nach unten hin korrigiert werden sollen. Allenfalls die Beschreibung könnte bei Bedarf angepasst werden, wobei über ihre Notwendigkeit generell diskutiert werden kann.

Weiterhin sind einem immer mehr Funktionen eingefallen, die man hinzufügen könnte. Eine dieser Funktionen wäre z.B., dass man bei einer Challenge-Anfrage eine **kurze Nachricht** an den zweiten Teilnehmer senden kann. Es gab auch weitere Features, wie eigene Challenges vorschlagen zu können, eine Challenge mit mehreren Teilnehmern durchzuführen und viele mehr, die aufgrund der Zeit leider keinen Einzug in die App gefunden haben.

Plan to throw one (implementation) away;  
you will, anyhow.

Fred Brooks



## 6. Fazit

Bei der Implementation der App spielte vor allem die **Datenkoordination** eine große Rolle und hat den Großteil des Entwicklungszeitraums in Anspruch genommen.

Hierzu gehören sowohl das Datenhandling in **Google Firebase** und **SQLite** als auch die Synchronisation der häufig asynchron ablaufenden Prozesse. Das Koordinieren von Challenges und das Integrieren eines Timers im Zusammenhang mit zwei beteiligten Nutzern ist ein beispielhaftes Problem bezüglich der Datenhaltung,

Das Initialisieren der persönlichen Daten und deren Rekonstruktion aus der SQLite-Datenbank ist hingegen ein beispielhaftes Problem bezüglich der Synchronisation.

Beide Probleme haben im Laufe der Entwicklung mehrfach zum **Verwurf von Code-Abschnitten** geführt, da beim Testen immer wieder unvorhergesehene Unstimmigkeiten auftraten, welche eine erneute Planung erforderten.

Eine Funktionalität einer App hinzuzufügen erfordert zudem viel Planung vorab. Erste Versionen von Funktionen mussten zum Teil komplett umgearbeitet werden, weil diese im weiteren Verlauf der Entwicklung zum Beispiel nicht erweiterbar waren.

Damit dieses schnell erkannt wird, wurde nach jeder Implementation einer neuen Funktion, diese **ausgiebig getestet**, um mögliche zukünftige Probleme zu erkennen und abzuwenden.

Abschließend war es interessant zu sehen, dass man in der Lage ist mit dem **Ionic-Framework** eine funktionierende Application zu erstellen und aufgrund der **Erfahrung** ist man nun in der Lage seine eigene App zu programmieren.

Plan to throw one (implementation) away;  
you will, anyhow.

Fred Brooks

## 7. Eigenständigkeitserklärung

Hiermit versichern wir, dass sowohl Projekt als auch Projektbericht eigenständig erarbeitet wurden.

Ort, Datum: .....

Unterschrift: .....

{Julia Schweigert}

Unterschrift: .....

{Julian Czech}