HAW HAMBURG

Fakultät Design, Medien und Information
Department Medientechnik

Bachelor-Thesis

Zur Erlangung des Grades Bachelor of Science

Konzipierung und Realisierung einer UX-optimierten App des Hamburger Verkehrsverbunds (HVV)

auf möglicher Grundlage von React Native

unter Zuhilfenahme von Redux und TypeScript

Vorgelegt von

Julia Kott

Matrikelnummer: [redacted]

Studiengang: Media Systems (B.Sc.)

Erstprüfer

Prof. Dr. Andreas Plaß

Zweitprüfer

Jannis Haker

Abgabedatum

26.08.2019

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelor-Thesis mit dem Titel:

_____

_____

selbstständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

_____

Gliederung

I. Einleitung

In dieser Arbeit geht es darum, Schwachstellen in der Bedienung einer bestehenden App zu finden, zu erläutern, entsprechend umzugestalten und eine mögliche Implementierung aufzuzeigen.

Die Motivation dafür beruht in erster Linie auf der Begeisterung für den Bereich der UX und dem Wunsch, eine frustrierende Nutzererfahrung zu analysieren und zu verbessern.

UX wird (fälschlicherweise) oft mit UI – der Benutzeroberfläche – gleichgestellt. Leider, denn das Aussehen bildet nur einen wirklich kleinen Teil davon und viel mehr liegt der Schwerpunkt in der Psychologie des Menschen und seinem Entscheidungsmuster.

Als *frustrierende Nutzererfahrung* wurde die App des Hamburger Verkehrsverbunds (kurz: HVV) gewählt. Die Version der App, die dieser Arbeit zugrunde liegt, ist 4.2.2 vom 27.02.2019.

Die Implementierung erfolgt mit React Native, da ich selbst seit mehreren Jahren mit React in der Frontend-Entwicklung arbeite und entsprechend mit der Funktionsweise von React Native bereits vertraut bin.

Zwar ist seit der Veröffentlichung der neuen Version am 21. November 2017[1] einiges an Zeit vergangen und in der App passiert, schenkt man jedoch den Bewertungen im beispielsweise Google Play Store weiter Beachtung, gibt es noch immer einige frustrierte Nutzer.

Unglücklicherweise lässt sich im Google Play Store die genaue Zahl dieser Nutzer nicht bestimmen – eine grobe Richtung bietet nur die allgemeine Übersicht der Bewertungen in Abbildung 1.

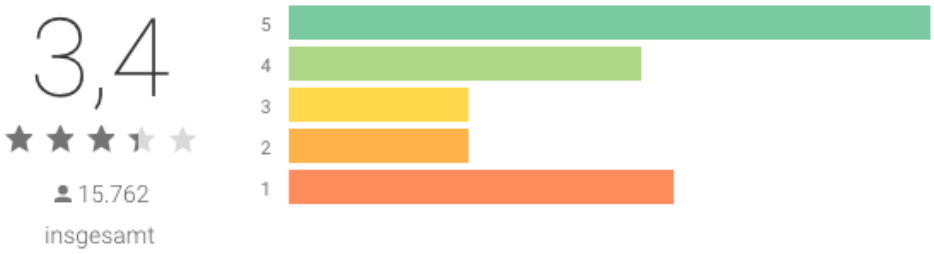


Abbildung 1 - Bewertung aus dem Google Play Store (https://play.google.com/store/apps/details?id=de.eos.uptrade.android.fahrinfo.hamburg - letzter Aufruf: 26.08.2019 - 7:29 Uhr)

[1] 2017, „HVV-App ab heute rundum erneuert – jetzt mit Monatskartenverkauf“, https://www.nahverkehrhamburg.de/hvv-app-ab-heute-rundum-erneuert-jetzt-mit-monatskartenverkauf-8959/, letzter Aufruf: 26.12.2018 – 18:02 Uhr

II. Konzeptionierung

Um die App einer Umgestaltung unterziehen zu können, müssen zuerst ein paar Fragen beantwortet werden.

Grundlegend dafür ist das allgemeine Verständnis, was UX eigentlich bedeutet.

Weiter muss natürlich geklärt werden, welche Bereiche die momentane App gut löst, welche sie nicht gut löst und wie andere Verkehrsverbünde die Problemstellungen gelöst haben.

II.I. Bedeutung von UX

UX und UI werden gern vertauscht, zusammengefasst oder für ein- und dasselbe gehalten.

Es ist logisch, dass das passiert: Das Endprodukt besteht für den Nutzer aus dem, was er sieht. Also der UI. Was davor passiert ist und welche Gedanken in das Produkt geflossen sind, ist schlichtweg nicht sichtbar.

Folgend wird also darauf eingegangen, was UX wirklich bedeutet.

UX steht für User Experience und kann direkt als Nutzererlebnis übersetzt werden.

Nutzerlebnis ist an sich aber ein nicht sehr vielsagender Begriff.

Produkte werden, wenn nach UX designt wird, auf der Grundlage der Nutzer entworfen. Im Gegensatz zu einem anderen Ansatz, bei dem Entwickler eine App gestalten und programmieren und die Erwartungen und Anforderungen von den Endnutzern nicht aus Sicht derer sehen, sondern aus der eigenen. Solche Apps ergeben für die Entwickler selber absolut Sinn. Ignoriert wird dabei jedoch, dass die entwickelte App nicht von den Entwicklern bedient werden muss, sondern von den eigentlichen Nutzern.

Es mag selbstverständlich klingen, Produkte so zu gestalten, dass sie die Probleme der Nutzer lösen. Sicher wird dies auch versucht und kein Entwickler gestaltet eine Anwendung böswillig, um die Nutzer zu frustrieren. Was jedoch fehlt, ist die direkte Kommunikation mit eben diesen Nutzern oder der Versuch, sich in diese hineinzuversetzen.

In den meisten Fällen formulieren die Entwickler also selber Anforderungen, die sie für sinnvoll erachten, ohne selbst in die Personengruppe zu fallen, in der sich die Nutzer befinden. Dadurch bleiben die spezifischen Probleme außer Acht gelassen, da keiner der Entwickler sie aus eigener Erfahrung kennt.

UX-Design fängt genau dort an: Sich aktiv mit dem Endnutzer auseinandersetzen, Probleme, Anforderungen und Wünsche sammeln und auch berücksichtigen.

In der Realität stehen diese Endnutzer aber oft nicht persönlich zur Verfügung.

Es muss also ein hohes Maß an Empathie aufgebracht und sich hineinversetzt werden.

Je nach Produkt – beispielsweise eine Kaffeemaschine – kann das schon genügen. Jeder hat in seinem Leben vermutlich schon eine Kaffeemaschine bedient. Von einer klassischen Filtermaschine wird erwartet, dass Wasser eingefüllt werden muss, Filter und Kaffee hinzugefügt werden müssen und wieder anschließend ‚Start' gedrückt werden kann. Man weiß auch, dass ein Behälter platziert werden muss, damit der Kaffee am Ende nicht über den Tisch fließt.

Dieser Gedankengang nennt sich Konzeptmodell. Konzeptmodelle sind etwas, das durch Erfahrungen entwickelt werden kann, durch Anleitungen vorgegeben und/oder visualisiert wird oder von Person zu Person weitergegeben wird. Am Ende ist solch ein Modell ziemlich verinnerlicht und dadurch intuitiv. Man könnte es als Erwartung ansehen, dass auf Aktion A immer Reaktion B erfolgt.[2]

Wenn dieses Konzeptmodell durch etwas gestört wird, beziehungsweise die Erwartung auf Reaktion B nicht erfüllt wird, erweckt das keine positiven Gefühle im Anwender. Stattdessen kann es schnell zu Frustrationen führen, da die erlernte und erwartete Bedienung nicht das gewünschte Ziel erreicht. Und genau diese Frustration soll verhindert werden.

Am Beispiel der Kaffeemaschine: Dass allgemeine Konzeptmodell, wie sie zu benutzen sein wird, ist bekannt. Mit etwas Empathie kann also eine Kaffeemaschine entworfen werden, die von jedem, der sie bedienen soll, ohne Erklärung bedient werden kann.

Ändert man die Nutzergruppe von ‚Mensch' nun auf ‚Lehrer'. Natürlich kann ein Lehrer eine einfache Kaffeemaschine problemlos bedienen. Stellt man sich hingegen die Frage, wie eine für Lehrer optimierte Kaffeemaschine entworfen werden könnte, sieht es schon etwas anders aus.

Wo wird diese Kaffeemaschine stehen? Wer wird sie bedienen? Wollen mehrere auf einmal einen Kaffee oder nur sporadisch ein paar Menschen? Wie viel Zeit hat man zur Verfügung, die Maschine zu befüllen? Wird unter Stress Kaffee gekocht? Und so weiter.

Das alles sind Fragen, die im Rahmen der UX beantwortet werden müssen. Mit genügend Einfühlen können einige Fragen selbst beantwortet werden.

[2] Norman, Don – *The Design of Everyday Things* (überarbeitete und erweiterte Auflage), S. 24 – „Konzeptmodelle", Verlag Franz Vahlen GmbH – München, 2016

Die Maschine wird beispielsweise mit hoher Wahrscheinlichkeit im Lehrerzimmer stehen – sie sollte also nicht zu viel Platz einnehmen. Vermutlich werden mehrere Lehrer auf einmal einen Kaffee haben wollen – eine Maschine, die nur eine Tasse zurzeit ausgibt, würde zu langen Wartezeiten führen.

Manche dieser Fragen können aber auch sehr spezifisch sein und können nicht selbst beantwortet werden, sofern man nicht schon selbst als Lehrer tätig war. Dann ist es also von Vorteil, direkt mit dieser Nutzergruppe zu sprechen.

Das Vorgehen an sich ist simpel. Man versetzt sich in die Menschen hinein, die das Produkt am Ende benutzen werden und erstellt auf dieser Grundlage das Konzept, das Design oder direkt das Produkt (oder lässt es auf dieser Grundlage entwickeln).

Leider ist es ebenso zeitintensiv wie nützlich. Ein positives Nutzererlebnis bindet den Nutzer an das Produkt, weil er es gern benutzt. Er wird es vermutlich sogar empfehlen. Das bedeutet wiederum, dass das Produkt sehr wahrscheinlich erfolgreich sein wird. Auf der anderen Seite stehen jedoch die Kosten, die dadurch entstehen. Mehr Zeit bedeutet zwangsläufig mehr Kosten.

II.II. Die momentane HVV-App auf dem Prüfstand

Um im Anschluss eine verbesserte Nutzererfahrung gestalten zu können, muss zunächst analysiert werden, wie die momentane App aufgebaut ist und wie andere ÖPNV-Gesellschaften ihre Apps aufbauen. Daraus kann man erkennen, welche Lösungen gut und welche eher schlecht funktionieren und welche man entsprechend umstrukturieren muss, um einen angenehmeren Weg zu finden.

Da der Umfang der Apps teils sehr groß ist, beschränkt sich diese Analyse auf den App-Start, die Verbindungssuche und den Fahrkartenkauf. Drei der Hauptfunktionen (wobei das später erwähnte On-Boarding nicht eine dieser Funktionen sein wird) werden in die umgestaltete App einbezogen, alle weiteren Funktionen werden zunächst nicht näher betrachtet oder verwendet.

Vergleich mit den ÖPNV-Apps anderer Städte

Zum Vergleich werden die Apps des BVG (Berlin – Version 6.3.10), KVB (Köln – Version 1.07) und des MVV (München – Version 4.6.20190321) herangezogen, aus dem einfachen Grund, dass die Apps dieser Städte noch nicht (von mir) persönlich verwendet wurden und das Urteil dadurch in keiner Weise durch Vorkenntnisse beeinflusst wird.

Alle Apps wurden direkt nach dem Download geöffnet oder zurückgesetzt und befanden sich im Werkszustand.

In dieser Arbeit handelt es sich wie bereits erwähnt um eine Umgestaltung und nicht um eine Neugestaltung, daher werden gute Lösungen beibehalten oder minimal abgeändert und die Probleme verbessert. Es ist also nicht verwerflich, die besten Teile der vier Apps zu verwenden. Allgemein ist das Ziel jedoch nicht, - umgangssprachlich formuliert – das Rad neu zu erfinden, sondern etwaige Kanten zu glätten, um es besser rollen zu lassen.

App-Start

Als erstes wird der Weg nach dem erstmaligen Öffnen der jeweiligen App bis zur Verbindungssuche betrachtet.

Die App des HVV und die des BVG setzen beim Start auf ein On-Boarding.

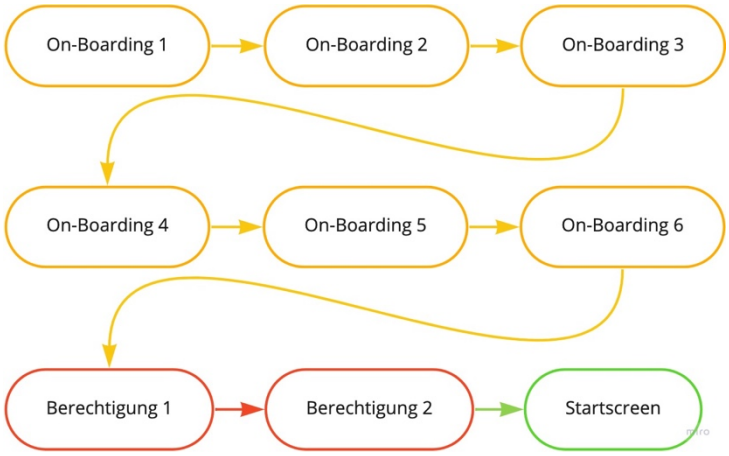


Abbildung 2 - App-Start Verlauf der HVV App

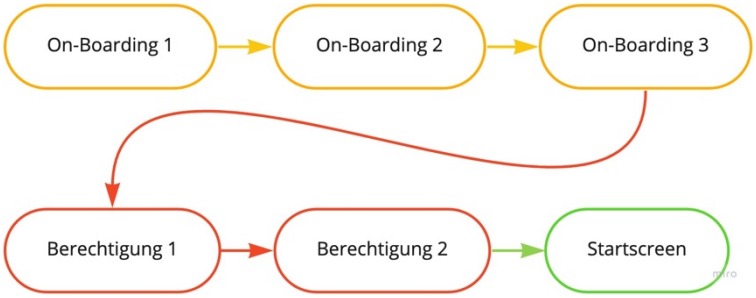


Abbildung 3 - App-Start Verlauf der BVG App

[3]Ein On-Boarding besteht im Optimalfall aus drei bis fünf Screens. Diese Screens beschreiben manche Funktionen der App, ihren Nutzen oder ihr Bedienkonzept. Die Screens sollten jeweils nur einen Punkt hervorheben.

Natürlich ist ein On-Boarding nicht immer notwendig. Sollte es jedoch richtig eingesetzt werden, profitiert die UX sehr davon. Das trifft vor allem auf komplexere Apps zu.

[3] Semler, Jan – *App-Design – Alles zu Gestaltung, Usability und User Experience*, S. 161 – „5.3 Holen Sie Ihre Nutzer ab“, Rheinwerk Verlag GmbH – Bonn, 2016
(die folgenden vier Absätze bedienen sich ebenfalls der Inhalte dieser Quelle)

Man unterscheidet generell drei Arten des On-Boardings. Zum einen das *funktionsorientierte*, welches die Funktionen der App beschreibt, zum Anderen das *vorteilsorientierte*, welches den Nutzen und den Mehrwert näherbringt und zu guter Letzt das *progressive* On-Boarding, welches den Nutzer gezielt an die Hand nimmt und durch die App begleitet – entsprechend nicht aus Screens beim erstmaligen Start besteht, sondern interaktiv durch die App verläuft.

Sofern eine App nicht sehr komplex ist oder viele versteckte Funktionen beinhaltet, reichen kurze funktions- und vorteilsorientierte On-Boardings aus. Alle drei lassen sich durchaus auch miteinander kombinieren, sollten aber nie den Rahmen sprengen. Das On-Boarding soll den Einstieg erleichtern, aber nicht das Gefühl entstehen lassen, aufgehalten zu werden. Ein Überspringen-Button ist daher notwendig.

Generell könnte man erst einmal davon ausgehen, dass das Bedienkonzept einer ÖPNV-App bekannt ist. Stark vereinfacht dargestellt, ist das Ziel primär: Von A nach B kommen. Man gibt den Start ein, das Ziel, im Idealfall die Uhrzeit, wann die Abreise oder Anreise stattfinden soll und als Resultat werden verschiedene Verbindungen aufgelistet, von denen eine ausgesucht wird. So funktionieren zumindest die hier behandelten Apps (später mehr über die Funktionswege).

Sollten jedoch Funktionen hinzukommen, die nicht der Regelfall sind – nehmen wir beispielsweise die Umgebungskarten und Fahrkartenkäufe – kann ein On-Boarding von Nutzen sein. Im On-Boarding hingegen zu beschreiben, dass Start, Ziel und Uhrzeit eingegeben werden müssen und wo diese einzugeben sind, würde die UX stören, da dieses Bedienkonzept geläufig ist. Weniger kann hier also durchaus mehr sein.

Der HVV und die BVG beziehen sich beide auf Funktionen, die auf den ersten Blick nicht gleich ersichtlich sind. Der KVB und der MVV verzichten hingegen auf ein On-Boarding, obwohl es versteckte Funktionen gibt und es durchaus sinnvoll wäre.

Manche der aufgeführten Funktionen werden in der Umgestaltung nicht berücksichtigt. Um Schwachstellen des On-Boardings aufzuzeigen, wird dennoch kurz auf sie eingegangen.

Beim HVV ist das On-Boarding mit sechs Screens in der Theorie einen zu lang und funktionsorientiert.

Die Funktionen sind mithilfe von Screenshots der App bebildert und enthalten folgende Informationen:

1. „Überblick verschaffen – Mit der Karte auf dem Startbildschirm kannst du dich noch besser im HVV-Gebiet orientieren und sofort die nächste Halstestelle in deiner Umgebung finden. Du kannst die Karte die Karte deaktivieren, indem du sie nach oben schiebst."

2. „Bring mich schnell nach… - Hause, zur Arbeit oder auf den Kiez. Deine Lieblingsadressen und -Haltestellen kannst du ganz einfach anlegen und verwalten. Mit einem Klick bekommst du sofort eine Verbindungsauskunft von deinem aktuellen Standort aus."

3. „Dein Weg ans Ziel – Mit der neuen Verbindungsübersicht siehst du deine Fahrtoptionen. Per Klick auf den Preisbutton kannst du Fahrkarten kaufen. Du willst das Design ändern, klicke auf den oben markierten Button."

4. „Die Fahrkarte bitte! – Deine aktuell gültige Fahrkarte aufrufen oder schnell das passende Ticket kaufen: Mit der Fahrkarten-Widget-Funktion alles kein Problem! Tipp: Ab sofort kannst du auch Monatskarten über die HVV App kaufen."

5. „Die Rechnung bitte! – Mit einem Klick kannst du dein letztes Ticket einfach noch einmal kaufen und dir den Kaufbeleg bequem per E-Mail zusenden lassen."

6. „Abfahrt! – In der Rubrik „Abfahrten" siehst du sofort, welcher Bus oder welche Bahn als nächstes von deiner Haltestelle aus fährt. Mit dem Filter wählst du deine bevorzugten Verkehrsmittel aus."

Betrachtet man die Funktion genauer und läuft den Weg ab, stellt man fest:

1. Die App startet mit einer deaktivierten Karte. Es wäre entsprechend sinnvoller, darauf hinzuweisen, wie sie aktiviert wird. Der Pfeil bleibt zwar bestehen und zeigt in die entgegengesetzte Richtung, aber das Bild, dass man sieht kommt einem dennoch nicht so bekannt vor und erfordert eine kleine Bedenkzeit.

2. Unter der Start-/Ziel-Eingabe befinden sich Icons, die ein Haus und eine Krawatte und ein „+" darstellen. Durch die Info über die Schnellziele, kann man sich denken, dass sie für „Zuhause", „Arbeit" und „Hinzufügen" stehen. Explizit wird aber nicht erwähnt, dass die Schnellziele unterhalb der Eingabe zu finden sind. Weiter ist nicht ersichtlich, wo die gezeigte Übersicht aller Schnellziele zu finden ist. Ein passenderes Beispiel wäre also effektiver.

3. Zunächst beschreibt der Screen zwei unterschiedliche Funktionen. Wie bereits erwähnt sollte pro Screen jedoch nur eine Funktion erläutert werden.
   Die Information, dass die Fahrkarte direkt gekauft werden kann, wird nicht auf dem Bild hervorgehoben, aber im Text erwähnt. Die Darstellung des Fahrkartenpreises suggeriert jedoch, dass es sich um einen Button handelt – hervorzuheben, dass es wirklich ein Button ist, ist somit nicht zwingend notwendig.
   Die zweite Information, dass über eine Schaltfläche, die Darstellung geändert werden kann, ist ebenso nicht zwingend notwendig, da die initiale Darstellung alle Informationen zur Verbindung wiedergibt und die Änderung entsprechend lediglich eine Zusatzfunktion ist, die für den Gebrauch nicht relevant ist. Weiter wird auch nicht erwähnt, inwiefern die Darstellung geändert wird oder einen etwaigen Mehrwert hat. Hinzukommt, dass der Button, um die Darstellung erneut zu ändern, der gleiche ist wie der bekannte Burger-Button – also der Button bestehend aus drei waagerechten Strichen, der mitteilt, dass es noch weitere Menüpunkte/Optionen gibt.

4. Genau genommen gibt es drei Widgets. Eines für die Abfahrten in der Nähe, eines für die meistverkauften Tickets und eines für die eigenen gekauften Tickets. Veranschaulicht wird lediglich letzteres. Es wäre entsprechend optimaler alle zu erwähnen oder explizit zu benennen, welches Widget beschrieben wird.

5. Die Darstellung an sich ist völlig korrekt. Dennoch ist sie unter Umständen irreführend. Im Screen wird von einer gültigen (also noch nicht abgelaufenen) Fahrkarte ausgegangen. Normalerweise ist es nicht nötig, die gerade gültige Fahrkarte erneut zu kaufen. Scrollt man in der App etwas weiter herunter, findet man den Reiter „Abgelaufene Fahrkarten“, welcher – natürlich variierend von der Smartphone-Größe – nicht sofort ersichtlich ist. Unter diesem Reiter befinden sich alle bisher gekauften Fahrkarten und man kann sie auf die dargestellte Weise nochmals kaufen oder den Kaufbeleg anfordern. Ein Hinweis über dieses Vorgehen wäre also durchaus von Vorteil.

6. Alle ÖPNV-Apps, die hier verglichen werden, verfügen über den Reiter „Abfahrten“. Es scheint also der Normalfall zu sein.
   Auf den ersten Blick sagt der Screen aus, dass dieser Reiter existiert, was entsprechend nicht notwendig ist. Auf den zweiten Blick wird aber erklärt, dass die Abfahrten nach Verkehrsmitteln gefiltert werden können und wie genau. Man sollte also den Fokus auf die Filter legen und nicht die Abfahrten.

Bei der BVG gibt es drei Screens, die die Funktions- und Vorteilsorientierung kombinieren:

1. „BVG App – Persönliche Reiseplanung mit Haltestellen, Abfahrtszeiten und Netzplänen. BVG-Tickets in dieser App kaufen oder unter www.bvg.de beziehen."

2. „Daten verwalten – Daten zur Person oder Rechnungen unter Einstellungen verwalten. Geburtsdatum und E-Mail Adresse verändert der Support: info@bvg.de"

3. „4-Fahrten-Karte – Die 1. Fahrt ist automatisch und sofort gültig – restliche Fahrten befinden sich im Guthaben. Für Kontrollen bitte eingeschaltetes Handy und alle Mitreisenden zusammenhalten."

Schaut man sich die gezeigten Inhalte auch hier genauer an, kann man feststellen:

1. Es ist vermutlich im Vorfeld klar, dass die App Fahrpläne, Haltestellen und Abfahrtspläne beinhaltet. Es ist jedoch nicht falsch, kurz zusammenzufassen und darüber hinaus zu erwähnen, dass die Reiseplanung und Netzpläne ebenfalls beinhaltet sind und Tickets erworben werden können, damit der Nutzer erkennt, dass er die richtige App heruntergeladen hat. Schaut man unter dem Suchbegriff „BVG" im Google Play Store nämlich die Ergebnisse an, werden die Apps „BVG FahrInfo Plus", „BVG Berlin Tickets", „Jelbi" und „BVG Bike" als erstes angezeigt[4].

2. Direkt unter dem Reiter „Einstellungen" sind die erwähnten Punkte „Daten" und „Rechnungen" nicht direkt ersichtlich. Es sind weitere Schritte nötig, um genau an diese Funktionen zu kommen.
   Der Nutzer folgt der Anweisung und findet das angekündigte Resultat nicht. Es wäre also besser entweder den Weg richtig zu beschreiben (in dem Fall der persönlichen Daten wäre das Einstellungen > Ticket Einstellungen > Persönliche Daten) oder zu bebildern.

3. Die 4-Fahrten-Karte ist nur über den Reiter „Tickets" zu finden, wird aber bei der Verbindung nicht zur Auswahl gestellt.
   Unter dem Reiter „Tickets" im Tab „Guthaben" wird erneut darauf hingewiesen, dass das Guthaben über die 4-Fahrten-Karte entsteht und dass die erste Karte direkt nach dem Kauf gültig ist. Einzige Verbesserungsmöglichkeit wäre hier also kurz zu erwähnen, wo das Ticket zu finden ist.

Es fällt auf, dass die On-Boardings teils irreführend sind und teils durchaus ihre Berechtigung haben.

---

[4] https://play.google.com/store/search?q=bvg&c=apps, letzter Aufruf: 16.07.2019 – 17:54 Uhr (Apps die „Berliner Verkehrsbetriebe (BVG)" als Publisher haben)

In Anbetracht dessen, dass auch nach der Umgestaltung versteckte Funktionen existieren werden und die Umgestaltung einen maßgeblichen Mehrwert erzielen soll, wird ein On-Boarding verwendet werden.

Um den KVB und den MVV nicht außer Acht zu lassen, werden diese Stolpersteine folgend auch betrachtet.

Wie bereits erwähnt, verwenden diese Apps kein On-Boarding, obwohl es auch hier Sinn machen würde, um den Nutzer wesentliche oder versteckte Dinge zu erläutern.

Die KVB (Abb. 4) startet direkt mit der Abfrage, ob die App auf den Standort zugreifen darf, ohne dass der Nutzer schon etwas gesehen hat. Danach sieht der Nutzer weiterhin nichts von App selbst, da er gleich zu Beginn zur Anmeldung/Registrierung aufgefordert wird. Einzig gut ist, dass der Anmelde-/Registrierungsprozess übersprungen werden kann und man so gleich auf die App zugreifen kann.

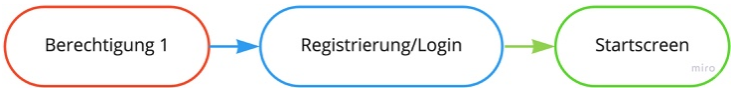


Abbildung 5 - App-Start Verlauf der KVB App

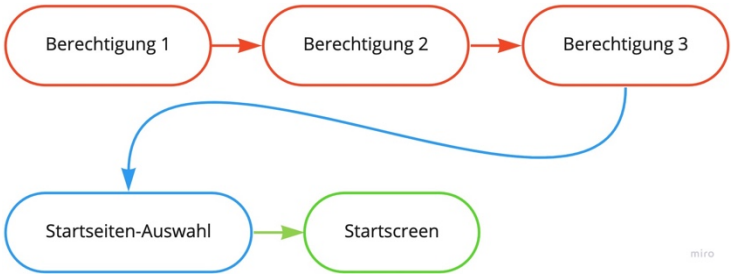


Abbildung 4 - App-Start Verlauf der MVV App

Der MVV (Abb. 5) fragt ebenfalls zu Beginn Berechtigungen ab, sogar drei – den Zugriff auf den Standort, die Medien und die Kontakte. Darauffolgend wird der Nutzer aufgefordert eine Startseite zu wählen, obwohl er auch wie beim KVB noch nichts Konkretes von der App gesehen hat. Die Funktion an sich ist keine schlechte Überlegung, würde jedoch an anderer Stelle – beispielsweise in den Optionen – eventuell besser aufgehoben sein. In diesem Falle bremst die Funktion beim ersten Start aus und der Nutzer wird gleich zu einer Entscheidung gezwungen, über die er sich vermutlich noch nie Gedanken gemacht hat.

Verbindungssuche

Schauen wir uns die Verbindungssuche vom jeweiligen Hauptbahnhof zu einer bekannten Sehenswürdigkeit und die Darstellungen der Verbindungen genauer an. Als Sehenswürdigkeiten der einzelnen Städte wähle ich die von den Städten selber empfohlenen aus – für Hamburg die Elbphilharmonie[5], für Berlin den Reichstag[6], für Köln den Kölner Dom[7] und für München die Frauenkirche[8]. Die Komplexität der Verbindung wird natürlich variieren. Das hält aber nicht davon ab, sich ein Bild machen zu können, ob man die Darstellung sofort versteht oder sich etwas hineindenken muss.

Die Wege, um zu einer Verbindung zu gelangen, sind bei allen Apps nahezu identisch. Kleine Abweichungen existieren hier und da, auf die folgend eingegangen wird.

Der allgemeine Ablauf baut sich wie folgt auf (Abb. 6):

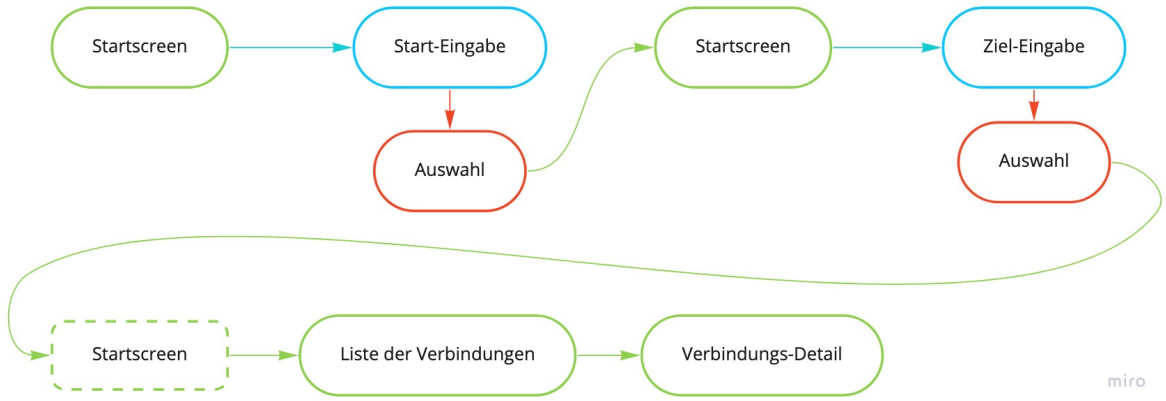


Abbildung 6 - Verbindungssuche-Ablauf

Die App des HVV und des KVB zeigen nach der Ziel-Eingabe erneut den Startscreen, um den Suchstart manuell zu bestätigen. Der BVG und der MVV überspringen den Startscreen und starten die Suche automatisch nach der Zieleingabe.

[5] „TOP 10 HAMBURG SEHENSWÜRDIGKEITEN“, https://www.hamburg.de/sehenswuerdigkeiten/, letzter Aufruf: 12.07.2019 – 13:40 Uhr

[6] „Top 10 Sehenswürdigkeiten in Berlin“, https://www.visitberlin.de/de/top-10-sehenswuerdigkeiten-berlin, letzter Aufruf: 12.07.2019 – 13:40 Uhr

[7] „Sehenswertes in Köln“, https://www.koeln.de/tourismus/sehenswertes, letzter Aufruf: 12.07.2019 – 13:40 Uhr

[8] „Die Top 20 Sehenswürdigkeiten in München“, https://www.muenchen.de/sehenswuerdigkeiten/bildergalerien/stadtrundfahrten-touren/top-20-sehenswuerdigkeiten.html, letzter Aufruf: 12.07.2019 – 13:40 Uhr

Der BVG hat darüber hinaus die Besonderheit, dass auf der „Home“-Seite gestartet wird, welche aus kleineren Widgets besteht wie unter Anderem einer vereinfachten Suchmaske für die Verbindung und Schnellzielen. Nach dem Anklicken des Starts/Ziels und entsprechender Auswahl gelangt man automatisch in den „Fahrplan“-Reiter, welcher die ganze Verbindungssuche beinhaltet.

Beide Methoden – also direkter Suchstart oder manuelles Bestätigen – haben ihre Vor- und Nachteile und Kriterien, die sie dadurch erfüllen müssen.

Wenn der Nutzer nach der Ziel-Eingabe zurück zum Startscreen gelangt, kann er die Uhrzeit noch ändern, bevor die Suche startet. Auch ein versehentlich falscher Start oder falsches Ziel können noch geändert werden. Der Nachteil besteht jedoch darin, dass die Verbindungssuche nicht ganz so schnell erfolgt, da ein weiterer Klick benötigt wird. Bei dieser Variante muss sichergestellt werden, dass sofort klar ist, wo und dass die Suche selber gestartet werden muss.

Bei der anderen Variante – dem automatischen Suchstart – ist der Vorteil klar die Schnelligkeit. Sobald das Ziel gesetzt wurde, werden die verfügbaren Verbindungen angezeigt. Sollte jedoch versehentlich das falsche Ziel aus der Liste gewählt worden sein oder die Uhrzeit noch nicht stimmen, gibt es keine sofortige Möglichkeit zum Korrigieren. Entsprechend muss hier sichergestellt werden, dass trotz Such-Start auf die Schaltflächen zugegriffen werden kann, ohne auf die Fertigstellung der Suche warten zu müssen. Weiter ist es von Vorteil, wenn die Suchmaske – Zeit, Start, Ziel – weiterhin die gleiche Optik besitzt wie beim Beginn.

Da versehentlich falsche Eingaben in der zweiten Variante ebenso schnell wie auf dem Startscreen bearbeitet werden können, sofern der Zugang zu den Schaltflächen weiter gewährleistet wird und sich nicht sehr von dem vorherigen Schritt unterscheidet, hat die erste Variante im Vergleich zur zweiten keinen Vorteil. Zur Umgestaltung wird daher die zweite Variante des automatischen Suchstarts verwendet werden, da kein Nachteil entsteht.

Die Start- und Ziel-Auswahlen funktionieren in allen Apps auf die gleiche Art (Abb. 7).

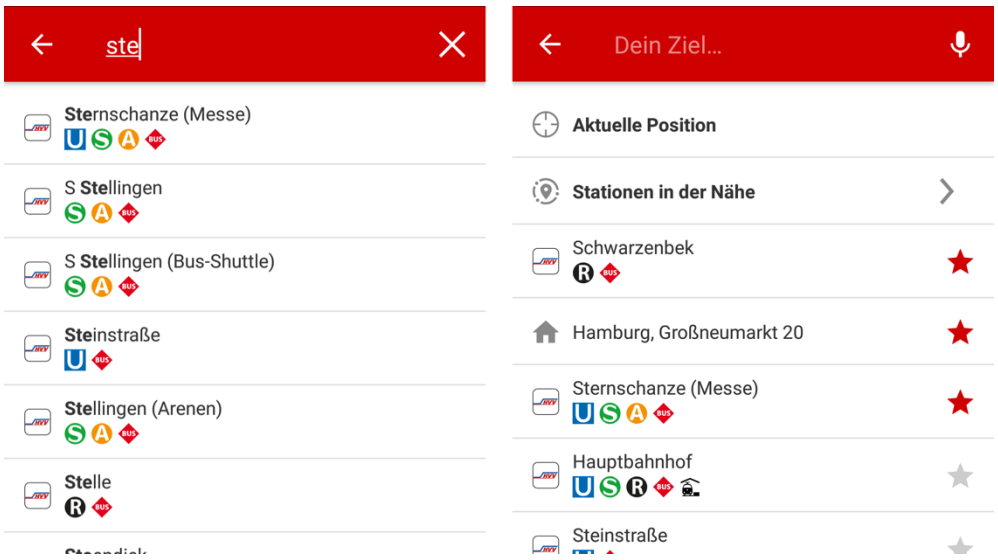


Abbildung 7 - Start-/Zieleingabe der HVV App

Man kann Haltestellen oder Adresse eingeben und anschließend aus einer Liste von Punkten, den zutreffenden wählen. Die Auswahl funktioniert wie eine Suchfunktion, die beim Tippen die Liste entsprechend der Eingabe filtert.

Dadurch, dass die Eingabe auf diese Weise nicht gänzlich frei erfolgen kann, wirken sich Tippfehler nicht auf das Ergebnis aus oder müssen nach einer Fehlermeldung beim Suchstart verbessert werden. Es wird also sichergestellt, dass die gewünschten Start- und Ziel-Punkte existieren. Auch müssen die Haltestellennamen oder Adressen nicht zwingend komplett eingeben werden, da die Liste bei jedem Tastendruck präziser gefiltert wird.

Der HVV sowie die BVG ermöglichen statt der Eingabe per Tastatur auch die Spracheingabe, was der Barrierefreiheit zugutekommt.

Alle der vier Apps haben zusätzlich in der Auswahl einen Verlauf der letzten angefragten Orte, bevor man eine Eingabe tätigt. Alle Orte in diesem Verlauf können über ein Stern-Icon rechts neben dem Bezeichner als Favorit markiert werden und befinden sich anschließend (vor einer Eingabe) immer oben im Verlauf, um schnell auf sie zugreifen zu können.

In dem Punkt der Start- und Ziel-Auswahl kann man also durchaus von einer guten UX sprechen.

Die Verbindungssuche kann bei allen Apps durch weitere Optionen verfeinert werden. Welche Einstellungsmöglichkeiten es gibt, hängt stark von der App ab.

Um eine weitestgehend auf den Nutzer maßgeschneiderte Verbindungen zu bekommen, sollten Optionen wie Barrierefreiheit, Umsteigezeit, Umsteigeanzahl, Gehgeschwindigkeit und ein Verkehrsmittel-Filter gegeben sein. Aus den Gründen, dass Personen mit Kinderwagen oder Rollstuhl eine ÖPNV-App ebenso nutzen wie Menschen, für die schnelles Gehen und Treppen kein Problem darstellen. Ebenso werden Touristen die App auch für kurze Zeit benutzen, ohne sich an den Bahnhöfen und in der Umgebung von Haltestellen auszukennen.

Das bedeutet wiederum, dass eine Umsteigezeit von 2 Minuten bei fehlenden Ortskenntnissen oder einer Haltestelle ohne Aufzüge für Rollstuhlfahrer zur Folge hätte, dass nicht nur die Bedienung der App zu Frustrationen führt, sondern die gesamte App.

Alle diese Punkte erfüllen nur der MVV und die BVG. Der HVV beschränkt sich auf die Option „Rollstuhlgerecht“ und die Möglichkeit, die Verkehrsmittel aus- und abzuwählen. Der KVB bietet lediglich an zwischen „Beste Route“, „Wenig Umstiege“ oder „Kurze Fußwege“ zu wählen, von welchen nur eins gewählt werden kann, sowie die Möglichkeit die Verkehrsmittel und Mobilitätsangebote (wie Car2go und DriveNow) auszuwählen.

Der HVV und die BVG haben die bereits kurz erwähnten Schnellziele. Beim erstmaligen Klick auf das Haus- oder Krawatten- (beziehungsweise Aktentaschen) Symbol, wird man aufgefordert, diesem Schnellziel eine Adresse oder Haltestelle hinzuzufügen. Diese zwei Schnellziele tragen automatisch ausgefüllt die Namen „Zuhause“ und „Arbeit“. Wenn die Information gespeichert wurde und man anschließend auf das Symbol klickt, öffnet sich automatisch die Eingabe für den Start.

Nach kurzem Nachdenken wird man erkennen, dass das Ziel das ausgewählte Schnellziel sein wird und man zum Suchstart noch den Start eingeben muss.

Auf den ersten Blick fehlt aber jeglicher Kontext. Man wählt ein Schnellziel und soll ohne Feedback, was gerade passierte, etwas eingeben. Zugegeben, dass dieser Ablauf durchaus intuitiv sein kann, kann er jedoch ebenso zu Verwirrung führen.

Ein Nutzer, dem der Ablauf logisch erscheint, wird sich nicht von einem kleinen Hinweis gestört fühlen. Umso größer ist der Mehrwert eines Feedbacks für einen Nutzer, der nicht genau weiß, was gerade passiert ist.

Ein weiterer nicht gut gelöster Punkt ist, dass trotz vorheriger Start-Eingabe bei der Schnellzielwahl, der Start erneut eingegeben werden muss. Ungeachtet dessen, ob der Startpunkt bereits gewählt wurde, öffnet sich automatisch die Start-Auswahl.

Es wäre gegebenenfalls sinnvoller, beim Klick auf ein Schnellziel den Zielort auf demselben Screen auszufüllen, oder die Suche zu starten, falls ein Startpunkt bereits gewählt ist.

Verbindungsliste

Nachdem alle Such-Eingaben getätigt wurden, gelangt man in allen Apps zu einer Liste mit verfügbaren Verbindungen.

Folgend eine tabellarische Auflistung der dargestellten Informationen in der Verbindungsliste der vier Apps:

Dargestellte Information	HVV	BVG	KVB	MVV
Abfahrtszeit	✓	✓	✓	✓
Verspätung	✓	✓	(✘)	(✘)
Ankunftszeit	✓	✓	✓	✓
Aufteilung der Linien	✓	✓	✓	✓
Reisedauer	✓	✓	✓	✓
Umsteigeanzahl	✓	✓	✘	✘
Meldungen	✓	✓	(✘)	✓
Ticketpreis	✓	✓	✘	✓
Ticketkauf	✓	✘	✓	✓
Zurückzulegender Fußweg	✘	✘	✓	✘
Information über Abfahrtszeit und Haltestelle sollte ein Fußweg zur Starthaltestelle erfolgen müssen	✓	✓	✘	✘
Visualisierung der zeitlichen Verteilung der einzelnen Linien	✓	✓	✘	✘
Linienbezeichner und -farben	✓	✓	✓	✓
Linienrichtung	✘	✘	✓	✘

Die umklammerten Eingaben bedeuten, dass die Darstellung nicht erfolgt, es aber nicht mit Sicherheit gesagt werden kann, dass die Information zur Darstellung nur nicht verfügbar ist.

Alle der aufgelisteten Informationen sind von Nutzen. Um alle auch entsprechend darzustellen, muss die Gestaltung gewährleisten, dass sie erkennbar und identifizierbar sind, ohne überladen und unübersichtlich zu wirken.

Einige davon können durchaus zusammengefasst werden. Beispielsweise können der Ticketpreis und der Fahrkartenkauf auf eine Schaltfläche zum schnellen Kaufen reduziert werden, so wie es auch bei allen bis auf den BVG der Fall ist.

Die Informationen müssen nicht zwingend viel Platz einnehmen. Die Verspätungen können beispielsweise direkt neben die Abfahrts-/Ankunftszeit gesetzt werden, Meldungen können durch ein Symbol angekündigt sein (und nur auf Bedarf angezeigt werden) und die Länge des Fußwegs kann an dergleichen Stelle stehen, wie der Linienbezeichner.

Die BVG und der HVV stellen die größte Menge der genannten Informationen dar und sind nicht zu unübersichtlich. Für die Umgestaltung wird sich entsprechend an diesen orientiert.

Verbindungsdetail

Nach einem Klick in der Liste auf ein Element, wird die Verbindung im Detail angezeigt. Folgend eine weitere tabellarische Ansicht darüber, welche Informationen zu der gewählten Verbindung angezeigt werden:

Dargestellte Information	HVV	BVG	KVB	MVV
Strecken-Kurzinfo	✘	✓	✘	✓
Verkehrsmittel	✓	✓	✓	✓
Linie	✓	✓	✓	✓
Richtung	✓	✓	✓	✓
Gleis	✓	✓	✓	✓
Start(-haltestelle)	✓	✓	✓	✓
Zielhaltestelle	✓	✓	✓	✓
Abfahrtszeit	✓	✓	✓	✓
Ankunftszeit	✓	✓	✓	✓
Zwischenhaltanzahl	✓	✓	✓	✓
Zwischenhalte	✓	✓	✓	✓
Meldungen	✓	(✘)	(✘)	✓
Verspätungen	✓	✓	(✘)	✓
Verbindungsalternativen	✓	✓	✘	✘
Fußweg Distanz	✓	✓	✓	✓
Fußweg Dauer	✓	✓	✓	✓
Fußweg Karte	✓	✓	✘	✘
Fußweg Beschreibung	✘	✓	✘	✓
Fußweg-Art	✘	✓	✓	✘
Wartezeit	✘	✘	✘	✓
Ticket-(Kurz-)Infos	✓	✓	✘	✓

Grundlegende Informationen werden von allen Apps angezeigt. Auf diese kann entsprechend auch nicht verzichtet werden.

Bei weitergehenden Funktionen und Informationen wie beispielsweise zum Fußweg, der zurückgelegt werden muss, zusätzlich zur Karte Richtungsanweisungen anzuzeigen oder Fußwege als „Umstieg“ oder „Fußweg“ zu kategorisieren gehen die Apps auseinander.

Inwieweit machen diese zusätzlichen Informationen also Sinn?

Streckenkurzinfo

Die erneute Anzeige der Streckenkurzinfo beim MVV (Abb. 8) macht insoweit Sinn, dass das Verbindungdetail initial mit einer Karte aufgerufen wird und zunächst nur die Informationen des Streckenverlaufs, der Kurzinfo und der Fahrkarte angezeigt werden. Um also gleich zu Beginn zu wissen, um welche Verbindung es sich handelt, ist die Kurzinfo hier klar von Vorteil.

Da der Streckenverlauf jedoch optional sein könnte, da er keine relevanten Informationen wiedergibt (Zu welcher Haltestelle muss ich zunächst gehen? Wann fährt die Bahn dort ab? Wie heißt die Zielhaltestelle?), könnte man beim Aufrufen der Verbindung darauf verzichten und somit auch auf die Kurzinfo.

Beim BVG hingegen ermöglicht die Kurzinfo nach rechts und links zu swipen und auf diese Weise durch die einzelnen Verbindungen zu navigieren. Zwar ist nicht ersichtlich, welche Verbindung davor oder dahinter liegt, aber eine Geste ist in diesem Fall schneller, als zurück zu navigieren und die nächste Verbindung auszuwählen.

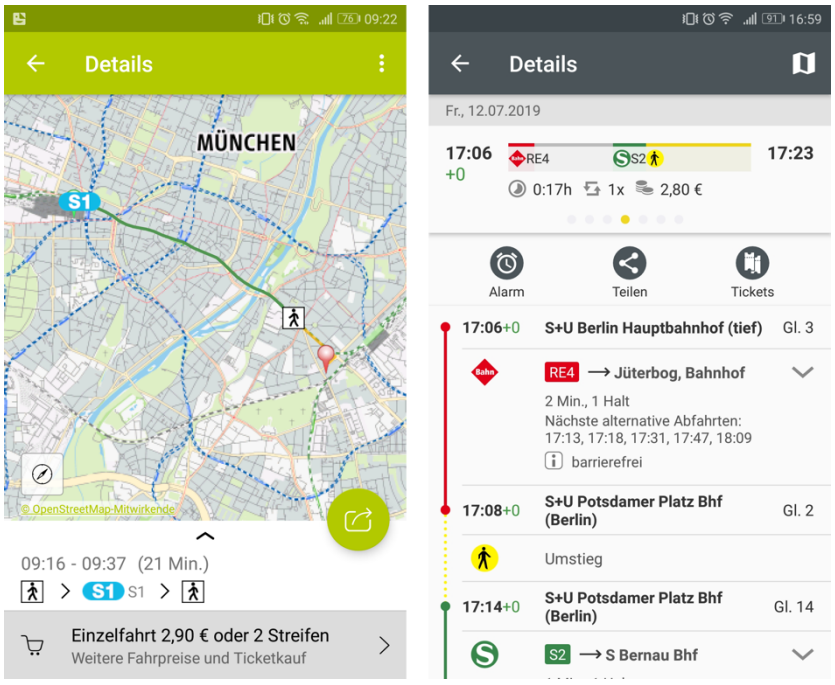


Abbildung 8 - MVV (links) und BVG (rechts) Verbindungsdetail

Verbindungsalternativen

Die Information darüber, in welchem Takt Alternativen verkehren, kann insoweit von Nutzen sein, dass bei einer verpassten Verbindung nicht zwingend eine neue Suche gestartet werden muss. Anzumerken wäre jedoch, dass bei mehreren Umsteigen dieser Takt nicht für jede Linie gleich ist. So kann ein Bus alle 5 Minuten fahren, die Anschluss S-Bahn aber nur alle 20. Wenn man diese Anschluss S-Bahn also verpasst, kann es durchaus der Fall sein, dass es dadurch eine alternative Verbindung gibt, die schneller ist, als 15 Minuten auf den Anschluss zu warten.

Wenn man hingegen nicht umsteigen muss, ist diese Information natürlich klar von Vorteil.

Fußweg Karte + Beschreibung

Leider verzichten der KVB sowie der MVV auf eine Karte für den Fußweg. Der MVV bietet jedoch zumindest eine Wegbeschreibung an – beim KVB hingegen fehlen jegliche Informationen zum Weg. Wie man also von seinem Standort zu seiner Haltestelle gelangt, muss eigenständig in Erfahrung gebracht werden. Auf die Option der Karte sollte also nicht verzichtet werden.

Für die Umgestaltung ist geplant, dieses Problem mit einem Google Maps Link zu überbrücken, aus dem einfachen Grund, die Einbindung von Open Maps zu umgehen und direkt auch auf eine Navigation zurückgreifen zu können.

Der BVG zeigt in diesem Fall sogar Karte sowie Wegbeschreibung an. Die Wegbeschreibung selber kann sich als sinnvoll erweisen, wenn eine schlechte Datenverbindung besteht oder auf das Laden einer Karte verzichtet werden möchte. In Sachen Datenverbrauch ist diese Möglichkeit entsprechend vorteilhaft.

Fußweg Art

BVG und KVB zeigen zusätzlich beim Fußweg an, ob es sich um einen Fußweg oder einen Umstieg handelt.

Da „Umstieg“ in diesem Falle so aufzufassen ist, dass es sich um den Fußweg an derselben Haltestelle handelt, kann in den meisten Fällen auch keine Wegbeschreibung (textuell oder als Karte) erfolgen.

Um sich nicht über die fehlende Karte oder Beschreibung zu wundern, kann der Begriff „Umstieg“ anstelle der groben Bezeichnung „Fußweg“ verwendet werden.

Wartezeit

Allein der MVV liefert die Information, wieviel die Wartezeit zwischen Verbindungen beträgt. In den anderen Apps muss darauf selber geachtet werden.

Wenn ein Bus beispielsweise um 10:45 Uhr ankommt, man 3 Minuten zur S-Bahn benötigt und diese um 10:55 Uhr startet, beträgt die Wartezeit also 7 Minuten. Der MVV gibt dies so an (3 Minuten Fußweg und 7 Minuten Wartezeit). Die anderen Apps zeigen die 3 Minuten Fußweg und geben den Startzeitpunkt der S-Bahn an. Für einen schnellen Überblick ohne kurzes Rechnen, ist die Wartezeit-Information also sinnvoll.

Ticket (Kurz-) Info

Der KVB zeigt als einzige App nicht die vorgeschlagenen Tickets, deren Bezeichnung oder Preis an. Beim Klick auf „Zum Ticketkauf“ werden jedoch nur die Fahrkarten angezeigt, die zu dieser Verbindung passen, was Fehlkäufe verhindert.

Für Personen, die ein Ticket kaufen müssen, ist prinzipiell unerheblich, wie teuer eine Fahrkarte ist, da sie so oder so gekauft werden muss. Dennoch würde eine kleine Information niemanden stören, sondern eher den Eindruck erwecken, dass man die richtige Fahrkarte kaufen wird. Die Auswahl der Fahrkarte weist nämlich nirgends daraufhin, dass sie auf die Verbindung abgestimmt ist.

Natürlich besteht ein gewisses Vertrauen vom Nutzer, dass man auch ohne Kenntnisse die richtige Fahrkarte vorgeschlagen bekommt. Dieses Vertrauen sollte aber auch weiterhin gestützt werden, indem klar kommuniziert wird, dass genau das passiert, was der Nutzer erwartet.

Fahrkartenkauf

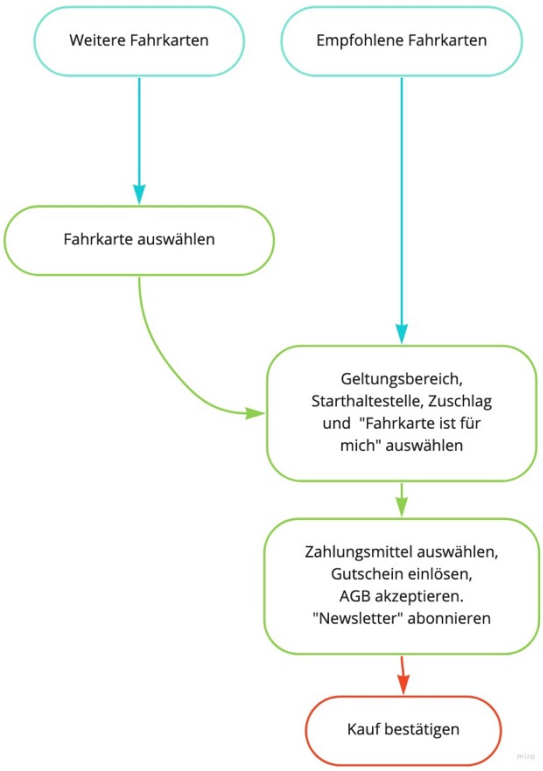


Abbildung 9 - HVV - Verlauf des Ticketkaufs aus einer Verbindung heraus

Der Ablauf des Kaufs einer Fahrkarte in der App des HVV (Abb. 9) kann entweder über die Verbindung überfolgen oder separat unter „Fahrkarten“. Aus einer Verbindung heraus hat natürlich den klaren Vorteil, dass der Geltungsbereich und die Fahrkartenart vorgeschlagen werden, sodass auf Anhieb das richtige Ticket gekauft werden kann.

Der Ablauf selber ist optimierbar und manche Zwischenschritte könnten eingespart werden.

Wenn eine empfohlene Fahrkarte gewählt wird, besteht die Möglichkeit den Geltungsbereich, die Starthaltestelle, den Zuschlag und für wen die Karte bestimmt ist zu ändern (Abb. 10).

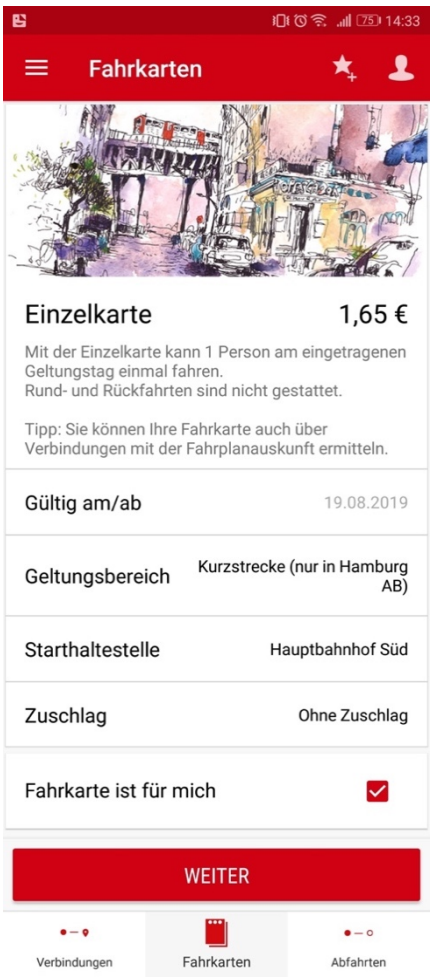


Abbildung 10 - HVV Fahrkartenkauf

Als Beispiel für die Verbindung vom Hamburger Hauptbahnhof zur Elbphilharmonie (Eingang) würde die Einzelkarte gewählt. Die Verbindung benötigt weder einen SchnellBus noch einen anderen Geltungsbereich. Den Geltungsbereich zu ändern oder den Zuschlag hinzu zu wählen ist also nicht nötig. Einzelkarten sind darüber hinaus nicht personengebunden, was die Möglichkeit, das Ticket für jemand anderen zu kaufen ebenfalls überflüssig macht.

Das Bild am Anfang des Screens hat darüber hinaus keinen Bezug zum Ticket oder eine Funktion und verschwendet daher eher Platz (Abb. 10).

Die Möglichkeit des Ticketkaufs aus einer Verbindung heraus, sollte dazu dienen, eine gültige Fahrkarte zu kaufen, ohne die Regelungen des HVV zu kennen. Die Optionen für die Fahrkarte also nochmals anpassen zu wollen, bestünde in diesem Fall nicht. Wenn der Nutzer die Regelungen vor der Auswahl nicht kannte, wird er sie auch jetzt nicht kennen. Wenn er sie kennt, muss er sie ebenfalls für die Verbindung nicht ändern.

Auf diese Weise könnte der Kauf entsprechend beschleunigt werden und sich auch nutzerfreundlicher anfühlen. Sobald der Nutzer die Möglichkeit hat, über die Optionen nachzudenken und in Frage zu stellen, kann er unter Umständen beginnen zu zweifeln, ob die Voreinstellungen auch korrekt sind.

Sinnvoller wäre es also, die Fahrkarte wählen zu können, zu entscheiden, wie sie bezahlt werden soll und sie daraufhin kaufen zu können.

Weiter müssen nach der Bestätigung, welche Fahrkarte gekauft werden soll, weitere Eingaben getätigt werden, wenn auch optional.

So kann ein Gutschein eingelöst werden, die AGB müssen akzeptiert werden und ein Newsletter kann abonniert werden.

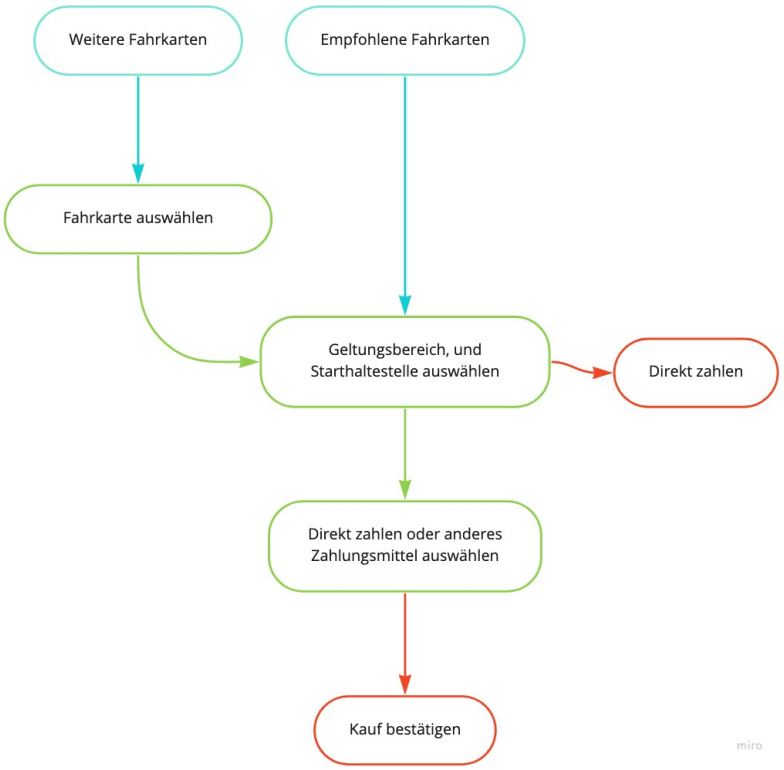


Abbildung 11 - BVG - Verlauf des Ticketkaufs aus einer Verbindung heraus

Beim BVG (Abb. 11) verhält es sich ähnlich, bis auf die Möglichkeit in der Bestätigung der Fahrkarte direkt zum Bezahlvorgang zu gelangen und dass bei den weiteren Angaben danach, die AGB (mit Hinweis) beim Kauf automatisch akzeptiert werden und weder Gutscheine eingelöst noch Newsletter abonniert werden können.

Die BVG beschleunigt den Ticketkauf also bereits, obwohl er prinzipiell auf die gleiche Art funktioniert. Die Fahrkarte kann jedoch wie beim HVV ebenfalls noch angepasst werden.

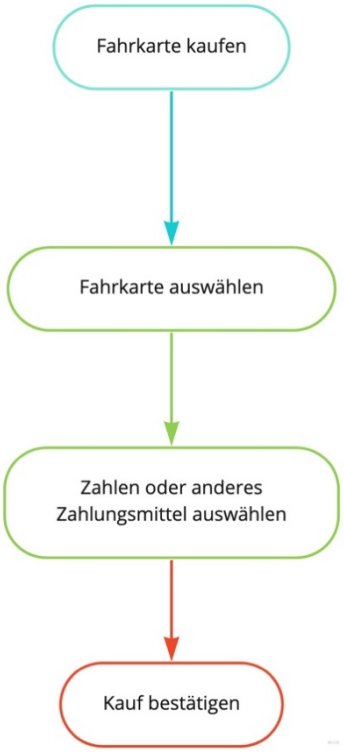


Abbildung 12 - KVB - Verlauf des Ticketkaufs aus einer Verbindung heraus

Wie bereits erwähnt, schlägt der KVB in der Verbindung direkt keine Fahrkarte vor, lässt aber nur aus den für die Verbindung zutreffenden Fahrkarten auswählen.

Anschließend wird das Zahlungsmittel gewählt und bezahlt (Abb. 12) – AGB werden auch hier mit einem Hinweis beim Bestätigen automatisch akzeptiert.

Der KVB hat also den schnellsten Weg, das passende Ticket zu erwerben und wird auf diese Weise als Vorbild für die Umgestaltung dienen.

Abbildung 13 - MVV - Verlauf des Ticketkaufs aus einer Verbindung heraus

Im Verbindungsdetail des MVV wird auf der Ticket-Sachaltfläche ein Vorschlag für eine Fahrkarte gemacht und deren Preis angegeben. Nach einem Klick darauf (Abb. 13), muss eine Fahrkarte ausgewählt werden. Die aufgelisteten Fahrkarten besitzen kaum Informationen und der Vorschlag wird nicht wiederholt.

Als Nutzer könnte man sich unter Umständen schnell verloren fühlen, da nicht klar ist welche Fahrkarte nun zu wählen ist. Eine Möglichkeit bestünde darin, zurück zu navigieren und sich den Vorschlag zu merken und diesen auszuwählen. Danach können wie beim HVV und der BVG Starthaltestelle und Geltungsbereich geändert werden und danach das Zahlungsmittel ausgewählt und bestätigt werden.

II.III. Verbesserung der momentanen Probleme

Nachdem die Schwierigkeiten erkannt wurden, können Lösungen erarbeitet werden, die später im Design berücksichtigt werden.

Umgestaltungskriterien aus dem Vergleich

Es ergeben sich nach dem Vergleich mit anderen ÖPNV-Apps folgende Anpassungen für die HVV App:

On-Boarding:

Das On-Boarding wird etwas verkürzt und stellt die Funktionen richtig dar.

Da die Umgestaltung nicht alle Funktionen erfasst, ist es einfach, den Umfang zu kürzen. Das eigentliche Problem besteht darin, dass die momentanen Erklärungen teilweise unzutreffend sind, weshalb darauf geachtet wird, die Funktionen unmissverständlich darzustellen.

Verbindungssuche:

Die Suchmaske bleibt beim Start der Suche an derselben Stelle.

Die momentane Eingabe springt nach dem Suchstart etwas nach oben, da die Karte entfernt wird, was bei einer unbeabsichtigten Eingabe zur Folge hat, dass man bei schneller Korrektur mit hoher Wahrscheinlichkeit an die falsche Stelle klickt. Es soll also gewährleistet werden, dass die Eingabe an der gleichen Stelle platziert ist.

Die Optionen für die Verbindungen werden umfangreicher.

Die bestehenden Optionen für die Barrierefreiheit umfassen lediglich den Punkt „Rollstuhlgerecht“, der sich aber nur auf Aufzüge bezieht, nicht jedoch auf Umsteigezeiten oder -anzahl. Diese Optionen werden entsprechend ergänzt, sowie die Auswahl der persönlichen Gehgeschwindigkeit und die Priorisierung von Verkehrsmitteln anstelle des einfachen Aus- und Abwählens.

Die Schnellziele werden überarbeitet.

Die Schnellziele erfordern jedes Mal die Eingabe eines Starts, unabhängig davon ob er bereits gesetzt wurde oder nicht. Sie werden soweit geändert, dass sie als Favoriten existieren, die jeweils einen Start (der auch der aktuelle Standort sein kann) und ein Ziel haben und bei Klick diese Verbindungssuche automatisch ausführen. Bei Bedarf kann auch ein eindeutigerer Name vergeben werden. Favoriten und Suchverlauf werden in separaten Tabs existieren, in denen Verbindungen aus dem Verlauf als Favorit hinzugefügt und als solcher bearbeitet werden können.

Verbindungsliste:

Die Distanz des Fußweges wird mit aufgeführt.

Um der Barrierefreiheit weiter zu helfen, wird in der Liste direkt die Distanz des Fußweges mit aufgeführt. Die geschätzte Dauer bleibt ebenfalls bestehen.

Verbindungsdetail:

Die Fußweg-Art (Umstieg/Fußweg) wird eingefügt.

Um Verwirrung mit etwaig fehlenden Karten zu vermeiden, werden die Fußwege in „Umstieg“ und „Fußweg“ geteilt.

Die Wartezeit wird ergänzt.

Für den schnellen Überblick wie viel Wartezeit zwischen Verbindungen liegt, wird diese explizit aufgeführt werden.

Die Navigation des Fußwegs wird mithilfe von Google Maps realisiert.

Um nicht auf Open Maps oder Ähnliches zurückgreifen zu müssen und gleichzeitig eine Navigation zu integrieren, kann der Fußweg mit Google Maps geöffnet werden.

Fahrkartenkauf aus einer Verbindung heraus:

Das Bearbeiten der Fahrkarten-Optionen wird entfernt.

Da die Fahrkarte bereits auf die gewählte Verbindung angepasst ist, wird auf das Bearbeiten des Geltungsbereichs und Zuschlags verzichtet, um mögliche Fehlkäufe zu vermeiden, Prozess schneller zu machen und das Vertrauen zu bestärken.

Der Kauf einer Fahrkarte wird beschleunigt.

Die AGB müssen zurzeit manuell bestätigt werden. Für die Umgestaltung werden diese mit einem Hinweis beim Bestätigen des Kaufs automatisch akzeptiert. Die Ansicht umfasst entsprechend nur noch einen Screen, der die Fahrkarte zusammenfasst, das Zahlungsmittel auswählen und den Kauf per Schaltfläche bestätigen lässt.

User Stories

Im Idealfall bilden die User Stories das Grundgerüst eines App-Konzepts.

Zugrunde liegen diesen meist sogenannte Personas.

Eine Persona ist kein direkter Nutzer, sondern vielmehr ein theoretischer Urtyp eines Nutzers, der dennoch mit einem Namen und einer Beschreibung versehen wird.[9]

In *The UX Book* wird auf A. Cooper hingewiesen, welcher in seinem Buch *The Inmates Are Running the Asylums: why high tech products drive us crazy and how to restore the sanity* beschreibt, dass die allgemeine Denkweise, Designs auf alle Nutzergruppen maßschneidern zu wollen, schlichtweg falsch ist. Stattdessen solle man lieber einen kleinen Teil dieser Gruppen nehmen und diese zufriedenstellen – und weiter sogar einen noch kleineren Teil heranziehen, um ihn völlig zu begeistern. Diese Art sei mit Abstand besser, als jeden Nutzer nur halb zufriedengestellt zurückzulassen.[10]

Für diesen Ansatz sind Personas essentiell. Ebenso helfen sie bei Diskussionen, die das Design betreffen. Dazu heißt es in *The UX Book* (Seite 267), *Cooper* referenzierend:

> *[…] What if the user wants to do X? Can we afford to include X? Can we afford to not include X? How about putting it in the next version? With personas, you get something more like this: "Sorry, but Noah will not need feature X." Then someone says "But someone might." To which you reply, "Perhaps, but we are designing for Noah, not 'someone'."*

[9] Hartson, Rex | Pyla, Pardha S. – *The UX Book – Process and Guidelines for Ensuring a Quality User Experience*, S. 264 – „7.5 USER PERSONAS", Elsevier – Waltham, 2012

[10] Hartson, Rex | Pyla, Pardha S. – *The UX Book – Process and Guidelines for Ensuring a Quality User Experience*, S. 266 – „7.5.2 What are personas used for? Why do we need them?", Elsevier – Waltham, 2012

Da diese Arbeit jedoch auf Personas aus Umfangsgründen verzichtet und sie nicht für Besprechungen benötigt werden, wird an dieser Stelle nicht näher darauf eingegangen. Sie sollten aber durchaus erwähnt werden.

User Stories als solche haben ihren Ursprung im Extreme Programming[11] - eine Art der agilen Softwareentwicklung.

Für gewöhnlich setzen User Stories Personas voraus. In dieser Arbeit wird jedoch bewusst der Fehler begangen, die Persona durch eine Personengruppe auszutauschen. In diesem Falle die Gruppen „Pendler“ und „Tourist“.

Würde die gesamte HVV App neu strukturiert und überarbeitet, würden selbstverständlich Personas erstellt. Im Rahmen der Umgestaltung der drei Funktionen (Verbindungssuche, Fahrkartenkauf und Favoriten) genügen hingegen die Gruppen, um ein erstes Bild von der Vorgehensweise zu bekommen und ein grobes Gefühl dafür zu entwickeln.

Eine User Story baut sich immer so auf, dass ein Nutzer etwas möchte, um damit ein Bedürfnis zu befriedigen.

Als Beispiel:

Als Tourist
möchte ich eine Fahrkarte vorgeschlagen bekommen,
um mir keine Gedanken um die jeweiligen ÖPNV-Regelungen machen zu müssen.

oder

Als Pendler
möchte ich Favoriten für meine Verbindungen anlegen können,
um Start und Ziel, welche ich regelmäßige benötige, nicht jedes Mal erneut eingeben zu müssen.

Neben den Anforderungen für die Umgestaltung, die aus dem Vergleich mit anderen ÖPNV-Apps hervorgingen, können User Stories darüber hinaus weiter dazu beitragen.

Basierend auf Bewertungen aus Google Play (Android), dem App Store (iOS) und eigenen Erfahrungen lassen sich einige spezifische User Stories erstellen, ohne Nutzer dazu direkt befragen zu müssen.

Um die Ausarbeitung der User Stories vollständig auszuführen, gehören auch offensichtliche Funktionen dazu.

[11] März 2017, Sherif Mansour, https://medium.com/@sherifmansour/how-weve-destroyed-user-stories-8b36120645c6, letzter Aufruf: 21.08.2019 – 12:28 Uhr

Diese offensichtlichen Funktionen können beispielsweise „... beim Login Name und Passwort eingeben...“ oder auch „... die Möglichkeit zum Löschen...“ sein. Diese Funktionen wirken selbstverständlich, können jedoch schnell vergessen werden. Zudem variieren diese „selbstverständlichen Funktionen“ je nach entwickelnder Person. So kann Entwickler A eine andere Vorstellungen von einer Registrierung haben als Entwickler B oder von Entwickler C auch in Vergessenheit geraten.

Entsprechend wichtig ist es in einer Konzipierung, in der mehrere Personen zusammenarbeiten, selbst kleine Funktionen und Abläufe genau zu definieren. So wird gewährleistet, dass jeder Beteiligte den gleichen Stand der Dinge hat und das gemeinsame Ziel übereinstimmt.

Selbst für den Fall, dass Konzept und Umsetzung alleinverantwortlich erfolgen, besteht der große Vorteil darin, keinen Aspekt vergessen zu können und die Ziele genau definiert zu haben.

Um es für diese Arbeit hingegen kurzzuhalten, wird sich aktiv dafür entschieden, die meisten dieser offensichtlichen Anforderungen auszuklammern. Sollten im später behandelten Design solche Anforderungen hingegen bewusst angepasst werden, wird dies selbstverständlich erwähnt.

Folgend entsprechend eine Auflistung einiger möglicher Stories, die sich lediglich auf die Hauptfunktionen beziehen und vielmehr als Beispiele anzusehen sind:

(aus dem Google Play Store)

1
Als **Pendler** möchte ich **Verkehrsmittel verschieden priorisieren können** um **meinen Weg für mich zu optimieren, ohne manche Verkehrsmittel gänzlich zu entfernen.**

2
Als **Pendler/Tourist** möchte ich **Störungsmeldungen zuverlässig einsehen können** um **nicht erst an der Haltestelle zu merken, dass meine Bahn/mein Bus z.B. ausfällt**

3
Als **Pendler** möchte ich **wirklich optimierte/schnellste Routen angezeigt bekommen** um **nicht selbst auf Umwegen diese herauszufinden.**

(basierend auf Ergebnissen des Vergleichs)

4
Als **Tourist** möchte ich **die Zeit für Umstiege anpassen können** um **meine Anschlüsse wirklich erreichen zu können**

5
Als **Pendler/Tourist** möchte ich **meine Route auf einen Kinderwagen/Rollstuhl anpassen können** um **mir den Zugang zu Anschlüssen und Verkehrsmitteln zu ermöglichen.**

6

Als

Pendler

möchte ich

Favoriten anlegen können

um

schnell meine of gesuchten Verbindungen aufrufen zu können.

7

Als

Pendler

möchte ich

für meine Favoriten Namen, Start/Ziel, Zwischenhalte und Verkehrsmittel-Prioritäten setzen und bearbeiten können

um

sie genau auf mich abzustimmen.

8

Als

Pendler

möchte ich

meine Favoriten bearbeiten können

um

für kleine Änderungen nicht alles neu eingeben zu müssen.

9

Als

Tourist

möchte ich

meine Fußwege mit Google Maps öffnen können

um

von der Navigation zu profitieren.

(Beispiele für offensichtliche Anforderungen)

10
Als **Pendler/Tourist** möchte ich **möchte ich Start, Ziel, Zeit und Verkehrsmittel-Prioritäten eingeben** um **meine möglichen Routen zu erfahren.**

11
Als **Tourist** möchte ich **in der Übersicht der Routen Abfahrt, Ankunft, Umstiege, Dauer, Linien, ... dargestellt bekommen** um **die für mich beste Route auszuwählen.**

12
Als **Pendler/Tourist** möchte ich **meine letzten Suchen anzeigen lassen** um **sie zu wiederholen.**

II.IV. Umsetzung der Lösungen

Um die erarbeiteten Lösungen umzusetzen, ist es nicht förderlich, direkt mit der Programmierung zu starten.

Besser ist es, sich im Vorfeld Gedanken darüber zu machen, wie sich das Konzept am Ende anfühlen soll, wie alle Informationen verpackt und dargestellt werden und wie der Nutzungsablauf aussehen soll.

Hierfür werden zunächst Wireframes erstellt und anschließend – basierend auf den Wireframes – das Screendesign entwickelt.

Zusammenfassend werden zunächst alle Lösungen aufgelistet, die sich bisher ergeben haben und berücksichtigt werden müssen:

1. Das On-Boarding treffender gestalten und im Umfang leicht reduzieren.
2. Die Sucheingabe nach Suchstart an derselben Stelle behalten.
3. Die Verbindungsoptionen um Verkehrsmittelprioritäten, persönliche Gehgeschwindigkeit, Umsteigezeit und -anzahl erweitern.
4. Schnellziele durch Favoriten (mit Name, Start und Ziel) ersetzen und unter der Sucheingabe in Tabs unterbringen, die „Favoriten“ und „Suchverlauf“ beinhalten.
5. Die Distanz des Fußwegs in der Übersicht ergänzen.
6. Die Art des Fußwegs (Umsteigen oder Fußweg) einführen.
7. Eine Angabe der Wartezeit zwischen Anschlüssen darstellen.
8. Die Navigation des Fußwegs mit Google Maps gewährleisten.
9. Das Bearbeiten der Fahrkarte (beim Kauf aus einer Verbindung heraus) entfernen.
10. Die AGB bei der Kaufbestätigung automatisch akzeptieren und einen Hinweistext hinterlegen.

Wireframes

Zum einen stehen dafür die sogenannten Wireframes zur Verfügung. Diese haben prinzipiell nichts mit der genauen Optik zu tun, sondern schematisieren den Aufbau und Ablauf. Es existieren normalerweise weder Grafiken noch Inhalte, sondern lediglich ein Layout, dass beschreibt, wo etwas liegen. Textblöcke, Bilder, Schaltflächen, usw. werden skizziert und sinnvoll angeordnet.

Sie geben also den groben Rahmen, an dem sich das eigentliche Design orientieren wird.

Sollte man ohne vorheriges Darstellungs-Konzept mit dem Design beginnen, kann durchaus viel Zeit verloren gehen. Es erweist sich zumindest als einfacher und schneller, die Wireframes anzupassen, bis das Layout stimmt, anstatt ein detailliertes Screendesign von Layout bis Schriftgröße anzupassen.

Wie man genau dabei vorgeht, ist nicht vorgeschrieben. Es existieren kleine Hilfen und Anleitungen, die jedoch mehr als Vorschlag aufzunehmen sind als ein direkter Ablauf des Prozesses.

Vermutlich hat jeder Designer andere Präferenzen. Manche schwören auf den digitalen Weg, manche auf Bleistift und Papier. Ich persönlich präferiere auch den zweiten Weg. Am Ende ist es eine Frage der eigenen Vorlieben.

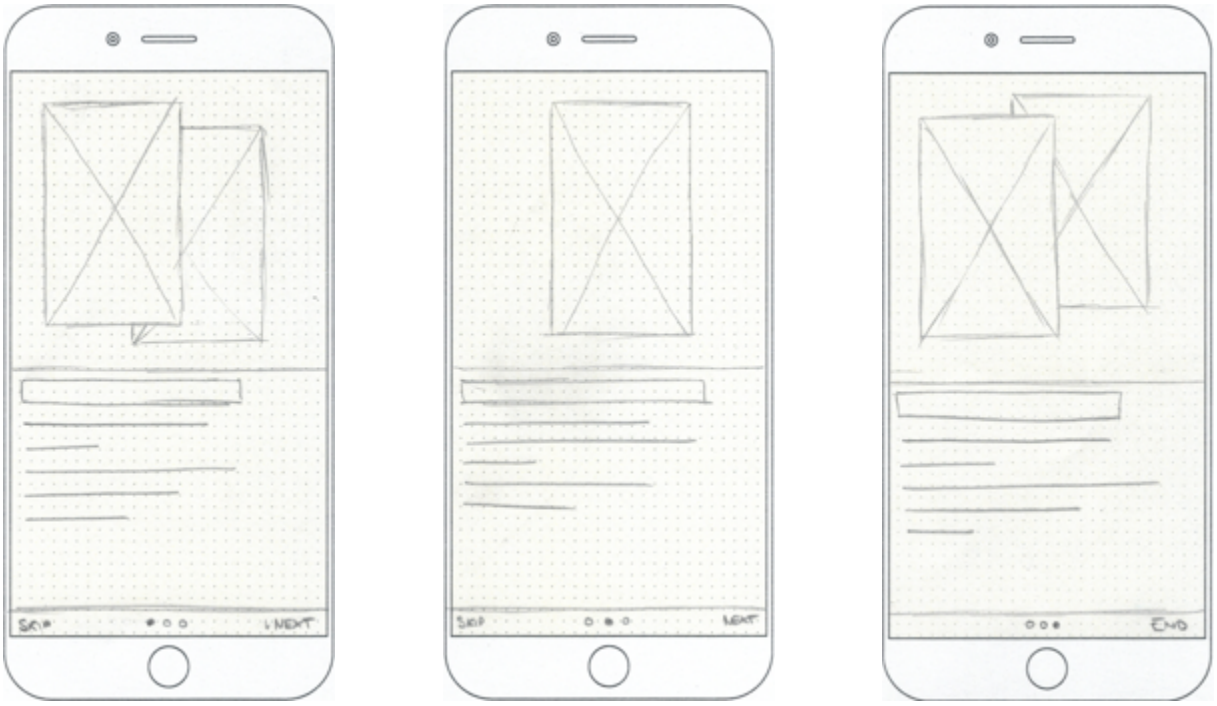
Es existieren ebenso Tools wie beispielsweise Balsamiq, Marvel oder Canva, die speziell für Wireframes konzipiert sind.[12] Im Allgemeinen kann aber für den digitalen Ansatz jedes Grafik- oder Design-Programm von Adobe Photoshop bis Sketch verwendet werden – ebenso je nach Präferenz. Ich selber greife für den digitalen Weg auf Sketch (Mac exklusiv) oder Figma (plattformunabhängig als Web-App mit Kollaborations-Vorteilen) zurück.

Für die Wireframes wird auf die festgelegten Kriterien zurückgegriffen, die sich im Laufe des Vergleichs ergeben haben.

Während der Erstellung können durchaus weitere Dinge auffallen, die sich als suboptimal herausstellen. Die werden entsprechend erwähnt.

[12] Dezember 2018, Maria Myre, https://zapier.com/blog/best-wireframe-tools/, letzter Aufruf: 19.08.2019 – 10:12 Uhr

On-Boarding



Das On-Boarding kann nur bedingt umgesetzt werden, könnte aber diesen Aufbau haben.

Da nicht die gesamte App umgestaltet wird, sondern nur Teilaspekte und sie damit nicht als Ganzes betrachtet werden kann, können die Erklärungen in diesem Falle nur auf die betrachteten Funktionen angewendet werden. Das macht es natürlich einfach das On-Boarding kurz zu halten.

Im Screendesign wird daher nicht näher darauf eingegangen.

Um jedoch beispielhaft zu nennen, welche Funktion näher erläutert werden sollte, wären das die Favoriten (wie lege ich neue an, wie sind sie zu verwenden und welchen Vorteil bringen sie). Alle weiteren Funktionen sind unverändert und nicht versteckt.

Später – im Kapitel der Umsetzung – wird erläutert, wie ein On-Boarding umgesetzt werden kann.

Verbindungssuche, -liste und -detail

Neben den Tabs und dem Entfernen der Karte (um die Sucheingabe nach Suchstart am selben Platz zu behalten), wird die Leiste, die Zeitangabe, Optionen und Suchstart ermöglicht, unterhalb der Eingabe eingefügt.

Im logischen Verlauf wird der Start gesetzt, danach das Ziel gewählt, gegebenenfalls die Uhrzeit gewählt und die Optionen festgelegt. Danach wird die Suche gestartet.

Die Anordnung in der Leiste folgt der Leserichtung – von links nach rechts.

In der derzeitigen App befindet sich diese Leiste über der Eingabe und bricht mit dem regulären Ablauf und kann dadurch kurz übersehen werden.

Die Eingabe erhält keinerlei Änderungen. Zwar könnten die Standortbestimmung sowie der Richtungswechsel (bei dem Start und Ziel getaucht werden) vom Nutzer durch die Anordnung mit der jeweiligen Eingabe verbunden werden, jedoch ist dieser Gedanke nicht so abwegig, wie er scheint. Die Standortbestimmung bezieht sich durchaus auf den Start – der aktuelle Standort wird dadurch als Start gesetzt – und der Richtungswechsel würde das Ziel nach oben setzen und somit Start und Ziel tauschen.

Die Sucheingabe bleibt wie beschrieben an derselben Stelle. Daneben wird die Möglichkeit, die Darstellung für die Liste zu ändern entfernt, da sie keinen Vorteil bietet. Es sei anzumerken, dass vor dem Update auf das jetzige Design die Darstellung nur die Spalten-Anzeige existierte. Es ist vermutlich als ein Überbleibsel anzusehen, dass mittlerweile jedoch entfernt werden kann.

Daneben wird (später im Screendesign zu erkennen) die Möglichkeit frühere oder spätere Verbindungen optimiert. Und zwar so, dass sich am Anfang der Liste eine „früher“-Schaltfläche und am Ende der Liste eine „später“-Schaltfläche befindet, da somit am Anfang schon erkannt wird, ob die erste Verbindung zu spät beginnt und gehandelt werden kann und am Ende der Liste, sollte die letzte angezeigte Verbindung zu früh sein.

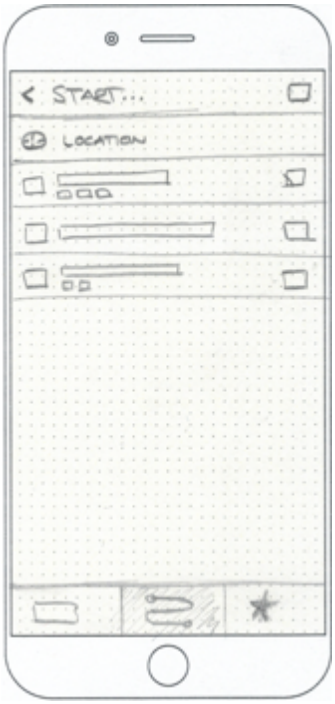
Momentan sind beide Funktionen am Ende der Liste und verweisen nach links (früher) und rechts (später).

Die einzelnen Verbindungselemente ändern sich leicht im Layout, da eine Information hinzukommt, was später im Screendesign aber noch genauer definiert wird.

Die Optionen werden erweitert und ausgerollt. Momentan kann ein Zwischenhalt angegeben werden, die Verkehrsmittel separat geöffnet werden und eine „Rollstuhlgerecht“ Checkbox aus-/abgewählt werden.

Im Nachhinein wurde in diesem Wireframe der Zwischenhalt vergessen sowie die Möglichkeit die Optionen zurückzusetzen - das sei zu entschuldigen, wird im Screendesign aber mit aufgenommen.

Die Verkehrsmittelprioritäten werden nicht separat geöffnet, sondern hier dargestellt sowie die Optionen für die Barrierefreiheit.



Die Suche von Start sowie Ziel verändert sich nur dahingehend, dass die Schaltfläche „Stationen in der Nähe“ entfernt wird.

Die Ermittlung einer Verbindung über die aktuelle Positionen navigiert automatisch zur nächstgelegenen Station inklusive der Wegbeschreibung.

Diese Möglichkeit für Stationen in der Nähe zielt dient vermutlich den Nutzern, die sich in der Umgebung gerade nicht auskennen. Für diesen Fall ist der Weg zu einer dieser Stationen jedoch ebenfalls unbekannt, was entsprechend für den Nutzer keinen klaren Mehrwert hat, da er sich anschließend um diesen Weg selber kümmern müsste, sollte von dieser Station aus eine Fahrt gestartet werden.

In der Detailansicht einer Verbindung ändern sich ebenfalls keine grundlegenden Dinge.

Einzige Änderungen werden die Zusatzinformationen zur Art des Fußwegs und der Wartezeit sein, sowie der Wegfall der Miniaturkarte für den Fußweg, der durch einen Google Maps Link ersetzt wird.

Unterhalb der Verbindungsansicht werden die möglichen Fahrkarten aufgelistet, die hier auf dem Wireframe leider nicht zu sehen sind, im Screendesign aber visualisiert werden.

Favoriten

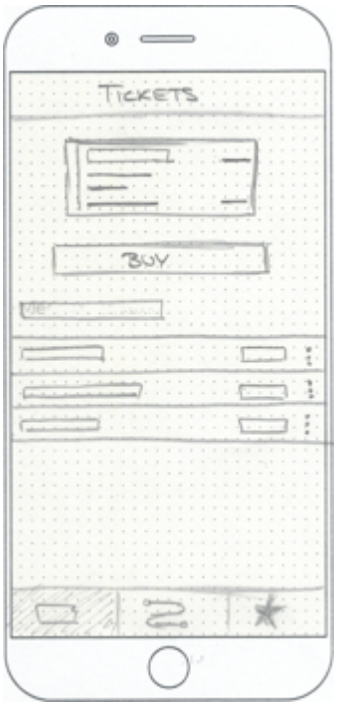
Die Favoriten, welche die Schnellziele ersetzen sollen, können in einem eigenen Reiter als Liste eingesehen werden, die die Möglichkeiten zum Hinzufügen und Bearbeiten bietet.

Initial trägt ein Favorit die Bezeichnung von Start und Ziel, wenn er über den Suchverlauf angelegt wird.

Bearbeitet werden können der Name, welcher optional vergeben werden kann, um den Überblick in der Liste zu gewährleisten, sowie Start, Ziel, Optionen und eventuelle Zwischenziele.

Die Optionen umfassen die gleichen, die auch in der Verbindungssuche aufgeführt sind, mit dem Unterschied, dass diese nur für den jeweiligen Favoriten angepasst werden und keine Auswirkungen auf andere oder die Suche ohne Favoriten haben.

Fahrkartenkauf



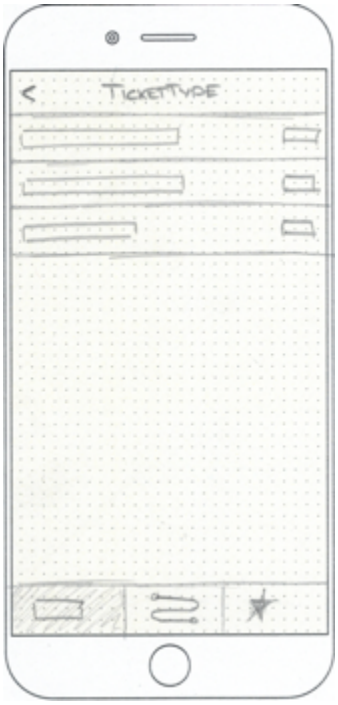
Unter dem Reiter für die Fahrkarten, wird die aktuell gültige an oberster Stelle angezeigt. Es können auch mehrere aktiv sein (beispielsweise eine Wochenkarte, zu der eine Ergänzungskarte gekauft wurde).

Unterhalb der gültigen Fahrkarte befindet sich eine Schaltfläche, um weitere zu kaufen. Sollte keine gültige Fahrkarte existieren, befindet sich die Schaltfläche ganz oben.

Darunter wird eine Liste dargestellt, die alle bisher gekauften Fahrkarten mit Bezeichner, Geltungsbereich, Datum und Preis auflistet und über eine Schaltfläche erneut gekauft werden kann oder eine Rechnung zur Verfügung stellt.

Die zum Kauf stehenden Fahrkarten selber werden nach Kategorie angezeigt, jeweils mit einer Vorschau, welche Typen sich hinter der Kategorie verstecken.

Das hat zum Vorteil, dass nicht primär in jeder Kategorie gesucht werden muss, sondern vor dem Öffnen der Kategorie klar ist, welche Fahrkarten zur Wahl stehen.



Je nach gewählter Kategorie kann eine Fahrkarte ausgewählt werden. Abhängig vom Typ werden in einzelnen Schritten Starthaltestelle, Geltungsbereich und Zuschlag gewählt.

Nachdem entsprechend alle Eingaben getätigt wurden, wird die Fahrkarte in der gleichen Art wie eine gedruckte zusammengefasst.

Unter dieser Zusammenfassung und gegebenenfalls Information zur Fahrkarte, kann das bevorzugte Zahlungsmittel ausgewählt und der Kauf daraufhin bestätigt werden.

Screendesign

Im nächsten Schritt sollte das Design greifbar visualisiert werden. Wireframes als diese dienen als Hilfe für das spätere Design, geben aber zu viel Interpretationsspielraum, um nur auf ihnen basierend mit der technischen Umsetzung zu beginnen.

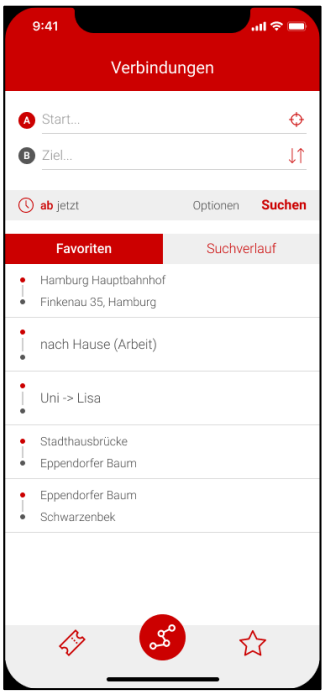
Im wirtschaftlichen Bereich, wenn es sich um eine Dienstleistung handelt, ist es ebenfalls stark von Vorteil, sich zunächst auf das Layout zu einigen, bevor das konkrete Design erstellt wird. Neben den klaren Vorzügen, dass der Ablauf für alle Beteiligten transparenter und strukturierter wird und die iterativen Anpassungen vom Umfang immer moderat sind, liegt ein weiterer Vorteil für den Kunden darin, dass er sich auf kleinere Teilaspekte konzentrieren kann, anstatt von Farbe, Schriftart, Abständen und Formulierungen abgelenkt zu werden. Änderungen an diesen sind selbstverständlich legitim und erfolgen auch entsprechend, sind hingegen, wenn es um das Konzept und das Layout geht, eine große Ablenkung.

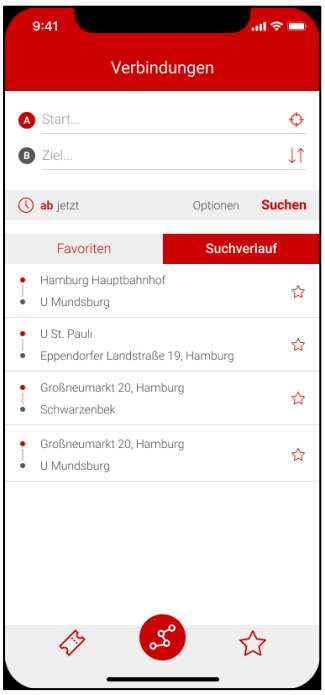
Nachdem den Wireframes also von allen Beteiligten zugestimmt wurde, kann die visuelle Gestaltung erfolgen.

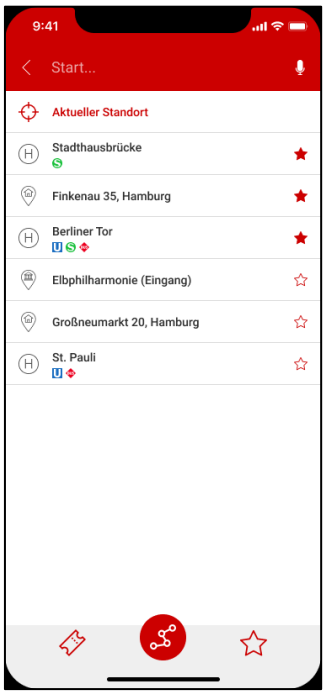
Der Umfang des Designs ist generell größer als der der Wireframes, da zur Umsetzung meist mehr visuelle Informationen gebraucht werden. Sollten – wie in diesem Fall – beispielsweise Tabs zum Einsatz kommen, sollten die jeweiligen Zustände auch dargestellt werden, um keinen Interpretationsraum für den Inhalt zu bieten, wohingegen die Wireframes nur die Tatsache ankündigen, dass Tabs verwendet werden sollen.

Verbindungssuche, -liste und -detail

Die Farbgebung orientiert sich am HVV und der momentanen App aus Wiedererkennungsgründen.







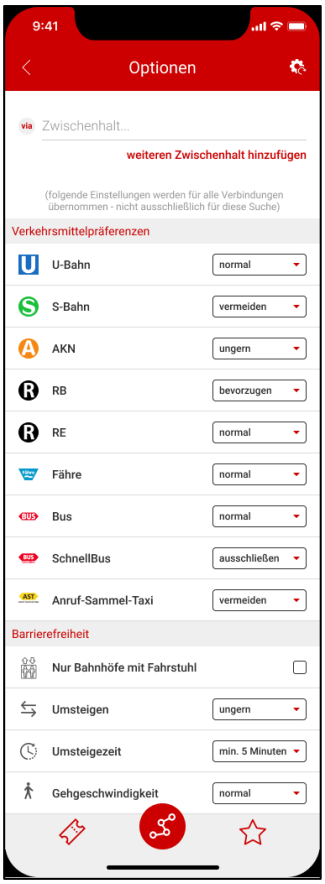
Auf diesen Screens ist basierend auf den Wireframes die Verbindungssuche dargestellt.

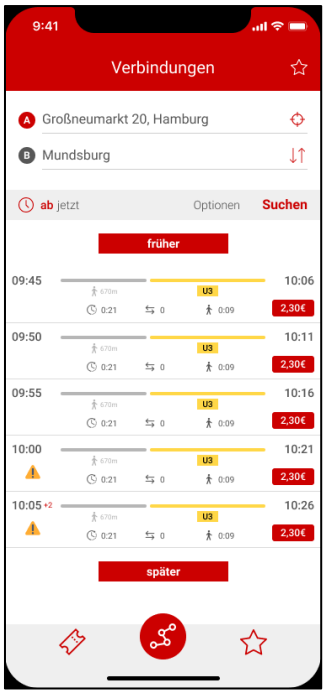
Die konkreten Inhalte, Texte und Icons werden definiert, sowohl das Verhalten der Tabs für Favoriten und Suchverlauf.

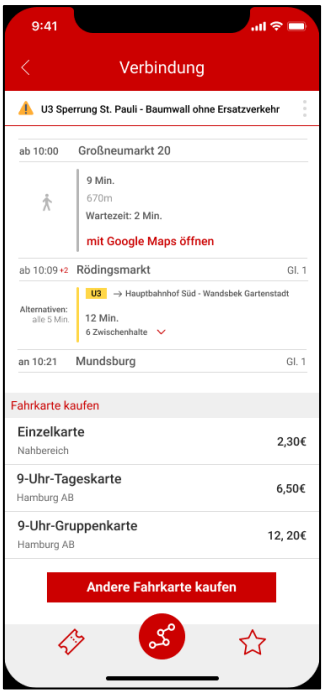
Im Suchverlauf besteht die Möglichkeit, Verbindungen zu favorisieren, die anschließend den Favoriten hinzugefügt werden und im Favoriten-Tab zu finden sind.

Die Element Start-/Ziel-Suche werden in Haltestellen, Adressen und Sehenswürdigkeiten/Orten unterteilt, die am voranstehenden Icon erkannt werden können. Daneben können einzelne dieser Elemente favorisiert werden, sodass sie sich ab dem nächsten Suchstart immer am Anfang der Liste befinden.

Als Start/Ziel kann ebenfalls in der Liste der aktuelle Standort gewählt werden.







Wie angekündigt, werden alle Einstellungsmöglichkeiten aufgelistet, ohne separate Views zu öffnen und die Möglichkeit Zwischenhalte hinzuzufügen wird auffälliger gestaltet. Dabei werden die Einstellungen gespeichert und für jede Suche verwendet, sofern ein genutzter Favorit nicht gesonderte Einstellungen besitzt. Der Vorteil besteht darin, seine Präferenzen und Barrierefreiheitseinstellungen nicht bei jeder Suche erneut eintragen zu müssen.

Die Anordnung in den zur Verfügung stehenden Verbindung wird leicht verändert, um den Platz optimaler zu benutzen und die Informationen klarer voneinander unterscheiden zu können. Sollte es sich um einen Fußweg handeln, wird die Distanz unter diesem mit aufgeführt. Die Verspätungsanzeige wird direkt neben der Zeit platziert. Die Information über eventuelle Meldungen zu der Verbindung wird von der Platzeinnahme auf einen Icon-Indikator reduziert, aus dem Grund, die Listenelemente weitestgehend konstant ihn ihrer Größe zu halten. Momentan wird neben diesem Icon noch der Text „Es liegt 1 Meldung vor.“ (oder „Es liegen [Anzahl] Meldungen vor“.) dargestellt. Welche Meldungen dies genau sind, wird erst im Verbindungsdetail zugänglich, wodurch der jetzige beanspruchte Platz zu groß ausfällt und die Information, dass es überhaupt Meldungen gibt, ausreichen sollte.

Wie erwähnt, werden „früher“ und „später“ Schaltflächen oberhalb und unterhalb der Liste eingefügt, um der logischen Richtung zu folgen.

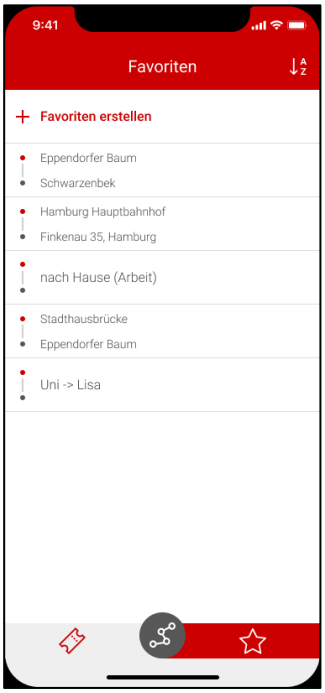
Im Verbindungsdetail werden die Titel der Meldungen mit dargestellt, um von vornherein sehen zu können, ob einen die Meldung wirklich betrifft und angesehen werden sollte. Durch einen Klick auf die Meldung, wird sie separat in einem Popup dargestellt werden.

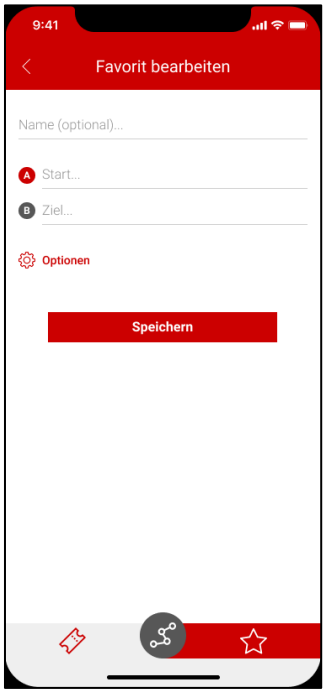
Sollte das Design von Außenstehenden verwendet werden, muss die Ansicht dessen selbstverständlich ebenfalls visualisiert werden. In diesem exemplarischen Fall genügt jedoch die Anmerkung.

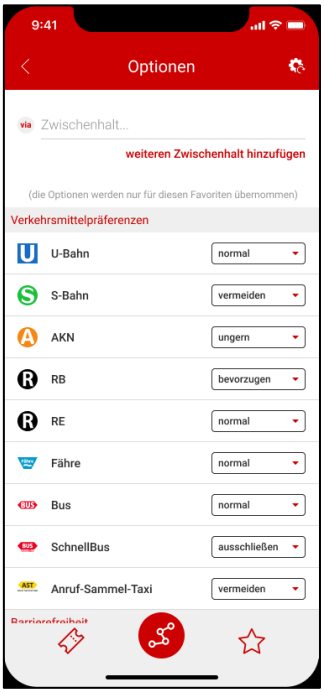
Ferner wird das Layout leicht angepasst, um bestehende White-Spaces optimal zu nutzen.

Sollte eine Wartezeit existieren, wird sie unterhalb der Distanz des Fußwegs angezeigt werden. Für den Fall, dass es sich um einen Umstieg handelt, werden Dauer, Distanz und Google Maps Link entfernt und durch ein „Umsteigezeit: [Dauer]“ ersetzt.

Favoriten



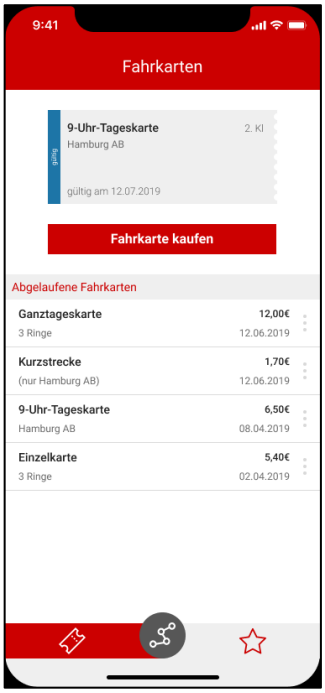


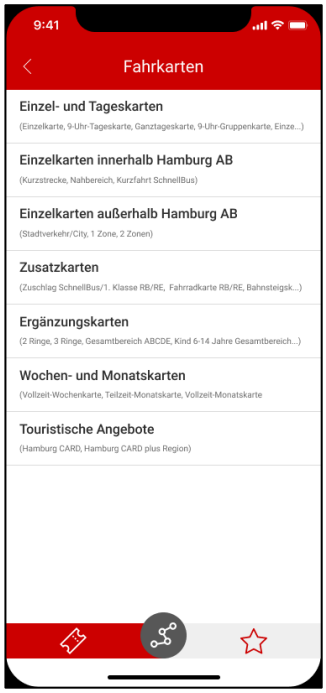


Wie beschrieben, werden unter dem Reiter „Favoriten“ die Favoriten aufgelistet, welche in der Liste alphabetisch auf- oder absteigend sortiert werden können.

Über die Funktion „Favoriten erstellen“ wird ein neuer View geöffnet, der die Eingaben Name, Start, Ziel und Optionen beinhaltet. Die Optionen haben den Hinweis, dass sie ausschließlich auf diesen Favoriten angewendet werden.

Fahrkartenkauf



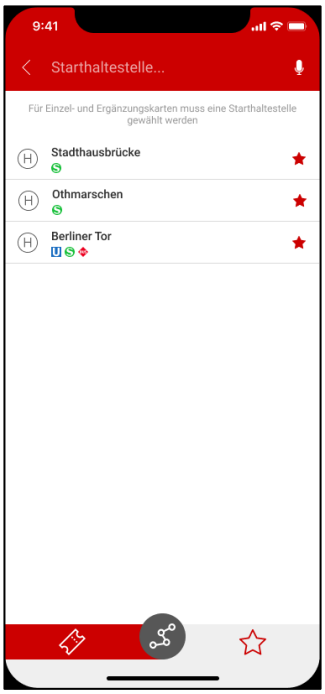


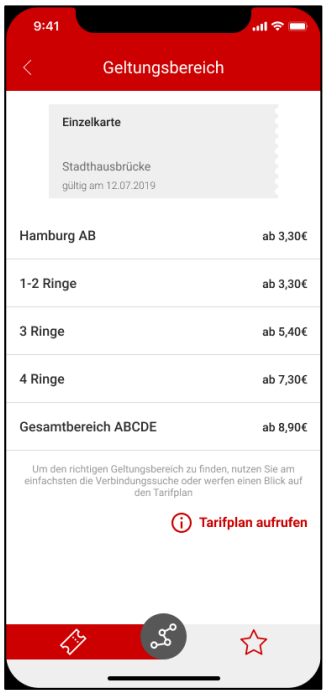
Der Fahrkartenkauf läuft wie in den Wireframes beschrieben ab.

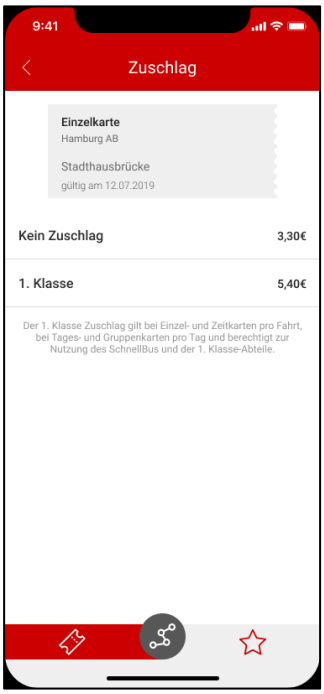
Die momentan gütigen Fahrkarten werden angezeigt, gefolgt von der Möglichkeit andere zu kaufen.

Darunter befindet sich die Liste mit den bisher gekauften Fahrkarten, die über das rechte Icon erneut gekauft werden können und eine Rechnung angefordert werden kann.

Der Kauf einer Fahrkarte folgt dem im Wireframe beschriebenen Verlauf.







Die einzelnen Schritt (Fahrkartenauswahl – (Starthaltestelle) – Geltungsbereich - Zuschlag) werden separat voneinander dargestellt, damit sich der Nutzer auf eine Aufgabe zurzeit konzentrieren kann. Ferner werden dadurch Klicks gespart, da nach der Auswahl automatisch der nächste Screen folgt und nicht jeder Schritt separat ausgewählt werden muss.

Die Zusammenfassung der Fahrkarte hat den gleichen Aufbau wie der einer gedruckten Fahrkarte von Automaten oder Bussen, um eine konsistente und schnell erkennbare Darstellung zu gewährleisten.

Im letzten Schritt des Kaufs wird das Bezahlmittel ausgewählt und anschließend die Fahrkarte gekauft.

Der Kauf einer Fahrkarte aus der Verbindung heraus, würde diesen Screen darstellen. So würde der Kauf auf zwei Schritte reduziert: Auswahl der Fahrkarte in der Verbindung und Kaufbestätigung.

II.V. Usability Testing

Steve Krug beschreibt in seinem Buch *Don't make me think – revisited* einfach zu handhabendes und für jedermann ausführbares Usabilty Testing.

Krug legt dar, dass solche Test – vor allem, wenn die Mittel und die Zeit nicht zu Verfügung stehen – nicht teuer oder zeitintensiv sein müssen.

Usabilty Test sind für den Erfolg eines Produkts hauptverantwortlich. Dadurch, dass das Produkt von den eigentlichen Anwendern benutzt wird – auch bevor es fertiggestellt ist – können Schwachstellen identifiziert und behoben werden.

Bei den Tests geht es immer darum, das Produkt zu testen und nicht die Nutzer. Nutzer geben sich gerne selbst die Schuld, wenn sie etwas nicht verstehen oder nicht finden, was sie suchen. Dabei liegt das Problem nicht beim Nutzer, sondern beim Produkt. Das Produkt ist nicht intuitiv, nicht selbsterklärend und nicht anwenderfreundlich genug – nicht der Nutzer.

Jeder, der schon mit Kunden zusammen eine Anwendung entwickelt hat, kam sicher mindestens einmal an den Punkt, an dem die Hände über dem Kopf zusammengeschlagen wurden, weil man sich das Verhalten nicht erklären konnte.

Zugegeben: Manche Kunden haben Wünsche, die in Anbetracht der Nutzerfreundlichkeit kontraproduktiv sind. Das liegt aber meist daran, dass der Kunde nicht nutzerzentriert agiert. Und das nicht mit böser Absicht.

Gemeint sind aber Dinge, die man als Designer oder Entwickler für logisch hält, vom Kunden hingegen nicht verstanden werden.

Wenn man diesen Kunden nun als Nutzer betrachtet (was er in diesem Moment durchaus ist: Er versucht sich mit dem Produkt auseinanderzusetzen und scheitert) sollte klar sein, dass nicht er das Problem ist, sondern der Weg, den wir für uns als logisch betrachten.

Im Prinzip macht das Usability Testing genau das mit unbeteiligten Personen. Und wie Krug (auf Seite 114) sagt:

> *Testing one user is 100 percent better than testing none.*

Selbst die schlechteste Wahl dieses einen Nutzers kann Dinge aufzeigen, die verbessert werden können.

Steve Krug hat über diese Art des Usabilty Testing ebenfalls ein separates Buch namens *Rocket Surgery made easy* verfasst.

In der Kurzfassung geht es darum, die Faustregel von drei Personen einmal im Monat an einem Vormittag dem Produkt vorzustellen und das Vorgehen verbalisieren zu lassen, während sich nur der Tester und der Testleiter in einem Raum befinden.

Für den Test werden verschiedene Aufgaben formuliert. In diesem Anwendungsfall beispielsweise „Suchen Sie eine Verbindung vom Hamburger Hauptbahnhof zu den Landungsbrücken.“ oder „Sie fahren jeden Tag die gleiche Strecke und überprüfen jeden Tag kurz vor Fahrtantritt noch einmal ihre Verbindung. Wie würden Sie vorgehen, um nicht jeden Tag die Eingaben neu tätigen zu müssen?“.

Der Ablauf dieses Tests wird mit Ton- und Screen-Aufnahme an eine Gruppe Beobachter außerhalb des Raumes übertragen, die sich jeweils zu jedem Tester drei Probleme notieren, die sie am wichtigsten empfanden.

Über ein Mittagessen können diese Punkte besprochen werden und entschieden werden, welche der Probleme am gravierendsten sind. Krug empfiehlt, sich für zehn zu entscheiden, sie zu mit einer Zahl von 1-10 zu bewerten und zu ordnen.

Es gilt also, sich um die größten Baustellen zu kümmern, anstatt die kleinen, schnell lösbaren anzugehen. Diese schnell lösbaren Probleme können selbstverständlich aufgelistet werden und nebenher gelöst werden, sollten aber nicht der Fokus des Tests sein.

Diese Arbeit hätte von dem Vorgehen ebenfalls profitieren können. Aus Umfangsgründen musste jedoch darauf verzichtet werden. Dennoch ist es wichtig zu erwähnen, dass Usability Tests nicht mit großen Zeit- und Geldverlusten verbunden sein müssen. Krug veranschlagt für das eigenverantwortliche Testen 100$ oder auch weniger pro Test-Tag – mit angeforderten Experten wird der Preis auf 5000 bis 10000$ pro Tag geschätzt.

III. Mögliche Umsetzung

III.I. Was sind React Native, Redux und TypeScript?

Für die Umsetzung der bisherigen Ergebnisse und der Konzipierung in einer konkreten App wurden Sprache, Framework und Libraries so gewählt, dass keine zu große Lernkurve besteht, wenn bereits mit React und Redux gearbeitet wurde.

(In Anbetracht des Umfangs dieser Thesis und ihrem Schwerpunkt, lag es einfach nahe, für die App React Native in der gleichen Konstellation zu benutzen, wie ich selber es bereits seit einigen Jahren mit React tue. So bleibt schlichtweg mehr Zeit für das Wesentliche, da sich die Einarbeitung auf ein Minimum reduziert.)

Im folgenden Abschnitt wird auf React Native, Redux und TypeScript näher eingegangen.

React Native

Um React Native erklären zu können, muss zunächst auf React eingegangen werden, da React Native seinen Ursprung genau dort hat und sehr ähnlich funktioniert.

Im React Native Tutorial unter https://facebook.github.io/react-native/docs/tutorial heißt es dazu:

> *React Native is like React, but it uses native components instead of web components as building blocks. So to understand the basic structure of a React Native app, you need to understand some of the basic React concepts, like JSX, components, state, and props. If you already know React, you still need to learn some React-Native-specific stuff, like the native components.*

Allgemeines zu React

React selber ist eine JavaScript Library (Library bezeichnet hier einfach gesprochen eine Sammlung von bereits geschriebenen Klassen, Funktionen, Unterprogrammen und Daten, die das Programmieren erleichtert und Funktionalitäten übernimmt[13]) und wurde ursprünglich für und von Facebook entwickelt.

[14]Der Anfang geht bis 2011 zurück, als die Facebook Ads (die auf den Nutzer maßgeschneiderten Werbeeinblendungen) immer mehr Funktionalität erhielten und immer mehr Mitarbeiter gebraucht wurden, damit alles reibungslos funktionieren konnte. Um Updates weiterhin effizient verwalten zu können und die gesamte Ads App kontrollierbar und wartungsfähig zu halten, musste eine neue Lösung her. Diese Lösung brachte daraufhin React.

Nachdem Facebook 2012 Instagram aufkaufte, sollte auch eben dieses von der neuen Technologie profitieren und man entschied sich, React eigenständig und open source werden zu lassen. 2013 wurde React dann der Öffentlichkeit präsentiert und open source.

Zwei Jahre später wurde React Native erstmals vorgestellt und im März 2015 für iOS und im September des gleichen Jahres für Android veröffentlicht.

React ist deklarativ und Komponenten-basiert[15]. Das bedeutet, dass eine React App (in diesem Falle also eine interaktive Webapplikation) aus kleinen und einfach zu wartenden Bausteinen besteht, die alle einen eigenen State besitzen und auf ihn reagieren können. Diese Bausteine werden automatisch re-rendered sollte sich ihr State verändern. Auf diese Weise ist jede Komponente für sich selbst verantwortlich und kann entsprechend einfacher debugged werden. Da alles in JavaScript geschrieben wird, können Komponenten auch übergreifend über Properties miteinander Daten teilen und kommunizieren.

Ein weiterer Vorteil von React besteht darin, dass Facebook darauf aufbaut. Was im Endeffekt bedeutet: Wenn eine Stelle ein Update bekommt, darf die App nicht plötzlich zusammenstürzen. Facebook ist gigantisch und sollte so etwas passieren, müsste man unzählige Komponenten anpassen, und das hieße viel Zeit, sehr viel Arbeit und noch mehr Kosten. Heißt also, React muss sehr verlässlich sein und es darf keine Notwendigkeit darin bestehen, nach einem Update viele Dinge umschreiben zu müssen. Das bildet natürlich einen enormen Vorteil auch für kleinere Anwender, da man keine Angst vor einem React-Update haben muss.

---

[13] „Software Library“, https://www.techopedia.com/definition/3828/software-library, letzter Aufruf: 24.06.2019 – 12:13 Uhr

[14] 2018, Andrea Papp, „The history of React.js on a timeline”, https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/, letzter Aufruf: 24.06.2019 – 13:06 Uhr
(die folgenden drei Absätze bedienen sich ebenfalls der Inhalte dieser Quelle)

[15] https://reactjs.org/, letzter Aufruf: 16.06.2019 – 15:33 Uhr

Programmierung mit React

(Der folgende Abschnitt ist eine kurze Zusammenfassung des ‚guide to main concepts' unter https://reactjs.org/docs/getting-started.html)

Um nun ein HTML-Element nach außen darzustellen, verwendet React weder einen String noch HTML-Code. Stattdessen wird JSX verwendet. Laut React ist dies kein Muss, der Hauptteil der Nutzer empfindet es jedoch als sehr hilfreich.[16]

Die folgenden Beispiele sind alle valide JSX:

```
const element = (
    <h1 className='someElement'>
        Hello {someVariable}!
    </h1>
)

//

const element = <h1>Hello world!</h1>

//

const element = <h1>Hello {someFunction()}! </h1>

//

if (someCondition) {
    return <h1>Hello world!</h1>
}

//

const element = <h1 someParameter={someVariable}> Hello world! </h1>

//

const element = (
    <div>
        <h1>Hello!</h1>
        <h2>Good to see you here.</h2>
    </div>
)
```

[16] https://reactjs.org/docs/introducing-jsx.html, letzter Aufruf: 24.06.2019 – 15:16 Uhr

Der einzige DOM-Node – also ein wirkliches HTML-Element, kein JSX oder String – der gebraucht wird, ist der, in den alles hinein-gerendert wird. Dieses DOM bekommt in der Regel *root* oder *app* als ID gesetzt und wird ‚root'-DOM genannt. Dieses liegt auch im bekannten index.html der ganzen Anwendung

Im Normalfall also:

```
<div id='root'></div>
```

Ausnahmen sind beispielsweise bestehende Webseiten, die auf React umgestellt werden und noch aus mehreren Unterseiten bestehen.

Prinzipiell können so viele eigenständige ‚root'-DOMs existieren, wie man möchte. Der sauberste Weg ist jedoch ein einziges.

Der eigentliche Inhalt wird dann auf JavaScript-Ebene in das ‚root'-DOM gerendert.

```
ReactDOM.render(element, getElementById('root'))
```

Den eigentlichen Inhalt bilden die Komponenten. Diese können auf zwei Weisen erstellt werden.

```
function Welcome(props) {
    return <div>Hello, {props.name}</div>;
}
```

oder

```
class Welcome extends React.Component {
    render() {
        return <div>Hello, {this.props.name}</div>;
    }
}
```

Mittlerweile hat sich der Standard etabliert, die erste Variante zu benutzen, wenn die Komponente nur zur Darstellung verwendet wird und entsprechend keinen eigenen State und keinen Zugriff auf Lifecycles (später mehr zu Lifecycles) benötigt. Das hat die Vorteile, dass die Komponente einfacher zu lesen und zu testen ist und weiter auch weniger Code in Anspruch nimmt.[17]

[17] Juli 2018, David Jöch, https://medium.com/@Zwenza/functional-vs-class-components-in-react-231e3fbd7108, letzter Aufruf: 24.06.2019 – 16: 15 Uhr

Um die Komponente auch darzustellen, wird sie also anschließend erstellt und mit ReactDOM.render() in das ‚root'-DOM gerendert.

```
const element = <Welcome name={'Alice'}/>
ReactDOM.render(element, getElementById('root'))
```

State und Lifecycles von React

Jede React-Komponente kann einen eigenen State halten. Ein State ist im einfachsten Sinne ein Objekt von Daten, die mit getState() gelesen werden und mit setState() geändert werden können.

Bei jeder Änderung des States wird die Komponente neu gerendert.

Soll beispielsweise per Klick auf einen Button ein Zähler inkrementiert und dargestellt werden, kann

```
class SomeComponent extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            counter: 0
        }
    }

    render() {
        return (
            <div>
                {this.state.counter}
                <button
                    onClick={() => this.setState(
                        {counter: this.state.counter + 1}
                    )}
                    label={'increment'}/>
            </div>
        );
    }
}
```

geschrieben werden. Beim *onClick* des Buttons wird der State verändert und entsprechend wird die Komponente neu gerendert und stellt den aktuellen Counter dar.

Dieses Prinzip macht es auch einfach, zum Beispiel Listen ein- oder auszuklappen oder allgemein Aktionen durchzuführen, die direkte Darstellungs-Änderungen mit sich ziehen sollen.

Wichtig hierbei ist, dass nur diese eine Komponente, deren State sich ändert, sich neu rendert. Alle weiteren Komponenten sind davon nicht betroffen – vorausgesetzt sie erhalten keine Informationen aus diesem State über die Props.

Sollte also die Counter-Darstellung eine eigene Komponente sein, wird sie mit

```
<CounterComponent counter={this.state.counter}/>
```

die Änderung des States ebenso registrieren und neu rendern, da sich ihre Props verändern.

Was automatisch zu den Methoden *shouldComponentUpdate()* und *componentDidUpdate()* führt. (In manchen Fällen werden *getDerivedStateFromProps()* und *getSnapshotBeforUpdate()* benutzt, für diesen Anwendungsfall sind diese Methoden jedoch nicht nötig, weshalb ich weiter darauf eingegangen wird)

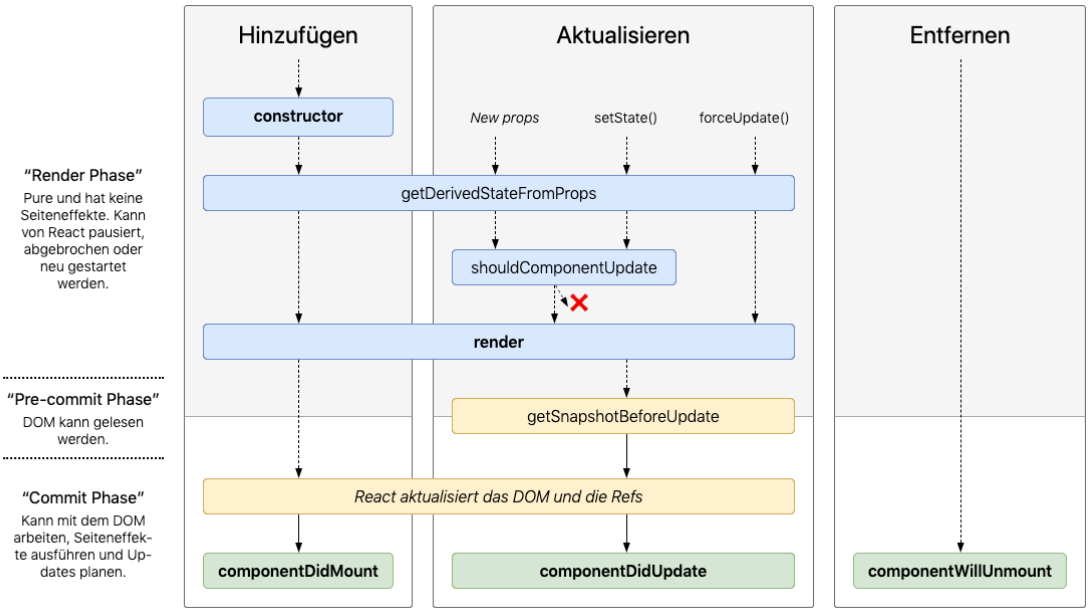


Abbildung 14 - React Lifecycles (http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/, letzter Aufruf: 24.06.2019 – 10:37 Uhr)

Wie in Abbildung 14 zu sehen, wird *shouldComponentUpdate()* vor *render()* ausgelöst, sprich: Bevor die Komponente gerendert wird, wird überprüft, ob sich der eigene State oder die Props verändert haben und entsprechend wird die Komponente aktualisiert oder nicht. Nach der Aktualisierung wird *componentDidUpdate()* aufgerufen – sollte keine Aktualisierung nötig gewesen sein, wird die Methode nicht aufgerufen.

React Native

React Native ist ganz einfach gehalten, nahezu identisch mit React, nur dass statt HTML-Elementen wie * *, *<p>* oder *<button>*, native Elemente verwendet werden wie *<View>*, *<Text>* und *<Button>*. Die Funktionsweise bleibt jedoch bestehen. In React ist es möglich CSS zu schreiben, um die Komponenten zu gestalten. CSS kann sowohl inline – direkt in JSX in camel case über die style-prop – als auch in separaten Dateien geschrieben werden.

```
class SomeComponent extends React.Component {
    render() {
        return (
            <div style={{backgroundColor: 'red'}}>
                Hello, world!
            </div>);
    }
}
```

Bei React Native wird inline geschrieben, mit der Möglichkeit über *Stylesheet.create()* ein Objekt zu erstellen, dass alle Styles beinhaltet und einem – wie der Name schon sagt – CSS-Stylesheet ähnelt. Dabei ist anzumerken, dass keine besonderen Sprachen oder Syntax erforderlich sind, aber sich im Allgemeinen wie CSS-Attribute verhalten, die im camel case geschrieben werden.

```
class SomeReactNativeComponent extends React.Component {
    render() {
        const styles = Stylesheet.create({
            viewStyle: {
                backgroundColor: 'red'
            }
        })

        return (
            <View style={[styles.viewStyle]}>
                <Text>Hello, world!</Text>
            </View>
        );
    }
}
```

Da manche Komponenten unter Umständen je nach Plattform spezifische Aufgaben oder Anforderungen haben, kann mit *Platform.select()* festgestellt und entschieden werden, was genau bei welchem System passieren soll.

```
Platform.select({
    ios: () => console.log("I'm an iOS device!)"),
    android: () => console.log("I'm an Android device!")
})
```

Durchaus kann es aber auch passieren, dass wirklich nativer Code je nach Plattform nötig ist. React Native lässt daher zu, auch nativ in beispielsweise Swift oder Java zu programmieren.

Da nativer Code für diese App jedoch nicht nötig ist, wird nicht weiter behandelt, welche Besonderheiten dieses Vorgehen mit sich bringen würde. Wobei an dieser Stelle anzumerken ist, dass das daraus resultierende Programmieren und Warten in drei Sprachen ein ausschlaggebender Punkt dafür waren, weshalb Airbnb React Native wieder fallen ließ.[18]

[18] Juni 2018, Braus, https://medium.com/braus-blog/airbnb-is-dropping-react-js-should-you-too-dcbff36def5c, letzter Aufruf: 24.06.2029 – 11:42 Uhr

Redux

Allgemeines zu Redux

Redux wurde im Juni 2015[19] veröffentlicht und wird als *„predictable state container for JavaScript apps“*[20] bezeichnet.

Die Bezeichnung entspricht bildlich und einfach gehalten einem Behälter von Dingen, von dem wir jederzeit wissen, was hinzugefügt wurde, was entfernt wurde und was geändert wurde und darüber hinaus auch von wem. Der Behälter hat einen durchsichtigen Deckel, wir können entsprechend alles sehen, was sich zu einem bestimmten Zeitpunkt in ihm befindet – aktiv von außen etwas ändern, können wir hingegen nicht. Wir können lediglich jemandem mitteilen, dass darin etwas passieren soll, welcher daraufhin veranlasst, dass dies auch passiert.

Dieser Behälter nennt sich Redux Store, die Dinge, die sich in ihm befinden, sind die Gesamtheit der States der Applikation und das, was dem Store mitteilt, dass etwas passieren und was genau passieren soll, nennt sich Action. Alles, was im Store gespeichert ist, kann gelesen werden, aber nicht direkt von außen, sondern nur von den Actions geändert werden. Damit die Actions nicht nur mitteilen, dass und was passieren soll, sondern auch etwas passiert, bedarf es einem Reducer, der sie ausführt. Das Ausführen der Action wird Dispatch genannt.

Der Übersicht halber als Schaubild in Abbildung 15.

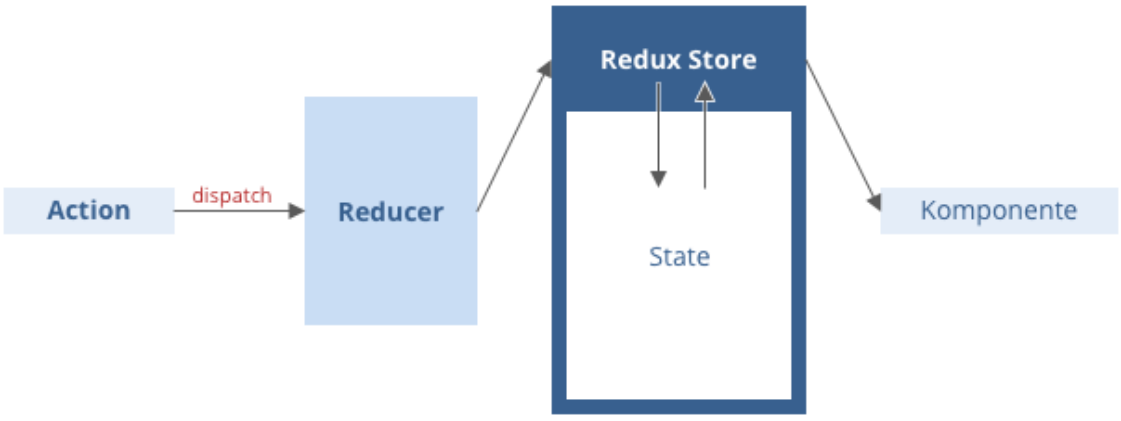


Abbildung 15 – Redux Aufbau

Einbindung von Redux

[19] https://github.com/reduxjs/redux/releases/tag/v0.2.0, letzter Aufruf: 24.06.2019 – 16:01 Uhr

[20] https://redux.js.org/, letzter Aufruf: 24.06.2019 – 12:24 Uhr

React Native und Redux problemlos verbunden werden. Die einzigen Libraries, die dafür nötig sind und explizit installiert werden müssen, sind *redux* und *react-redux.* Erstere übernimmt hier die Store-Reducer-Action Logik und zweitere dient der unkomplizierten Zusammenführung in der Komponente.

Wenn wir React (Native) mit Redux verwenden, werden unsere Komponenten in zwei Teile geteilt, sofern sie einen State und Actions benötigen. Den einen Teil bildet ein Container, der den State und die Actions an die jeweilige Komponente weitergibt. Den anderen Teil bildet die darstellende Komponente. Der Container stellt selber nur Daten bereit und teilt sie der Komponente mit, wohingegen die Komponente die Informationen nur wiedergibt.

Die Schritte zur Einbindung werden folgend am Beispiel einer To-Do Liste erklärt.

(Das Beispiel lehnt an das Tutorial von https://react-redux.js.org/introduction/basic-tutorial (letzter Aufruf: 14.08.2019 – 10:45 Uhr) an. Die Erklärungen in diesem Abschnitt sind ebenfalls diesem Basic Tutorial zu entnehmen.)

Für solch eine To-Do Liste benötigt man in der einfachsten Variante eine Liste, in der alle To-Do-Elemente dargestellt werden und Elemente hinzugefügt werden können. Die To-Do-Elemente müssen abgehakt werden können. Auf eine Löschfunktion wird der Einfachheit halber verzichtet.

Die benötigten Bausteine sind also:

- TodoListContainer (der Container für die Liste)
- TodoList (die darstellende Komponente der Liste)
- TodoElement (die darstellende Komponente eines To-Do-Elements)
- TodoListReducer (der Reducer, der das Hinzufügen und Abhaken umsetzt)
- TodoListActions (die Actions, die dem Reducer mitteilen, welche Aufgabe auszuführen ist)

Die To-Do-Liste selbst muss wissen, welche Elemente sie darstellen muss. Entsprechend muss dem Container ein Array an To-Dos mitgeteilt werden. Dieses Array befindet sich im State, damit bei Änderungen die jeweiligen Komponenten mittgeteilt bekommen, zu re-rendern, wenn sich die To-Do-Liste ändert.

Auf ein Input-Feld wird ebenfalls für die Einfachheit verzichtet. Stattdessen erhalten die To-Dos, die neu erstellt werden, den title ‚To-Do - No.' + eine fortlaufende Nummer, die im State unter todoCount gehalten wird und bei jedem Erstellvorgang erhöht wird.

Für dieses Beispiel wird festgelegt, dass ein To-Do einen *title* und ein *completed* besitzt.

Um To-Dos als erledigt (oder unerledigt) zu markieren und zu erstellen, muss die Liste die Funktion erhalten, die dies ermöglichen. Diese Funktionen werden von TodoListActions.js bereitgestellt – in diesem Fall also *addTodo* und *toggleTodo*.

(Im Folgenden wird vorausgesetzt, dass gängige JS-Funktionen wie beispielsweise *map()* und einfache React Native Komponenten wie View und Button bekannt sind. Weiter werden in den Code-Passagen die import-Statements nicht mit dargestellt.)

TodoListActions.js:

```
export function addTodo() {
    return {
        type: 'ADD_TODO',
    }
}


export function toggleTodo(index) {
    return {
        type: 'TOGGLE_TODO',
        index
    }
}
```

Wenn eine dieser Funktionen dispatched wird, führt der Reducer die Aktion durch, die durch den mitgegebenen *type* definiert wurde.

TodoListReducer.js:

```
const defaultState = {
    todos: [],
    todoCount: 0
}
const todoReducer = function (state = defaultState, action) {
    switch (action.type) {
        case 'ADD_TODO': {
            let newTodos = [...state.todos];
            newTodos.push(
                {title: 'To-Do – No.' + state.todoCount,
                 completed: false}
            )
            return {
                ...state,
                todos: newTodos,
                todoCount: state.todoCount + 1
            }
        }
        case 'TOGGLE_TODO': {
            let newTodo = state.todos.map((todo, index) => {
                if (index === action.index) {
                    return {
                        ...todo,
                        completed: !todo.completed
                    }
                } else {
                    return todo
                }
            })
            return {
                ...state,
                todos: newTodo
            }
        }
        default:
            return state
    }
}
export default todoReducer;
```

ADD_TODO führt also dazu, dass das Array *todos* um einen Eintrag erweitert wird. *TOGGLE_TODO* führt dazu, dass mithilfe des mitgegebenen *index* nur der betroffene Eintrag in *todos* als *completed* den negierten Wert des *completed* des Eintrags erhält.

TodoListContainer.js:

```
const mapStateToProps = state => {
    return {
        todos: state.todoReducer.todos
    };
};

const mapDispatchToProps = dispatch => {
    return {
        toggleTodo: (index) => dispatch(toggleTodo(index)),
        addTodo: () => dispatch(addTodo())
    }
}

const TodoListContainer = connect(mapStateToProps, mapDispatchToProps)(TodoList)
export default TodoListContainer;
```

Um der Liste selber die Informationen zu übergeben, wird auf die *mapStateToProps* und die *mapDispatchToProps* Funktionen zurückgegriffen. *mapStateToProps* gibt in diesem Falle den State als Props weiter und *mapDispatchToProps* die ausführbaren Aktionen.

Mithilfe des *dispatch* wird dem Reducer die Action mitgeteilt, deren type in TodoListActions definiert wurde. Ohne diesen *dispatch* würde keine Kommunikation mit dem Reducer stattfinden.

Die Funktion *connect()* verbindet anschließend die Komponenten mit dem Store. Es ist möglich *connect()* ohne Argumente aufzurufen. In diesem Falle wird die Komponente jedoch bei einer Änderung des States nicht re-rendern oder *dispatch* als Prop erhalten. Für den Fall, dass lediglich *mapStateToProps* übergeben wird, wird die Komponente bei State-Änderungen re-rendern, der Dispatch von Actions muss jedoch manuell in der Komponente erfolgen. Wenn *mapDispatchToProps* ebenfalls mitübergeben wird, wird die Action beim Funktionsaufruf in der Komponente automatisch dispatched.

TodoList.js:

```
export class TodoList extends React.Component{
    render(){
        const todos =( this.props.todos || []).map((todo, index) => {
            return (
                <TodoListElement
                    toggleTodo={() => this.props.toggleTodo(index)}
                    title={todo.title}
                    completed={todo.completed}/>
            )
        })

        return(
            <View className="todoListWrapper">
                <Text>To-Dos</Text>
                <Button
                    onClick={() => this.props.addTodo()}
                    title={'Add new To-Do'}/>
                {todos}
            </View>
        )
    }
}
```

TodoListElement.js:

```
export class TodoListElement extends React.Component {
    render() {
        return (
            <View className="todoElementWrapper"
                 onClick={() => this.props.toggleTodo()}>
                <Text style={{
                    textDecoration: this.props.completed ?
                        'line-through' :
                        'none'
                }}>
                    {this.props.title}
                </Text>
            </View>
        )
    }
}
```

App.js:

```
const reducers = combineReducers({...todoReducer});
const store = createStore(reducers);

ReactDOM.render(
    <Provider
        store={store}>
        <div className="App">
            <TodoListContainer/>
        </div>
    </Provider>
    ,
    document.getElementById('root')
);
```

In der App.js werden mithilfe von *combineReducers()* die Reducer zu einem zusammengeführt. Anschließend wird mit *createStore()* durch diesen kombinierten Reducer der Store erstellt. In diesem Beispiel existiert lediglich ein Reducer. In komplexeren Anwendungen werden hingegen mehrere Reducer verwendet, die in der Regel nach Kontexten erstellt werden.

Um den Store der Anwendung auch zur Verfügung zu stellen, wird die *Provider*-Komponente verwendet. Im Normalfall können Komponenten, die mit connect() aufgerufen werden, nicht außerhalb dieses Providers verwendet werden.

Die Provider-Komponente wird für gewöhnlich also nur als Top-Level benutzt, da alle anderen Komponenten mithilfe von connect() mit dem mitgegebenen Store verbunden werden können.

Diese To-Do-Liste ist auf diese Art natürlich in keiner Weise von Nutzen. Sie dient hier lediglich als Beispiel, wie Redux mit React Native verwendet wird. An dieser Stelle kann noch erwähnt werden, dass der Aufbau für Redux in React identisch ist.

Damit dieser Code auch lauffähig ist, gehen selbstverständlich noch Schritte voraus. Diese Schritte werden im Abschnitt Projekt Setup näher erläutert.

TypeScript

Als weiteres Hilfsmittel kommt TypeScript zum Einsatz und ist als Erweiterung zum herkömmlichen JavaScript zu betrachten.

In JavaScript existieren keine Typen wie beispielsweise in Java, wo Variablen festgelegte Datentypen benötigen und Funktionen festgelegte Rückgabe-Datentypen.

Eine Variable kann in JavaScript zunächst eine Zahl sein und bei der nächsten Belegung ein String, ohne dass es zu Fehlern bei der Interpretation kommt.

Anders sieht es aus, wenn man TypeScript verwendet. Variablen wird entweder durch die initiale Zuweisung ein Typ gegeben oder explizit.

```
let someVariable = 42;
```
hat zur Folge, dass die Variable vom Typ number ist.

```
let someVariable: number;
```
weist der Variable explizit den Typ number zu.

In beiden Fällen wird es zur Fehlermeldung kommen, sollte versucht werden, sie mit einem String zu überschreiben.

Manche Variablen können unter Umständen die Möglichkeit haben, mehrere Typen anzunehmen. Für diesen Fall wird ein einzelner senkrechter Strich verwendet.

```
let someVariable: number | string;
```

Das bedeutet, dass die Zuweisungen

```
someVariable = 'abc';
```

sowie

```
someVariable = 42
```

valide sind.

Die Entwicklungsumgebung wird im Falle einer falschen Zuweisung einen Fehler anzeigen. Spätestens beim Kompilieren wird es zu Problemen kommen.

Typen können auch selbst als interface oder type definiert werden. Ob interface oder type die bessere Wahl ist, bleibt prinzipiell jedem Entwickler selbst überlassen.[21]

[21] März 2018, Martin Hochel, https://medium.com/@martin_hotell/interface-vs-type-alias-in-typescript-2-7-2a8f1777af4c – letzter Aufruf: 14.08.2019 – 11:30 Uhr

Für das vorherige Beispiel der To-Do Liste, sähe das Interface für ein To-Do wie folgt aus:

```
export interface Todo {
    title: string,
    completed: boolean
}
```

Als Type würde folgende Schreibweise verwendet:

```
export type Todo = {
    title: string,
    completed: boolean
}
```

Manche Felder innerhalb eines Interface oder Type können auch als optional gekennzeichnet werden, indem

```
someField?: boolean
```

geschrieben wird.

Sollte eine Variable nur gewisse Werte erhalten dürfen, bietet sich ein Type an, der beispielsweise so aussehen könnte:

```
export type SpecificValues = 'a' | 'b' | 'c' | 0;
```

Diese typisierte Programmierung hat zum Vorteil, dass Fehler schon während des Schreibens identifiziert werden können und keine langen Debugging-Zeiten benötigt werden, um am Ende vielleicht festzustellen, dass man sich lediglich vertippt hat.

Sollte hingegen völlig unerheblich sein, welchen Typ eine Variable hat, kann der Typ `any` verwendet werden.

Weiter enden TypeScript-Dateien auch mit .ts oder .tsx. Der hauptsächliche Unterschied dieser beiden Endungen besteht (neben kleinen Besonderheiten) darin, dass in tsx-Dateien JSX verwendet werden kann. Für React und React Native kommt also nur .tsx in Frage, sofern JSX auch verwendet wird.

Auf den ersten Blick mag es den Anschein haben, dass TypeScript mehr Schreibarbeit erfordert. Zugegeben, das ist auch der Fall – wie bei Redux auch. Auf lange Sicht hat es jedoch den enormen Vorteil, dass Fehler um einiges schneller erkannt werden können und viel Zeit im Debugging eingespart wird.

Einbindung von TypeScript

Um TypeScript verwenden zu können muss lediglich die *typescript* Library installiert werden.

In den meisten Fällen existiert ein *tslint.json*-Datei, die genauer definiert, welche TypeScript-Regeln ignoriert werden sollen – beispielsweise alphabetisch sortierte *import*-Statements. Für das einmalige Ignorieren einer TypeScript-Regel kann

```
//@ts-ignore
```

über die betroffene Zeile geschrieben werden.

Für die spätere Umsetzung der Umgestaltung, wird zusätzlich auf *typescript-fsa* und *typescript-fsa-reducers* zurückgegriffen, die zusätzliche Möglichkeiten speziell für Redux mit sich bringen.

Im Abschnitt Projekt Setup – Module wird genauer darauf eingegangen.

III.II. Projekt Setup

Basierend auf dem „Getting Started“-Guide auf https://facebook.github.io/react-native/docs/getting-started.html ist der einfachste Weg, eine React Native App zu erstellen, Expo zu nutzen.

Vorausgesetzt Node 10+ ist bereits installiert, muss zunächst Expo installiert werden. Empfohlen wird dies über über die Konsole per *npm install -g expo-cli*. Um die App zu initialisieren wird einfach *expo init MyProject* ausgeführt. Nachdem die Erstellung fertiggestellt wurde, kann mit npm start ein Development Server gestartet werden, welcher sich typischerweise in einem Browserfenster öffnet.

Im Grunde ist es damit schon getan. Um die App auch sehen zu können, muss entweder ein Android- oder iOS-Emulator definiert sein. Eine andere Möglichkeit, um keine Entwicklungsumgebung dahingehend einrichten zu müssen, ist, den Expo Client unter https://play.google.com/store/apps/details?id=host.exp.exponent&hl=de auf seinem Smartphone zu installieren.

Um anschließend die App schreiben zu können, kann die App.js editiert werden.

In den meisten Fällen werden noch weitere Libraries mit Node.js installiert, wie *react-navigation*. Welche das im konkreten Fall dieser Arbeit sind, wird im nächsten Abschnitt aufgegriffen.

Module

Um sich nicht selbstständig um beispielsweise Navigation und Redux kümmern zu müssen, kommen einige Module zum Einsatz.

Im *package.json* der App werden diese hinterlegt, sodass sie per npm install gebündelt installiert und aktualisiert werden können. Node Module werden unter *node_modules* gespeichert und sind somit bereit, verwendet zu werden.

Die wichtigsten dieser Module und deren Funktion werden folgend beschrieben.

Redux

redux

Mithilfe dieses Moduls wird die Funktion von Redux selbst gewährleistet und wird vorwiegend für das Erstellen des Stores verwendet.

react-redux

Um auf die *connect*-Funktion zugreifen zu können, die wie bereits im Abschnitt zu Redux erwähnt mit den Parametern *mapStateToProps* und *mapDispatchToProps* den Store mit der Komponente zusammenführt, wird react-redux benötigt.

TypeScript

typescript

Um Typescript nutzen zu können, muss es natürlich bereitgestellt werden. Dadurch wird die Verwendung der Types ermöglicht.

typescript-fsa

Dieses Modul bezieht sich speziell auf die Verwendung von Redux und erleichtert die Erstellung von Actions. Anstatt für jede Action

```
export function toggleTodo(index) {
    return {
        type: 'TOGGLE_TODO',
        index
    }
}
```

schreiben zu müssen, kann durch dieses Modul ein *actionCreator* erstellt werden, der die Action durch

```
export const toggleTodo = actionCreator<number>("TOGGLE_TODO");
```

erstellt und typesafe macht.

`<number>` stellt in diesem Fall den *index* dar. Alternative könnte ebenfalls

```
export const toggleTodo = actionCreator<{index:number}>("TOGGLE_TODO");
```

geschrieben werden. Bei einem Parameter wird darauf jedoch für gewöhnlich verzichtet.

Diese Art und Weise spart Zeit, da weniger Code geschrieben werden muss und verbessert die Lesbarkeit dadurch ebenfalls.

typescript-fsa-reducers

Um nicht nur bei den Actions von TypeScript zu profitieren, gibt es die Möglichkeit ebenfalls ein Modul für Reducer zu installieren.

Dadurch können Reducer typesafe schnell geschrieben werden, indem sie mit *reducerWithInitialState()* erstellt werden:

```
export const SomeReducer = reducerWithInitialState(defaultState)
    .case(someActions.toggleTodo, (state, payload) => {
        return {
            ...state,
            //something happening
        };
    })
```

React Navigation

react-navigation

Dieses Modul für die Navigation lässt Drawer-, Stack- und Tab-Navigationen erstellen. React Native selber verfügt nicht über eine API zur Navigation, wodurch dieses Modul vermutlich von nahezu allen verwendet wird.

Die einzelnen Routen der Navigation müssen entweder Komponenten oder *Navigators* sein.

Um die Navigation mit der App zu verbinden, wird beispielsweise

```
const AppNavigatorContainer = createAppContainer(AppDrawerNavigator);
```

geschrieben.

react-navigation-redux-helpers

Da die Navigation selber einen State hat und dieser in vielen Fällen auch von anderen Komponenten benötigt wird, kann er mit diesem Modul zum Store hinzugefügt werden, um den reibungslosen Ablauf mit Redux zu ermöglichen.

On-Boarding

react-native-onboarding-swiper

Um wie erwähnt ein On-Boarding zu realisieren kann beispielsweise auf dieses Modul zurückgegriffen werden, welches über eine *Onboarding* Komponente verfügt, die als Props unter anderem die verschiedenen Seiten als Array, Anpassungen an „Done“- und „Skip“-Button, sowie eigene Komponenten für diese erhalten kann. Für dieses Modul liegt jedoch kein TypeScript-Support vor.

III.III. Mögliche Programmierung

Abschließend werden Beispiele gezeigt, wie manche Komponenten umgesetzt werden könnten. Dabei wird die Funktionalität jedoch außer Acht gelassen. Unter anderem Actions und Reducer werden nicht aufgezeigt. Die Beispiele zielen auf die Darstellung und Strukturierung der Komponenten ab.

Navigation

Für die Navigation wird ein *TabNavigator* benötigt, der die jeweiligen *StackNavigators* der Tabs erhält.

Der *TabNavigator* kann aus drei Tabs bestehen – der Verbindungssuche, dem Ticketkauf und den Favoriten.

```
const AppTabNavigator = createBottomTabNavigator(
    {
        Tickets: {
            screen: TicketsStackNavigator,
        },
        Routes: {
            screen: RoutesStackNavigator
        },
        Favorites: {
            screen: FavoritesStackNavigator,
        },

    },
    {
        initialRouteName: "Routes",

        tabBarComponent: CustomTabBarContainer,
        animationEnabled : true
    }
);

export default AppTabNavigator;
```

Die einzelnen *StackNavigator* könnten folgendem Aufbau folgen:

```
export const RoutesStackNavigator = createStackNavigator(
    {
        RouteSearch: { //der Name der Route/des Screens
            screen: RouteSearchContainer //die darzustellende Komponente
        },
        //... alle weiteren Routen, die sich im Stack befinden sollen
    },
    {
        initialRouteName: 'RouteOptions', //der Screen, der als erstes angezeigt wird
    }
);
```

Um die TabBar nach der Vorlage zu gestalten, muss sie eigenhändig geschrieben werden. Dafür würde der der State der React Navigation herangezogen, um auf die einzelnen Routen zugreifen zu können.

```
const {
    routes
} = navigation.state;


const tabsToRender = this.propsroutes.map((route, index) => {
    const focused = index === navigation.state.index;

    switch (route.key) {
        case "Tickets":
            return (
                <TouchableOpacity key={'route' + route.key}
                                  onPress={() => {
                                      this.props.navigation.navigate(route.routeName)
                                  }}
                                  style={{
                                      zIndex: 1,
                                      height: 40,
                                      //...
                                  }}>
                    <Image style={{height: 30, width: 30}}
                           source={/*Icon für den (in)aktiven Tab*/}/>
                </TouchableOpacity>
            );
        case "Favorites":
            // ...wie 'Tickets'

        case "Routes":
            return (
                <View key={'route' + route.key}
                                  style={{
                                      zIndex: 1,
                                      height: 40,
                                      //...
                                  }}>

                    <TouchableOpacity key={'route' + route.key}
                                      onPress={() => {

this.props.navigation.navigate(route.routeName)
                                      }}
                                      style={{
                                          height: 50,
                                          width: 50,
                                          //...
                                      }}>

                        <Image style={{height: 30, width: 30, tintColor: "#ffffff"}}
                               source={/*Icon für den (in)aktiven Tab*/}
                    </TouchableOpacity>
                </View>

            );
    }
})
```

Suchmaske

Da die Suchmaske nicht wie ein Input funktioniert, sondern auf einen Klick einen anderen Screen öffnet, ist es nicht notwendig, die *TextInput* Komponente zu verwenden. Stattdessen kann (in einer Funktion, da die Darstellung immer demselben Muster folgt) sie herausgelöst werden und nur mit *Touchables* und *Views* realisiert werden.

```
customSearchInput = (icon: any, iconEnd: any, placeholder: string, value: string) => {
    return (
        <View
            style={{
                flexDirection: 'row',
                alignItems: 'center',
               //...
            }}>
            <Image
                source={icon}
                style={{
                    height: 20,
                    width: 20,
                }}
                resizeMethod={'scale'}
            />
            <View
                style={{
                    paddingHorizontal: SMALL,
                    flexGrow: 1,

                }}>
                <View
                    style={{
                        paddingVertical: SMALL,
                        flexGrow: 1,
                        //...
                    }}>
                    <Text
                        style={{
                            fontFamily: 'Roboto',
                            fontWeight: '100',
                            //...
                        }}
                    >
                        {value ? value : placeholder}
                    </Text>
                    <Image
                        source={iconEnd}
                        style={{
                            height: 18,
                            width: 18,
                        }}
                        resizeMethod={'scale'}
                    />
                </View>
            </View>
        </View>
    )
}
```

Die Leiste für Uhrzeit, Optionen und Suche könnte wie folgt aussehen:

```
<View
        style={{flexDirection: 'row', justifyContent: 'space-between',/*...*/}}>
        <TouchableOpacity
            onPress={() => console.log('set time')}
            style={{flexDirection: 'row', justifyContent: 'flex-start', /*...*/}}>
            <Image
                style={{width: 18, height: 18, /*...*/}}
                resizeMethod={'scale'}
                source={Images.icons.time}
            />
            <Text style={{color: '#CC0000', fontWeight: '500', /*...*/}}>
                ab
            </Text>
            <Text style={{color: '#4A4A4A'}}>
                jetzt
            </Text>
        </TouchableOpacity>
        <View style={{
            flexDirection: 'row'
        }}>
            <TouchableOpacity onPress={()=>
this.props.navigation.navigate('RouteOptions')}>
                <Text style={{color: '#4A4A4A'}}>
                    Optionen
                </Text>
            </TouchableOpacity>
            <TouchableOpacity style={{marginLeft: BIG}} onPress={() =>
console.log('start search')}>
                <Text style={{color: '#CC0000', fontWeight: '500',  /*...*/}}>
                    Suchen
                </Text>
            </TouchableOpacity>
        </View>
    </View>
</View>
```

Verbindungsoptionen

Die Präferenz-Optionen könnten ebenfalls als darstellende Funktion geschrieben werden und über die einzelnen Verkehrsmittel gemappt dargestellt werden.

```
listItemMultiple = (icon: any, title: string, /*...*/ key: string) => {
    return (
        <View key={key} style={{
            flexDirection: 'row',
            justifyContent: 'space-between',
            /*...*/
        }}>
            <View style={{
                flexDirection: 'row',
                justifyContent: 'flex-start',
                /*...*/
            }}>
                <Image source={icon}
                       style={{
                           width: 24,
                           height: 24,
                           marginRight: DEFAULT
                       }}
                       resizeMethod={'scale'}/>
                <Text style={{fontWeight: '400'}}>
                    {title}
                </Text>
            </View>

            <TouchableOpacity style={{
                paddingHorizontal: SMALL,
                paddingVertical: TINY,
                /*...*/
            }}
                              onPress={() => console.log('open popup')}>
                <Text style={{color: '#4A4A4A', fontSize: 12,  /*...*/}}
numberOfLines={1}>
                    option
                </Text>
                <Image source={Images.icons.arrow down} style={{width: 8, height:
4}}/>
            </TouchableOpacity>
        </View>
    )
}


//------


const vehiclePrefsToRender = vehiclePref.map((pref, index) => {
    return (
        this.listItemMultiple(Images.vehicles[pref.icon], pref.name, demoOptions, 2,
('pref' + index))
    )
})


//------
```

```
<ScrollView>
    {this.customInput(Images.icons.via, "Zwischenhalt...", '')}
    <TouchableOpacity style={{alignItems: 'flex-end', paddingHorizontal: DEFAULT}}>
        <Text style={{color: '#cc0000', fontSize: 14, /*...*/}}>
            Weiteren Zwischenhalt hinzufügen
        </Text>

    </TouchableOpacity>

    <View style={{
        paddingHorizontal: DEFAULT,
        marginTop: BIG,
        marginBottom: SMALL
    }}>
        <Text style={{
            color: '#989898',
            textAlign: 'center'
        }}>
            (folgende Einstellungen werden für alle Verbindungen übernommen –
            nicht ausschließlich für diese Suche)
        </Text>
    </View>
    <View style={{
        backgroundColor: '#EFEFEF',
        padding: SMALL
    }}>
        <Text style={{
            color: '#CC0000'
        }}>
            Verkehrsmittelpräferenzen
        </Text>
    </View>
    {vehiclePrefsToRender}

    <View style={{
        backgroundColor: '#EFEFEF',
        padding: SMALL
    }}>
        <Text style={{
            color: '#CC0000'
        }}>
            Barrierefreiheit
        </Text>
    </View>
    {disabiltyPrefsToRender}
</ScrollView>
```

IV. Fazit

Zusammenfassend lässt sich sagen, dass die App des HVV gar nicht so nutzerunfreundlich ist, wie zunächst erwartet wurde, auch wenn einige Probleme bestanden.

Diese Probleme konnten identifiziert, behoben und die Lösungen entsprechend dargestellt werden.

Optimaler wäre es jedoch gewesen, die Arbeit nur auf den UX-Teil zu beschränken, um Verhalte tiefer betrachten zu können. Das hätte zum einen den Vorteil gehabt, UX-Tests ausführen zu können und enger mit dem Nutzer zu arbeiten und zum anderen manche Sachverhalte klarer und genauer darstellen zu können.

Der technische Teil hätte herausgelöst werden können, um ihn in einer separaten Arbeit aufzugreifen. Dadurch hätten die Themen näher beschrieben werden können und eine lauffähige App wäre das Resultat gewesen.

Während der Arbeit wurde nichtsdestotrotz viel dazugelernt. Unter anderem, dass UX von Kühlschrank bis App reicht und nicht auf digitale Produkte beschränkt ist und die Umwelt dadurch etwas anders zu betrachten.

Abbildungsverzeichnis

Literaturverzeichnis

Hartson, R. | Pyla, P. S. (2012). *The UX Book – Process and Guidelines for Ensuring a Quality User Experience* (1. Auflage)
Waltham: Elsevier.

Krug, S. (2014). *Don't make me think, revisited – A Common Sense Approach to Web Usability* (1. Auflage)
San Francisco: New Riders.

Norman, D. (2016). *The Design of Everyday Things* (überarbeitete und erweiterte Auflage)
München: Verlag Franz Vahlen GmbH.

Semler, J. (2016). *App-Design – Alles zu Gestaltung, Usability und User Experience*
Bonn: Rheinwerk Verlag GmbH.

Internetverzeichnis

2017, „HVV-App ab heute rundum erneuert – jetzt mit Monatskartenverkauf“, https://www.nahverkehrhamburg.de/hvv-app-ab-heute-rundum-erneuert-jetzt-mit-monatskartenverkauf-8959/, letzter Aufruf: 26.12.2018 – 18:02 Uhr

„TOP 10 HAMBURG SEHENSWÜRDIGKEITEN“, https://www.hamburg.de/sehenswuerdigkeiten/, letzter Aufruf: 12.07.2019 – 13:40 Uhr

„Top 10 Sehenswürdigkeiten in Berlin“, https://www.visitberlin.de/de/top-10-sehenswuerdigkeiten-berlin, letzter Aufruf: 12.07.2019 – 13:40 Uhr

„Sehenswertes in Köln“, https://www.koeln.de/tourismus/sehenswertes, letzter Aufruf: 12.07.2019 – 13:40 Uhr

„Die Top 20 Sehenswürdigkeiten in München“, https://www.muenchen.de/sehenswuerdigkeiten/bildergalerien/stadtrundfahrten-touren/top-20-sehenswuerdigkeiten.html, letzter Aufruf: 12.07.2019 – 13:40 Uhr

„Software Library“, https://www.techopedia.com/definition/3828/software-library, letzter Aufruf: 24.06.2019 – 12:13 Uhr

2018, Andrea Papp, „The history of React.js on a timeline”, https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/, letzter Aufruf: 24.06.2019 – 13:06 Uhr

https://reactjs.org/docs/introducing-jsx.html, letzter Aufruf: 24.06.2019 – 15:16 Uhr

Juli 2018, David Jöch, https://medium.com/@Zwenza/functional-vs-class-components-in-react-231e3fbd7108, letzter Aufruf: 24.06.2019 – 16: 15 Uhr

Juni 2018, Braus, https://medium.com/braus-blog/airbnb-is-dropping-react-js-should-you-too-dcbff36def5c, letzter Aufruf: 24.06.3029 – 11:42 Uhr

https://github.com/reduxjs/redux/releases/tag/v0.2.0, letzter Aufruf: 24.06.2019 – 16:01 Uhr

https://redux.js.org/, letzter Aufruf: 24.06.2019 – 12:24 Uhr

März 2017, Sherif Mansour, https://medium.com/@sherifmansour/how-weve-destroyed-user-stories-8b36120645c6, letzter Aufruf: 21.08.2019 – 12:28 Uhr

Dezember 2018, Maria Myre, https://zapier.com/blog/best-wireframe-tools/, letzter Aufruf: 19.08.2019 – 10:12 Uhr

März 2018, Martin Hochel, https://medium.com/@martin_hotell/interface-vs-type-alias-in-typescript-2-7-2a8f1777af4c, letzter Aufruf: 14.08.2019 – 11:30 Uhr

https://react-redux.js.org/introduction/basic-tutorial, letzter Aufruf: 14.08.2019 – 10:45 Uhr